CHAPTER 8

Developing Rich User Interfaces for BPM with ADF
Most business processes have a human interaction aspect to them. It is important to provide intuitive and easy-to-use user interfaces to the people interacting with Human Tasks to facilitate adoption, lower learning costs, and improve productivity. Oracle BPM 11g sits on top of Oracle Application Development Framework, which is a very powerful and capable development framework that includes rich user interface capabilities. Correct usage of its capabilities leads to significant benefits, while requiring little incremental investment in learning and development. Since this chapter covers some topics in significant detail, readers may prefer to skim some sections (specifically the code snippets), and return to them later as needed. At the same time, this chapter cannot claim to do justice to the breadth and depth of capabilities offered by ADF; interested readers may pursue the pointers provided at the end of this chapter for further learning.

Overview of Oracle ADF

Oracle Application Development Framework (ADF) is an end-to-end application development framework and is built on the principles of model-view-controller (MVC) architecture. It sits on top of the standard Java Platform, Enterprise Edition (Java EE), and leverages this standard. Oracle ADF is pre-integrated with Oracle BPM Suite 11g for the generation and development of task forms, which are the user interfaces leveraged by process participants to start a process or perform their activity in a process. However, Oracle ADF is a very capable and deep framework and is used to build all of Oracle’s middleware products, including BPM Suite 11g, and Fusion Applications. ADF can also be used by customers to build their own custom applications.

Because Oracle ADF is used extensively and exclusively within Oracle for building Fusion Applications and Middleware, it implements the best practices and patterns encountered in building applications. The repository of scenarios, use cases, and requirements it draws upon, and provides built-in support for, is very broad and deep. For example, it not only simplifies data access but also provides out-of-the-box behavior such as master-detail synchronization (automatically changing displayed records for child or detail elements when parent or master record selection is changed), query-by-example (QBE) (querying for records by providing patterns for one or more fields), and transaction management.

The main building blocks of Oracle ADF are shown in Figure 8-1.
ADF Faces Rich Client Framework (View)

ADF Faces Rich Client (ADF Faces for short) is the default view layer for ADF. It comprises more than 150 rich JSF (Java Server Faces) components with built-in AJAX (Asynchronous JavaScript and XML) functionality. Components include hierarchical tables, trees, menus, dialogs, accordions, tabs, and dividers. ADF Faces also includes ADF Data Visualization components for rendering dynamic charts, graphs, gauges, Gantt charts, and other graphics.

In addition to a rich set of visual components, ADF Faces provides built-in advanced functionality including drag-and-drop framework, dialog and pop-up framework, and active-data push to automatically push updated data to user dashboards as it becomes available on the server. ADF Faces also promotes reuse with features including templates and declarative components. Plus, it supports customization and skinning, along with internationalization and accessibility and rendering on multiple devices. In addition to ADF Faces, Oracle ADF also supports Apache MyFaces Trinidad, Swing, ADF Mobile, and MS Excel (via Desktop Integration) as view layer technologies.
ADF Mobile
The ADF Mobile framework enables extension of an ADF application to mobile devices, reusing all the business services and other concepts.

The ADF Mobile Browser delivers a mobile-optimized native look-and-feel user interface to mobile browsers. It adjusts to the device’s browser capabilities, leveraging advanced capabilities of the smartphone browsers, optimizing UI with browser-specific CSS, and degrading gracefully to support HTML browsers.

ADF Desktop Integration
ADF Desktop integration enables the creation of spreadsheets that are bound to the bindings exposed by the ADF model layer. This allows a user to create spreadsheets that can access and update data from, and invoke operations on, back-end services including BPM, all without writing a single line of code.

ADF Task Flows (ADF Controller)
ADF Task Flows are the controller layer for ADF applications providing an enhanced navigation and state management model on top of JSF page flow. Task Flows, unlike the basic JSF navigation mechanism, can not only handle navigation between pages, but can also include method invocations and case statements. ADF Task Flows are discussed in the section “ADF Task Flows” later in this chapter.

ADF Data Control and Bindings (Model)
ADF model provides a service abstraction called data control, which provides a consistent service and data interface on top of a variety of back-end technologies; the data control is bound to user interface elements via a binding layer, typically as part of an EL (Expression Language) expression. A number of data controls are built-in and developers can add their own. Data controls and bindings are discussed in the section “The ADF Model Layer—Data Control and Bindings” later in this chapter.
ADF Business Components (Business Services)

ADF Business Components (BC) provides a declarative fourth-generation language (4GL) for building database-centric business services. It includes support for entity objects that represent tables in the underlying database and view objects that represent query specifications and that return collection of entities in response to a query. ADF-BC supports declarative specification of properties for attribute, validation logic, list-of-values (LOV), and calculated fields. While ADF-BC enables most common requirements in a declarative fashion, it also provides hooks for developers to extend or customize the functionality with Java. ADF Business Components are discussed in the section “ADF Business Components” later in this chapter.

Some Basics

Before diving further into ADF, it is useful to review a couple of basic JSF concepts that are fundamental to further discussion.

Expression Language

In this chapter, expressions of the form #{...} will be frequently encountered. These are JSF Expression Language (EL) expressions. Here are a few examples to illustrate JSF EL:

- #{myBean.myVar}  myVar property (could have been a method as well) of myBean
- #{bindings}  The ADF bindings object
- #{bindings.title.inputValue}  The inputValue attribute of a binding named title
- #{!bindings.Delete3.enabled}  If enabled attribute of a binding Delete3 is false
- #{a.b != “test”}  If a.b is not equal to test
- #{a.b == “test” ? val1 : val2}  If a.b is equal to test, then val1, else val2
NOTE

In addition to #{ } JSF EL also continues to support the JSP syntax of ${ }; the former uses deferred evaluation (that is, the JSF controller evaluates the expression at the appropriate stage in the page’s life cycle) and the latter uses immediate evaluation (that is, the JSP engine immediately parses and evaluates the expression), and should be used only for read-only access.


Managed Beans

Throughout this chapter, Java code snippets will be provided and the term managed bean will be used. ADF and JSF applications support the concept of managed beans—beans whose life cycle is managed by ADF/JSF. The bean is a Java class that allows access to its persistent state through setters and getters. A managed bean is registered with ADF as part of either the bounded task flow definition or the adfc-config.xml file from the Managed Beans vertical tab within the Overview horizontal tab within JDeveloper’s task flow editor.

A scope needs to be specified for the managed bean that governs its life cycle; scopes help control the amount of objects kept in the memory. The rule of thumb is to use the smallest possible scope for managed beans. A bean that does not hold any state or that does not need to persist its state across requests should be set to requestScope, which is a standard servlet scope. However, in ADF Faces partial requests (for AJAX behavior; discussed later in this chapter within section “ADF and JSF Life Cycle”) spawn a request cycle, and if the bean’s state is to be persisted across partial requests viewScope should be used. For beans that should save state across the task flow pageFlow scope is appropriate. Other scopes with broader scope including application and session are rarely needed when using ADF for BPM task forms and should be avoided.
Properties exposed by a bean can be initialized in the managed bean by adding managed properties corresponding to the bean properties. For more information on the object scopes, please see section “4.6 Object Scope Lifecycles” within the “Fusion Developer’s Guide.”

**BPM Task Forms**

BPM task forms are the user interfaces that end users use to initiate processes as well as perform activities in the process. A BPM task form typically has a form for viewing and editing data. It usually includes the ability to add comments and attachments. Most importantly, it gives end users the ability to take actions (e.g., approve, reject, escalate, and so on) on their work items. By default, Oracle BPM 11g leverages ADF for task forms. BPM 11g includes form-generation capabilities that can generate fully deployment-ready forms that can be used without any further edits. These generated forms can also be further edited or extended to meet even the most sophisticated requirements.

**Task Form Generation**

Oracle BPM 11g has three flavors of form generation.

- **One Click**  A ready-to-deploy task form is automatically generated in a new project; no options are offered.

- **Wizard Driven**  Offers more options, including the choice of a template and multi-row-column layout; a ready-to-deploy task form is generated in a new project.

- **Task Flow Based on Human Task**  Only a task flow and task data control are generated; the page needs to be created using ADF designer’s drag-and-drop form creation capabilities leveraging the generated Human Task data control and possibly other data controls. This option generates the artifacts inside an existing project.
In all three options, a task data control and a bounded task flow are generated. In the first two options, a complete page corresponding to the view activity in the task flow is also generated. In the third option, a page is not generated; however, the Human Task drop handler (as described in the next section and shown in Figure 8-3, Human Task drop handler is the Human Task specific menu presented when the Task element is dragged and dropped on the page from a Human Task data control) can be used to create a complete page. Therefore, even the third option is not much extra work. While the first two options are great for those who would be comfortable with the auto-generated form and do not want to get inside ADF, the third option is better for anyone with even minimal comfort with ADF as it provides more flexibility on the structuring of projects and so on.

BPM Data Control

The most important artifact generated as part of any of the form generation options is the Human Task data control. The generated Human Task data control is named <ui project name>.<task name>. A sample Human Task data control displayed inside JDeveloper’s Application Navigator window is shown in Figure 8-2.

As can be seen in Figure 8-2, the TableExUI_EnterDocTask task data control has an element Task that is a child node of Data Control >>getTaskDetails <<Return; this is the effective root node for working with task data controls. Some of the interesting elements contained within Task are:

- **Payload**  As can be seen in Figure 8-2, Task has a child element Payload; each parameter of the task is a child of the Payload element. The DocElement and the Section elements seen within the Payload node in Figure 8-2 are based on the schema used to define the variables in the payload (the concept of payload variables is discussed within the previous chapter on Advanced Human Tasks); the schema used in this example is shown in Figure 8-23.

- **Task Attributes**  Attributes of the Task element contain task summary information such as task title, due date, priority, and others. These attributes, as well as the system attributes discussed next, are the same for every task.

- **System Attributes**  Detailed information about the task; has child elements for assignee (that is, the users, groups, or roles that the task is assigned to), task history, and other information.
Chapter 8: Developing Rich User Interfaces for BPM with ADF

FIGURE 8-2. Human Task data control
Human Task Drop Handlers

In addition to standard ADF drop handlers, when the Task element is dragged and dropped on to the page canvas, Human Task specific drop handlers are available that offer options of creating task headers, task actions, comments, and so on, as shown in Figure 8-3.

NOTE

The parameters that are prompted for when the Task element of a task data control is dropped onto a view can be left unspecified.

Refreshing Task Data Control

If the task definition changes within the BPM application, the task data control can be refreshed to reflect the new changes. If the changes are...
additive, existing bindings are not impacted and new elements can be added by dragging and dropping them. If a bound element is deleted, then the binding of the UI elements will need to be updated or the UI elements need to be deleted.

Unlike ADF-BC and other data controls, the Human Task data control is not refreshed by the refresh icon in the data controls panel. Instead, to refresh Human Task data control, right-click the data control and select Edit Definition, as shown in Figure 8-4.

**Data Control Element Definitions**
Under Application Sources inside a package with the same name as the data control, XML files containing the definitions of task data control elements are generated. In particular, for each parameter type, a `typeName.xml` file is

![FIGURE 8-4. Refreshing task data control](image)
generated. This XML file can be edited to add validation rules as shown in Figure 8-5; the additions made here are not lost when the data control is refreshed (as discussed in the preceding paragraph).

**Process Initiation Alternative**

The default option for a user-initiated process is to use an initiator task and generate a Task Form for it, as discussed earlier. The benefits of this approach are:

- BPM takes care of exposing the initiation link only to authorized users.
- The process can have initialization logic to personalize the form or populate it with appropriate data.
An alternative option is to separate the initiation UI from the process and have the initiation UI launch the process either via a service invocation or via an event (events are discussed later in this chapter in the section “ADF Business Components”). In this approach, a custom mechanism for exposing the initiation link to authorized users needs to be provided because swim-lane roles are not available outside the process. The benefits of this approach are:

- The initiation UI can be built into a page, which may provide easier access.
- If the initiation is abandoned, no process is created.

**Tips and Recommendations**

This section presents a few tips and suggestions to improve the productivity and overall experience of working with ADF, especially in the context of BPM.

**Working with ADF Page Editor**

While the ADF page editor is a visually intuitive development environment, a couple of tips provided here can improve the overall experience.

**Design This Container**

Ancillary information on BPM Task forms, such as task header, comments, and attachments, can be distracting when working with the main form content. The design canvas can be focused on the form content by selecting the appropriate element in the structure window (`af:showDetailHeader: #{resource.CONTENTS}` if using the auto-generated form), using right-clicking it, and then selecting Design this Container. This causes the design canvas to display only the selected element and its descendents; the normal view can be restored by clicking either the small blue box with two left arrows or anywhere in the canvas outside of the element’s design area (the tooltip will show Click to edit main page in both cases).
Leveraging the Structure Window
It is sometimes tricky to find the correct drag-and-drop insertion point inside nested layout elements in the design window. The Structure window offers a much more precise mechanism for dragging and dropping content into the form as well as for repositioning elements in the form. When adding from the component palette, drag and drop is not even needed; just selecting a node in the structure window and clicking the component palette item adds it inside the selected node. Also, right-clicking a structure window node offers insertion choices including Insert Inside, Insert Before, and Insert After. Cut-copy-paste is also available in the structure window and can be used to add similar items very easily.

Debugging and Running in Integrated Server
Although BPM and SOA 11g cannot be run inside the Integrated WebLogic Server (the one running inside JDeveloper), BPM forms can be debugged and run inside the Integrated Server because BPM tasklists and task forms can be deployed on non-BPM domains using foreign JNDIs (and other mechanisms).

To enable this feature, set up the JDeveloper integrated server with the required foreign JNDIs and libraries (a script to do this, along with instructions, can be downloaded from this book’s web site). To debug or run a form inside the integrated server, right-click the bounded task form XML file in the application navigator window and select Debug or Run. The page that is launched should be ignored; instead, if the environment has been set up correctly, the URL for the form on the BPM server is updated to point to the integrated server. Thus, opening a task from the BPM Workspace will use the integrated server. In addition to debug features such as setting up break points and stepping through code, debugging or running in the integrated server environment provides another powerful benefit—any changes to the page’s metadata are picked up on form reload, without needing any compilation or deployment. The automatic reload works only for page metadata changes. Any change in bindings or managed beans requires compilation and deployment (that is, executing the debug or run command again). Once the debug/run session is over, deploying the form to the BPM server will update the form URL to point to the form deployed on the BPM server.
JDeveloper also allows remote debugging—that is, debugging the form running on the BPM server. However, whenever possible, the integrated server option is recommended as the alternative requires running the whole of BPM server in debug mode and this may degrade performance.

Form Reuse

Although the primary unit of reuse in Oracle BPM is a task file, in some situations there are multiple task files with the same payload data. Different task files may exist due to differences in assignment and other policies. In such cases, the same form can be reused for the different tasks. To automatically register the same task flow for multiple tasks, add additional hwTaskFlow entries to the hwTaskFlow.xml file in the Application Sources folder, as shown in the following snippet (three tasks are pointing to same task flow):

```
<hwTaskFlow>
  <WorkflowName>FRETask1</WorkflowName>
  <TaskDefinitionNamespace>.../FormResuseEx/FRETask1</TaskDefinitionNamespace>
  <TaskFlowId>FRETask2_TaskFlow</TaskFlowId>
  <TaskFlowFileName>WEB-INF/FRETask2_TaskFlow.xml</TaskFlowFileName>
</hwTaskFlow>

<hwTaskFlow>
  <WorkflowName>FRETask2</WorkflowName>
  <TaskDefinitionNamespace>.../FormResuseEx/FRETask2</TaskDefinitionNamespace>
  <TaskFlowId>FRETask2_TaskFlow</TaskFlowId>
  <TaskFlowFileName>WEB-INF/FRETask2_TaskFlow.xml</TaskFlowFileName>
</hwTaskFlow>

<hwTaskFlow>
  <WorkflowName>FRETask3</WorkflowName>
  <TaskDefinitionNamespace>.../FormResuseEx/FRETask3</TaskDefinitionNamespace>
  <TaskFlowId>FRETask2_TaskFlow</TaskFlowId>
  <TaskFlowFileName>WEB-INF/FRETask2_TaskFlow.xml</TaskFlowFileName>
</hwTaskFlow>
```
Differences in actions can also be handled, if needed, by defining the union of actions across all tasks for which the form is to be reused as the set of actions for the task from which the data control is generated, and then configuring the Access settings for this task to disable the extra actions.

Parallel Development of the BPM Process and Task Form

In many cases, different developers work on the BPM process and its related forms. In such cases, development can be parallelized by the BPM developer providing a proxy process application. This application should have a task corresponding to each task in the real process application, with the same payload and actions, along with processes that just create tasks (that is, have no real process flow), initializing data, if needed. The UI developer(s) can then develop and work against these proxy tasks—not only isolated from changes in the process, but also with an easy path to getting to the task (that is, with no need to play the whole scenario). If the payload or action for the actual task changes, the process developer should make the changes to the corresponding task in the proxy task, and the UI developer can update the task data control. The same form that the UI developer develops against the proxy task can be used for the real task as well by just adding an additional entry in the `hwtaskflow.xml` file as described in the subsection on Form Reuse.

Another benefit of the preceding approach is that the UI developers don’t necessarily need to have a BPM server instance of their own; they can use the integrated server, as mentioned in the subsection on Debugging and Running in Integrated Server, and access the proxy tasks on the BPM developer’s BPM server instance.

The placeholder data control (described later in this chapter within the section Data Control) also has a useful role to play in the early phase of development before the data shape has been defined. However, shift to the task data control as early as possible to avoid surprises—the capabilities of the task data control are sometimes different from the placeholder’s.
Deployment

BPM deployment offers the option of deploying the forms (in a separate application) along with the BPM application. The author recommends only using this option when auto-generated forms will be used as is. The UI and the process go through separate life cycles, and typically the UI may iterate a lot more than the process. Also, as recommended earlier, the integrated server should be used for the UI during development. Tying the process application and UI application together is therefore not advisable.

Forms should be deployed instead by right-clicking the application node in the Application Navigator (not the project node but the application node), selecting deploy, and then selecting a deployment profile. Deployment profiles for the application can be created, updated, and deleted by right-clicking the application node in the Application Navigator, selecting Application Properties, and then choosing Deployment in the properties window.

For application management reasons, all forms for a process application are usually deployed as one application. However, for iterative development, it is much better to deploy one form at a time. When using the integrated server during iterations, this is not an issue because the debug or run command will deploy and run only one project, so the deployment profile (that deploys to the BPM server) can be configured to deploy a single application. If using BPM server during UI iterations, it may be useful to have multiple deployment profiles: one with all forms in one application, and one each for every project by itself. If doing so, Web Context Root conflicts should be avoided by using different project deployment profiles with different web context root settings (just as for application deployment profiles, this is available from the project properties). At the end of iterations, the end-state deployment profile (one with all forms in one) should be deployed and tested to avoid surprises (such as class path conflicts).

NOTE

If deploying multiple projects in one application, each project should have a different view prefix to ensure that files with common names are in unique packages.
Forms for E-mail

BPM e-mail notifications include (by default, but can be turned off if needed) a static HTML version of the form. However, since e-mail clients have different HTML and CSS capabilities, it is advisable to provide a form tailored for e-mail instead of relying on the web form to be rendered in e-mail as well. This is even more important if the web form is a multipage task flow.

To provide a different form for e-mail, a router activity can be used to route to an e-mail–specific view based on the test: #{pageFlowScope.bpmClientType == 'notificationClient'}, as shown in Figure 8-6. The router activity should be made the default activity for the task flow. Within the e-mail specific page the Human Task drop handler, Task Details for Email, may be used to generate a form targeted at e-mail clients.

FIGURE 8-6. An e-mail-specific form in task flow
Task Forms in Non-ADF Technology

While ADF is the default technology choice for developing BPM forms with many benefits, both from the perspective of ADF’s features as well as the integration between BPM and ADF, customers may have standardized on a different framework. BPM provides APIs to support worklists and task forms to be built on a different framework. Chapter 31, “Building a Custom Worklist Client” and Chapter 32, “Introduction to Human Workflow Services” of the “Oracle Fusion Middleware Developer’s Guide for Oracle SOA Suite,” which is part of Oracle’s product documentation, provide good information in this regard. Also, you can find samples and links to samples for this at the BPM’s site at the Oracle Technology Network.

The ADF Model Layer—Data Controls and Bindings

Bindings were one of the most powerful new features in JSF. They enabled separation of code from the view definition, allowing access to values from backing bean properties or resource bundles via simple expressions of the form: #{myBean.myAttr}.

Oracle ADF takes the concept further, and using a combination of Data Control and Bindings provides a unified access to data and methods exposed by underlying business services. As shown in Figure 8-7, the Data Controls are framework adapters working with the specifics of different business service technologies. The Bindings layer sits on top of the Data Controls and provides the interfaces needed by the view layer in the application.

Data Control

A data control consists of two pieces: the Java classes implementing the business service adapter, and the metadata, which includes its graphical presentation, describing the business service attributes, collections, and
methods. While it’s possible for developers to write their own data control implementation, in this chapter we will limit our discussion to the built-in data controls and focus on the metadata aspect. The following are some of the built-in data controls:

- **Task Data Control** The task data control works with Human Tasks in BPM (and is sometimes referred to as the BPM Data Control). This data control is discussed in detail earlier in this chapter in the section “BPM Task Forms.”

- **BAM Data Control** The BAM Data Control lets ADF interface with queries on top of BAM data objects. While Oracle BAM provides
dashboards that can be designed by business users in web-based BAM studio, the BAM Data Control enables the integration of BAM events and dashboards embedded in ADF pages.

- **ADF-BC Data Control** The ADF-BC Data Control provides access to ADF-BC views exposed through an ADF-BC module. Unlike other data controls, this data control does not need to be explicitly created and there is no additional XML file used for this data control’s metadata. An ADF-BC data control is automatically available for every ADF-BC module in the application. As discussed later in this chapter, ADF-BC is useful in many BPM scenarios. Therefore, for BPM developers, this is usually the second most used data control after the task data control.

- **Placeholder Data Control** As the name indicates, a placeholder data control is a stand-in data control that works without any backing business service layer and is useful for prototyping. Placeholder data controls can be used to create page layouts and page flows, including all view behavior such as validations and drag-and-drop without using a business service-bound data control. Placeholder data controls can also include sample data that makes it easy to visualize the page, especially when data-visualization components like charts and graphs are used.

A placeholder data control can be created using the *New* gallery, which can be launched by using the *File / New* menu. In the *New* gallery, *Place Holder Data Control* is available as an *Item* when in the *Categories* tree the *Data Controls* child node of the *Business Tier* node is selected. Once the data control is created, its type is defined by right-clicking the data control node in the *Data Controls* accordion panel in the *Application Navigator* window and selecting *Create Placeholder Data Type*, which launches the placeholder data type editor shown in Figure 8-8.
The placeholder data control supports String, Boolean, Date, and Number types for attributes. The Sample Data tab of the placeholder data type editor allows entering or importing sample data as a CSV file. Instead of creating the type definition manually, the attributes can be created based on an imported CSV file—attributes of type String are created based on the first row in the CSV file.

The placeholder data control supports hierarchical types. To create a child element, right-click a placeholder type node in the Data Controls accordion panel and select Create Placeholder Data Type (as in the preceding paragraph). The parent–child relationship is maintained using the first attribute of the parent as the foreign key.
in the child. This means that more than two levels of parent–child can be created but cannot be seeded with sample data correctly (this limitation is planned to be addressed in future releases).

- **Web Service Data Control**  A web-service data control provides a quick and zero-code option for invoking web services. It is created from the New gallery (the same location as the placeholder) based on a WSDL. However, for more robust usage, using WS Proxy and creating a bean data control from it is a better option because it provides more flexibility in exception handling and service invocation.

- **Bean Data Control**  Data control based on a Java bean can be created by right-clicking the Java file in the Application Navigator window and selecting Create Data Control.

- **EJB Data Control**  Works with an EJB session bean.

- **URL Data Control**  URL Data Control allows the accessing of CSV and XML files from a URL. This is useful when working with servlets and other providers that return simple text content such as an RSS feed. This type of data control is read-only.

- **JMX Data Control**  Works with JMX MBeans from an MBean server.

### Bindings

As shown in Figure 8-7, *Bindings* work in conjunction with data controls, gluing UI components to the underlying data controls and using standard Expression Language (EL), independent of the type of data control.

When UI components are created either using drag-and-drop from a data control or using BPM form generation wizards, the appropriate bindings are automatically created and wired to the component’s properties. Therefore, the only thing about bindings that many developers need to know is how to use them in expressions—for example, when making one input field conditionally dependent on another. Bindings are accessed via expression language using the form: #{bindings.bindingAttribute.property}—for example #{bindings.title.inputValue}, #{bindings.title.hints.label}—where *title* is the name of a bound attribute. Expression Builder can be used to edit expressions
using bindings; bindings are available within the bindings child node of the ADF Bindings node, as shown in Figure 8-9. The expression builder can be launched from within the Property Inspector window by clicking the down-arrow button on the right side of the property-value field for most properties and then selecting Expression Builder from the pop-up menu.

The bindings for a page can be accessed from the Bindings subtab of the page editor, as shown in Figure 8-10. While JDeveloper automatically adds and deletes bindings, this Bindings subtab can be used to manually add, delete, or edit bindings.

FIGURE 8-9. Bindings in expression builder
In Figure 8-10, in addition to the Data Control, we see Bindings and Executables; the bindings are linked to the data controls via executables. The other thing we see in Figure 8-10 is that different bindings have different icons, indicating that they are of different types. There are different types of bindings because there are different types of UI components. The different types of bindings are:

- **Attribute Binding**  Binds a single attribute value; typically used in input components or output text fields.

- **Tree Binding**  Binds a collection of data; typically used for tables, trees, tree-tables, and hierarchy viewer components. This is discussed further in the section “Collection Components” later in this chapter.
**Action Binding**  This binds operations provided by data controls. Human Task data control exposes Create and Delete operations for collection subelements. ADF-BC data controls expose a variety of operations including Execute, Find, Create, CreateInsert, and Delete for collection subelements. Typically, such a binding will be bound to a button or a link.

**Method Binding**  Binds methods exposed by a data control. The Human Task data control exposes methods including Update, UpdateAction, Reassign, Escalate, and Claim, as well as methods for actions such as OK, APPROVE, and REJECT. Just as in the case of action binding, typically these will be bound to a button or a link.

**List Binding**  Used for defining the contents of data-bound list components, such as radio groups and drop-down lists. This is discussed further in the section “Select Components” later in this chapter.

**ListofValues Binding**  Binds data to list-of-values (LOV) components. This works primarily with ADF-BC components that have model-driven LOVs. For more information on this option, please see the discussion on List of Values in the section “View Objects” later in this chapter.

**Executables**

The other element seen in Figure 8-10 is Executables. The different flavors of executables are:

**Iterators**  In most scenarios, the executable used will be an iterator, which links other bindings to a data control via attributes and collections. Iterators enable working with collections. For example, if a Human Task has a Section child of a DocElement, the iterator for Section is:

```xml
<accessorIterator MasterBinding="DocElementIterator"
Binds="section" RangeSize="25" DataControl="TableExUI_EnterDocTask"
BeanClass="TableExUI_
```
Chapter 8: Developing Rich User Interfaces for BPM with ADF

As can be seen in the preceding snippet, the iterator points to the parent’s iterator. The attributes for the Section element are exposed through the iterator defined in the preceding snippet as illustrated in the following underlying XML snippet:

```xml
<tree IterBinding="sectionIterator" id="section">
  <nodeDefinition DefName="TableExUI_EnterDocTask.SectionType" Name="section0">
    <AttrNames>
      <Item Value="title"/>
      <Item Value="type"/>
      <Item Value="subtype"/>
    </AttrNames>
  </nodeDefinition>
</tree>
```

- **Variables** Enables local variables defined as part of bindings; typically used for arguments to a method call.
- **Invoke Action** Enables invoking an action on another executable. For example, if an ADF-BC component exposes the `ExecuteWithParams` operation, the operation can be invoked using an invoke action executable. This will be used later in this chapter to implement dependent lists in the section “Selection Components.”

Accessing Bindings from Java

While the declarative bindings enable a wide variety of scenarios in a zero-code manner, there are scenarios where some logic needs to be programmed in backing Java beans. Scenarios include handling collection drag-and-drops, copying values from one data control to another, and advanced custom behavior. The bindings can be accessed from Java code. Some of the important classes to be aware of are:

- **DCBindingContainer and BindingContainer** The `oracle.adf.model.binding.DCBindingContainer` class corresponds to the bindings
element in the declarative bindings and provides access to bindings contained in a page. It implements the interface oracle.binding.BindingContainer. The binding container can be accessed in the backing bean, as shown in the following code snippet:

```java
import oracle.adf.model.BindingContext;
import oracle.binding.BindingContainer;
import oracle.adf.model.binding.DCBindingContainer;
...
BindingContext bctx = BindingContext.getCurrent();
BindingContainer bindings = bctx.getCurrentBindingsEntry();
DCBindingContainer bindingsImpl = (DCBindingContainer) bindings;
```

The binding container may also be injected into the backing bean by having a property bindings in the bean of the type BindingContainer, having getters and setters for it, and then passing #{bindings} as a managed property for the backing bean, as shown in Figure 8-11.

**DCIteratorBinding** The oracle.adf.model.binding.DCIteratorBinding is used to access the iterator binding. Typical scenarios include accessing the iterator binding to get or select the current row.

**FIGURE 8-11. Injecting bindings to managed bean**
In this chapter, we will see usage of iterator binding in programmatically selecting a row in a table and in handling collection drag-and-drop. The following snippet shows how an iterator binding can be accessed from a binding container by name.

```java
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.jbo.Row;
...
DCIteratorBinding iterBinding =
bindingsImpl.findIteratorBinding("sectionIterator")
```

As with bindings, the iterator binding can also be injected into the managed bean via managed properties, instead of being accessed by name. Also, the iterator binding can be retrieved from a control binding (DCControlBinding and its subclasses).

- **DCControlBinding** The `oracle.adf.model.binding.DCControlBinding` class is the base class for all component binding classes. It provides methods including `getRowIterator` and `getCurrentRow` to get the current `RowIterator` and `Row`, and `getDCIteratorBinding` to get the iterator binding. The subclasses for different flavors of bindings are:
  - **Attribute Binding** `oracle.jbo.ui.binder.JUCtrlAttrsBinding`, which implements the interface `oracle.binding.AttributeBinding`.
  - **Tree** `oracle.jbo.ui.binder.JUCtrlHierBinding`; nodes within the tree are of subclass `oracle.jbo.ui.binder.JUCtrlHierNodeBinding`.
  - **Action and Method** `oracle.jbo.ui.binder.JUCtrlActionBinding`, which implements the interface `oracle.binding.OperationBinding`. If parameters need to be passed to the method from Java, they can be passed by putting `<param-name, param-value>` pair to the hash map returned by the `getParamsMap` method.
  - **List** `oracle.jbo.ui.binder.JUCtrlListBinding`.
ADF Faces

As mentioned in the introduction, ADF Faces is the view layer of the ADF stack and provides a large number of AJAX-ready components, browser-safe layouts, drag-and-drop functionality, and other behavior. While we start this section with a discussion on layout, readers who wish to simply extend generated forms may prefer to jump to the section “Data Forms.”

Layout Components

In older UI technologies, developers would position elements on a page. However, such positioned page layouts are not ideal for today’s web-based UIs, where not only different users have different screen sizes, but they can also access the page from different devices. Therefore, the new generation UI technologies such as ADF and JSF provide page designs based on relative positioning through the nesting of layout containers that adjust nicely to different devices and screen sizes.

ADF has a very rich set of layout components (containers), but the choice and power available, along with the accompanying rules and constraints, may seem intimidating at first. Readers should note that ADF is a full UI development framework and many of the capabilities included are more targeted toward stand-alone UI pages. For BPM forms, which don’t typically need to include application facets such as the navigation bar and so on, many developers can get away with simply understanding and using a single layout component: the Panel Group Layout. However, some BPM form developers may want to make their form layout richer. A few example scenarios include:

- Providing summary information like customer profile or spending report dashboard on the left side of the form, such that this left-content does not scroll with the main-content.
- Using tabs to organize and arrange content (please also see the section “ADF Task Flows” in this chapter because task flows also provide a mechanism for arranging content in multiple pages).
- Ensuring that task header information—including title, due date, and actions—occupies the top of the available view area, and task footer
information like comments and attachments occupy the bottom of the available view area, while the rest of the form occupies the remaining center position and scrolls within it.

- Providing the ability to resize or minimize certain areas to maximize the view area available to the primary content.

In this section, we discuss some of the ADF layout components needed to achieve the scenarios just described. The depth of discussion in this section is intended to provide readers with an overview of the available options. Readers who decide to use rich layouts may refer to the additional material referenced in the section “For More Information” at the end of this chapter. The author strongly recommends that instead of building layouts on their own, developers should use the layout options provided by JDeveloper, as shown in Figure 8-12. Or they can use the templates shipped with BPM, or develop their own (as discussed earlier in the section “BPM Task Forms,” BPM form generation wizards include templates and support custom templates).

FIGURE 8-12. Creating a JSF page based on templates and quick start layouts
Stretching vs. Flowing
An important concept to understand regarding layout in ADF is that of stretching versus flowing. Flowing layout means laying out the child elements of a layout component sequentially, horizontally, or vertically, and wrapping as needed. Child element sizes are determined by their style and other attributes and content. As the name indicates, stretching involves stretching the elements to adjust the view space available.

This concept is best explained with an example. Figure 8-13 shows a flowing page and Figure 8-14 shows its stretched counterpart.
The first browser window in Figure 8-13 highlights the implication of not stretching when more view space is available. Two things can be noticed in this window:

- The Footer is placed significantly above the bottom of the browser window. This is because the table in the center has only that many rows and has not been stretched.
- Although horizontal space is available, the table in the center is truncated (as can be seen in Figure 8-14; it has more columns).

FIGURE 8-14.  *Stretched page*
NOTE
The table here is simply used as an example of a child element that may need to be stretched. The particular issue of columns being truncated in spite of available horizontal space can be addressed in the preceding example by setting the table’s property styleClass to AFStretchWidth.

The second and third browser windows in Figure 8-13 highlight the implication of not stretching when view space available is less than needed and scrolling is needed. As can be seen in these windows, scrolling scrolls the whole page and may scroll the header and footer out of the viewing area.

The first browser window in Figure 8-14 shows how a stretched layout maximizes the use of available view space. The table in the center is stretched—in this case, since there are not enough rows, there is white space—but unlike Figure 8-13, the white space is coming from the component and not the browser. Among other things, this means that the footer is at the bottom of the browser window. The second browser window in Figure 8-14 shows how stretching enables the header and footer to always be within the viewing area even when the table needs to be scrolled.

In ADF, there are two aspects of stretching: whether a component stretches its content or children and whether the component itself can be stretched or not. Table 8-1 summarizes the behavior of some of the components in this regard. Components that cannot be stretched should not be placed inside components that stretch their children; transition components, such as Panel Group Layout with the type set to scroll (or vertical), should be used to nest non-stretching components inside stretching components. As a best practice, when using stretching components, start with stretching components, and within stretching components (which may be hierarchically nested) create islands of flowing content using Panel Group Layout.
## Component Is Stretched? Stretches Its Children

<table>
<thead>
<tr>
<th>Component</th>
<th>Is Stretched?</th>
<th>Stretches Its Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel Group Layout</td>
<td>If: layout=&quot;vertical&quot;, or layout=&quot;scroll&quot;</td>
<td>Yes; provides transition between stretching and flowing components</td>
</tr>
<tr>
<td>Panel Stretch Layout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Panel Splitter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Decorative Box</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Panel Tabbed</td>
<td>If: dimensionsFrom = &quot;parent&quot;</td>
<td>If: dimensionsFrom = &quot;parent&quot;, and ShowDetailItem contains only one child, or stretchChildren=&quot;first&quot; for the ShowDetailItem</td>
</tr>
<tr>
<td>Panel Box</td>
<td>Stretched if: type=&quot;default&quot; and its parent layout component is stretching the panelBox (for example, panelDashboard) type=&quot;stretch&quot;</td>
<td>Not stretched if: type=&quot;default&quot; and its parent layout component is not stretching the panelBox (for example, panelGroupLayout) type=&quot;flow&quot;</td>
</tr>
<tr>
<td>Panel Header</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Show Detail Header</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Panel Label and Message</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Panel Window</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dialog</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**TABLE 8-1. The stretching behavior of components**
When working with stretched containers, it should be noted that the browser scroll bar cannot be relied on to scroll the content inside the container correctly. The first browser window in Figure 8-15 shows an example where there are two stretched containers (in this case, Panel Box). One of them encloses a table and the second encloses the table rows outputted as text (using Iterator and Output Text). The second box includes an “End of Content” text at the end. The second browser window shows the same example with the browser resized. As can be seen in Figure 8-15, even with the browser scroll bar all the way to the bottom, the bottom content of the second box is truncated. Since the table offers its own scroll bar, the content inside the first box behaves correctly.

The preceding example highlights that stretched containers should not contain overflowing content. The issue in this example can be resolved by using a Panel Group Layout with type set to Scroll. As shown in Figure 8-16, the Panel Group Layout offers its own scroll bar that scrolls the content correctly.
Multiple scroll bars may lead to distracting pages. Stretched layouts should avoid creating multiple horizontal containers with possibly overflowing content. It may be a good design to primarily use side columns to display content that need not be scrolled and let only the center area scroll.

The Panel Group Layout Component
As mentioned earlier, this is the layout component that every developer needs to know about. This is not only the most commonly needed component, but it is sufficient for many scenarios. This component arranges its children
in one of a few simple patterns based on its `layout` attribute. The behavior for different values of `layout` is:

- **default** Consecutive layout as defined by the browser, with wrapping allowed, and following the bidirectional layout algorithm. As a result, in the presence of mixed right-to-left and left-to-right text, contents may not display consecutively.

- **horizontal** Strictly consecutive without wrapping.

- **vertical** Vertically stacked.

- **scroll** Mostly same as vertical; the difference from vertical is that scrollbars are displayed if content overflows (this layout option is only intended for cases where the panelGroupLayout is being stretched as discussed earlier).

**TIP**
*When using horizontal layout, the `valign` attribute specifies the vertical alignment. Valid values include: middle, top, bottom, and baseline.*

As discussed earlier, this component is the ideal component for transitioning from stretched components to non-stretched components. This component is also useful for creating complex input fields like Name where multiple subfields need to be arranged. This is discussed in the section “Data Forms” in this chapter.

### The Panel Stretch Layout Component
This component is probably the most used stretch component. This component stretches the child in the center facet to fill all available space. This component also supports `top`, `bottom`, `start`, and `end` facets and stretches them as well. A common page design pattern is to have a branding bar on the top along with primary navigational links and some header information about the page, a vertical bar on the left providing utilities such as categories in the
context of shopping, a vertical bar on the right offering session and profile information like shopping cart and recent purchases, and a footer with copyright, contact, and other similar information surrounding the main content in the center. The facets of this component directly map to this type of page design.

The sizes of the surrounding facets are controlled by the `topHeight`, `bottomHeight`, `startWidth`, and `endWidth` attributes; however, if a facet is empty, it does not render. Instead of specifying absolute pixel values, percentage values may be specified to achieve a multicolumn layout. For example, the page shown in Figure 8-15 and Figure 8-16 has `startWidth` set to 50%.

The Panel Splitter Component

The Panel Splitter Component does not give the end-user any control over the sizing of the different regions. Many times, it is desired that the end-user be able to resize the different regions or even hide them, as needed. This is accomplished with the Panel Splitter, nesting them as needed.

A Panel Splitter divides a region into two parts—the first facet and the second facet—with a repositionable divider. The splitter tries to stretch the contents of both sections. A panel splitter can split horizontally or vertically based on the `orientation` attribute. Another interesting attribute for this component is `positionedFromEnd` which not only controls which side the initial position of the repositionable divider is measured from, but also which side of the divider can be collapsed—the side from which the position is measured is the side from which it can be collapsed.

The Decorative Box

The decorative box component applies a bordered look (that is, rounded corners) to its children. It also supports changing the rendered theme of its children, so it can act as a visual 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. Otherwise, it is similar to the Panel Stretch component but divides a region into only two sections: a `top` facet and a `center` facet.
The Panel Border Layout
The panelBorderLayout component can be considered the flowing counterpart of Panel Stretch Layout. This layout element lays out all of its children consecutively in its middle, and supports 12 other facets including top, bottom, left, right, start, and end. For horizontal facets, two flavors—left/right and start/end—are supported. The first is fixed left and right, whereas the second respects bidirectional-text settings and adjusts left and right accordingly.

Tabs and Accordions
The Panel Tabbed and Panel Accordion layout components can be used to display a group of contents. For each tab or accordion panel, a Show Detail Item (common) component must be added to the tabbed or accordion layout component. The content goes inside the Show Detail Item.

Headers
Often it is required to group together sections of a form and give it a header. ADF Faces components supporting this include:

- Panel Box  Used to place ancillary information on a page, offset by a certain color.
- Panel Header  Places a label and optional icon at the top of a section. This can be many levels specified by the size attribute. It can support fancy headers with its facets. It is also used for displaying ADF messages such as errors, warnings, and others.
- Show Detail Header  This is just like Panel Header but collapsible.
- Show Detail  This is collapsible too but supports a simple header (disclosed and undisclosed text attributes).

Other Layout Components
ADF Faces has many more layout components including:

- Panel Dashboard  Arranges fixed-size height children (panel boxes) into rows and columns and supports drag-and-drop reordering. This is useful for displaying a set of fixed-size dashboards, including charts and other visualization elements.
Chapter 8: Developing Rich User Interfaces for BPM with ADF 385

- **Dialog and Popup Window**  
  *Dialog* control displays its children in a dialog window; it must be a child of *Popup*, an invisible control. *Panel Window* is the same as a dialog except for its buttons.

Some other layout components including Panel Form Layout and Panel Collection are also discussed later in this chapter.

**Data Forms**

Most BPM UI scenarios involve presenting forms to process participants where they can view and input data elements for the process. Such forms would constitute the content of one or more of the sections created by other layout components in the preceding section. In many cases, the data form is the only element in the UI and, as discussed earlier, can be contained simply within a Panel Group Layout (vertical).

**Panel Form Layout**

Typically, input and output elements will be surrounded by a Panel Form Layout, which positions its children such that their labels and fields align vertically. A Panel Form Layout is automatically added when using a drop handler to create an *ADF Form* or *ADF Read Only Form*. If needed, it can also be added from the component palette.

**NOTE**

*While Panel Form Layout is technically a layout component, given its importance to data forms, it is covered in this section instead of the preceding section on layout components.*

As can be seen in Figure 8-17, Panel Form Layout supports the multicolumnar layout of its children. By default, it lays out its children in a single column because the *rows* attribute is set to a very high number. The *rows* attribute specifies the number of rows after which a new column should be started—for example, a value of 1 means that even if there are only 2 elements, use multiple columns, whereas a value of 5 means that use a second column only if there are 6 or more elements, and a third column only if there are 11 or more elements. The number of columns to
use is governed by the maxColumns attribute, which defaults to 3 for web and 2 for PDAs; however, if the Panel Form Layout is nested within another, only a single column is used.

The labelAlignment attribute specifies where the labels for the child element should be placed: start or top. The Panel Form Layout also has a footer facet where buttons and button bars should be placed.

**Group**

When elements are wrapped in multiple columns, sometimes elements that should be together for ease of understanding can get separated into different columns—for example, in the top-most box of Figure 8-17, Street and City are in different columns. This can be addressed by surrounding the components that should be logically placed together in an af:group element. The easiest way to do this is to use right-click one of the elements, select Surround With, and then choose Group from within the ADF Faces category. This adds a group element that the other elements can be dragged and dropped into, preferably within the structure window. The second box (from the top) of Figure 8-17 shows the result of grouping Street and City this way.

Within a Panel Form Layout, separators are drawn around the grouped elements if they are adjacent to other form elements.

**FIGURE 8-17.** Form examples
Panel Label and Message
In some scenarios, it is required to not only group related elements, but also provide a common label. Sometimes this is referred to as compound fields. The Panel Label and Message component can be used for creating such compound fields. In Figure 8-17, the third and the fourth box shows the result of using Panel Label and Message to create Name from First Name and Last Name and Address from Street and City.

Panel Label and Message can be created similar to group by right-clicking one of the elements, selecting Surround With, and then choosing Panel Label and Message in the ADF Faces category, and afterward dragging and dropping the rest of the elements inside the added component.

By default, Panel Label and Message will lay out the children vertically, as seen in the third box in Figure 8-17. Also, the labels for the children can be turned off by setting their attribute Simple to true, as has been done for First Name and Last Name in this example. If the labels are not turned off, they will display as per the label display for the enclosing Panel Form Layout; the result can be seen for Street and City in the third box in Figure 8-17.

Often in compound fields, it is desired to have the subfields laid out horizontally instead of vertically. This can be achieved by nesting a Panel Group Layout (Horizontal) in the Panel Label and Message. As mentioned in the preceding paragraph, if labels are not turned off, they will be positioned per the setting for the enclosing Panel Form Layout. The Address field in the fourth box in Figure 8-17 shows the resultant layout. A Panel Form Layout nested inside another panel form layout lays out its labels on top, so if it is desired to lay out the labels for the subfields on top, the subfields should be wrapped in a nested Panel Form Layout. The Name field in the fourth box in Figure 8-17 shows the resultant layout.

The components used and their nesting is shown in Figure 8-18 for the second, third, and fourth box in Figure 8-17.

Input Components
ADF has a rich collection of input components that allow users to view and edit data. Some of the commonly used input components are shown in Figure 8-19. These components can not only be used in simple forms but also within tables, trees, and others, as discussed in following sections.
As seen in Figure 8-19, some commonly used input components by data type are:

- **Text (String)**  
  Input Text is the component for entering text. If formatted text input is desired, then Rich Text Editor can be used. The Secret attribute of Input Text can be used to hide or mask the input value.
Numbers  Input Text is the most commonly used component for entering numbers as well. The built-in converter—af:convertNumber—should be used to handle percentages and currencies by setting its type attribute to percent or currency, respectively. The result of using this can be seen in the Number (Percent) and Number (Currency) elements in Figure 8-19. In addition to Input Text, components including Input Number Slider and Input Number Spinbox are available to input numbers.

Date and Times  Input Date is the component for entering dates. The attribute type of the built-in converter—af:convertDateTime—should be used to specify whether the component is used for date, datetime, or time-only input values. The values for the type attribute are date, both, and time. By default, the date and time chooser is a pop-up launched by clicking the calendar icon next to the input area; however, if an inline chooser is desired, a Choose Date component can be added to the form and the Chooseld attribute of the Input Date component should be set to the ID of the Choose
Date component (also, as discussed earlier in the section “Group” in this chapter, the two components should be grouped to keep them together). All flavors of the date component discussed here can be seen in the Dates and Times section in Figure 8-19.

- **Boolean**  The Select Boolean Checkbox component can be used for displaying Boolean values. If a set of Boolean values are desired to be displayed as radio buttons, the Select Boolean Radio component can be used.

In addition to entering values, ADF provides a number of components to select one or more values. These will be discussed later in the section “Select Components.” Also, in addition to the components covered here, ADF has a rich set of components including input file selector, color picker, carousels for pictures, and others.

The input components share many attributes in common. Some of the more useful attributes are discussed next:

- **Value**  The data element that is tied to this component. Typically of the form: #{bindings.<attr>.inputValue}. When dragging and dropping from the data control, this is automatically set.

- **Label**  The label to display. When dragging and dropping from a data control, it is typically bound to #{bindings.<attr>.hints.label}. Task data control creates the hints.label based on the attribute name—thus, for an attribute named attributeName, the label will be “Attribute Name”. To use custom labels, as well as use localized strings from resource bundles, the value of the label attribute should be changed accordingly. Some attributes related to a label are:
  - **Simple**  If set to true, the label is not displayed. This is useful when the input component is nested within a table’s column or a Panel Label and Message.
  - **Access Key** and **Label and Access Key**  An access key is a mnemonic or key stroke that navigates to the component when
the browser-specified modifier (ALT+SHIFT for Mozilla and ALT for
Internet Explorer) is used. The Label and Access Key lets a user
set the label and access key in one setting, appending “&” to the
access key character—for example, a value of Or&der will result
in a label “Order” and an access key of “d”. If needed, the access
key can be set separately from the label using the Access Key
attribute.

■ ShortDesc  The value of this attribute is displayed as a tooltip.
When bound to a data control, the value of this attribute is typically
set to #{bindings.<attr>.hints.tooltip}. However, task data control
does not provide a value for hints.tooltip, so if tooltips are needed,
the value for ShortDesc should be specified.

■ ReadOnly  This indicates whether the component is read-only or
editable. Can be an expression based on values of other data elements.

■ Required  This indicates whether the component is required. This,
similar to read-only and other attributes, can be an expression based
on values of other elements; however, dependent validations are
somewhat tricky and are discussed in the section “ADF Life Cycle”
in this chapter. When bound to a data control, the value of this
attribute is typically set to #{bindings.greeting.hints.mandatory}.
However, task data control always sets hints.mandatory to false,
so if certain elements are required, the value of Required should be
specified.

■ RequiredMessageDetail  The value of this attribute is displayed
when a required element is not entered.

■ Auto Submit  This indicates whether the value of this component
should be automatically pushed when the component loses focus
without waiting for a form submit event such as a button click. This
is typically needed when there are other components on the page
that depend on this component’s value. This is discussed in the
section “ADF Life Cycle.”
While the theme of this section is input components, the same components can be used in read-only mode as discussed earlier. Moreover, the Output Text component can be used to display read-only data. The Output Text component comes without a label, but can be wrapped in a Panel Label and Message. The two options for displaying read-only data are shown in the top-right corner (the third column with Text) of Figure 8-19.

Select Components

ADF includes a rich set of select components, some of which are seen in the bottom section of Figure 8-19. The list of choices available in the select components can be configured declaratively for most scenarios.

The easiest way to add a select component is to drag and drop the corresponding element from the data control to the ADF page and choose the select component from the options presented. This launches the List Binding editor shown in Figure 8-20. The Base Data Source specifies the data element that is updated with the selection. When dragging and dropping an element from data control, this value is automatically populated to the iterator containing the dragged-and-dropped element. Three options are available for specifying the list of choices:

- **Dynamic List** The list is obtained dynamically from the specified data source. A common scenario is to obtain the values from a database—in which case, an ADF-BC view object can be used to get the values. In addition to getting the values from a database, they can be retrieved from any other supported data service such as a Web Service, or even passed as part of the same data control as the element to be updated. As seen in Figure 8-20, multiple elements of the list data source can be mapped to elements of the base data source—all mapped elements are updated based on selection. In the example shown in Figure 8-20, selecting a value for type will not only update Attr3 but it will also update Attr1 to the value of the Id corresponding to the selected Type.

- **Fixed List** As the name indicates, a static list of values can be specified. This is useful when the list of choices is not expected to change.
Model Driven List  This option is available for ADF-BC and other data controls that support the configuration of LOV settings in the model itself. This option is not available for task data control and will not be discussed here. The ADF LOV components require model-driven lists and are therefore not applicable to page elements bound to task data controls.

FIGURE 8-20. The list binding editor
NOTE

If instead of dragging and dropping from data control, a select component is added from the component palette, it expects a List binding to be available to bind to the component. If needed, a new list binding can also be created from the Bindings tab.

Dependent Lists

As discussed earlier, when using the dynamic list option, the list binding can be configured to select multiple related values in one selection. However, there are scenarios where the choices available in a list are based on a selection in another list. For example, contract terms may be organized in categories, and for ease of selection the UI requires the user to first select a category and then the terms, the choices for which are narrowed down based on the category selection. This scenario is illustrated in Figure 8-21.

(In the example shown in Figure 8-21, the Term select component is tied to the list binding shown earlier in Figure 8-20, with the Term field bound to Attr3 and the ID field bound to Attr1 of Test.TestType.)

FIGURE 8-21. The dependent list scenario
For the list of choices to filter the data source, the list needs to support filtering. If ADF-BC is used to retrieve the list from a database, as in the earlier example, View Criteria can be defined on the ADF-BC view to do the needed filtering. The list data source (iterator) needs to be configured to get the filtered view. This is accomplished by:

- Defining an Action binding to the appropriate operation exposed by the data source. For this example, the action binding binds to the operation `ExecuteWithParams` as shown in Figure 8-22. The parameter `cat` exposed by the ADF-BC view object is bound to the data value for the `Category` field (Attr2) via the EL expression `#{bindings.Attr2.inputValue}`.

- The action binding defined above needs to be invoked by adding an `invokeAction` executable within `Executables`. The settings for this executable are shown in Figure 8-22. The `Refresh` attribute is important in getting this to work properly. If you are not sure, use `ifNeeded` as a safe choice.

**FIGURE 8-22.** Using action binding and invoke action to get a filtered list
Finally, the controlling component should be configured for auto-submit and the controlled component should have its partial-trigger set to the controlling component so the choices in the controlled component automatically refresh when a controlled component’s value changes. (The concepts of auto-submit and partial-triggers are discussed in the “ADF Life Cycle” section in this chapter.)

Collection Components

BPM scenarios often involve working with collection data elements—that is, data elements that may occur multiple times. Also, these elements may often be nested hierarchically. For example, Figure 8-23 shows a sample
data type where a document may have one or more sections and each section may have one or more paragraphs. This data type will be used to illustrate the collection components in the following discussion.

**Panel Collection**

Before diving into collection components, it is useful to mention *Panel Collection*, which is a layout component designed to wrap collection components. In addition to providing a visually appealing rendering of collection components, it offers significant additional built-in functionality related to collections, such as hiding/showing columns. This component provides facets for toolbars, menus, and status bars, and it has a built-in view menu, to which additional menu items can be added using the `viewMenu` facet. Features provided by the panel collection that are not desired can be turned off by setting the value of the attribute `featuresOff` to a space-separated list of features to be turned off—common features include `statusBar`, `viewMenu`, `wrap`, `freeze`, and `detach`.

The panel collection component is stretched by stretching containers. If placed within a flowing container, the `styleType` property for the component can be set to `AFStretchWidth` to occupy 100 percent of the parent’s width.

**Tables**

The ADF Faces Table component is the most commonly used component for tabular display. The table component provides rich behavior including row and column selection, filtering, sorting, banding, and so on. The columns of a table component can use the input and select components discussed earlier. The table component can be stretched by its parent. Figure 8-24 displays a form for the data type shown in Figure 8-23 designed using the table component. This figure will be used throughout the discussion on tables to illustrate various features. For now, readers may just review the tables and ignore the other details—the relevant details will be called out later.

A table component is typically bound to a *Tree* binding, which is usually created when the table is inserted by dragging and dropping a collection element from a data control to an ADF page. The `value` attribute of the table is typically set to the `collectionModel` property of the tree binding—an
example value is #{bindings.section.collectionModel}. The var attribute of the table specifies the name of a variable that would hold the value for each row of the table; input and other components within the table would typically access data via this variable—for example, using expression #row.bindings.title.inputValue, where row is the value for the var attribute and title is an attribute in the collection data element.

Some of the interesting attributes of the Tree component are:

- **Row Selection and Column Selection** None, single, or multiple. (The examples in this section assume the row selection is set to single.) Row selection is discussed in greater detail later in this section. Enabling column selection allows the freeze column behavior.
Chapter 8: Developing Rich User Interfaces for BPM with ADF

- **Column Stretching**  Specifies how to fill up empty horizontal space. The default is to do nothing. From a visual perspective, this should be set to either the column that is expected to have the most content or to empty, which causes an empty column to be added.

- **Grids and Banding**  *HorizontalGridVisible* and *VerticalGridVisible* specify the visibility of grid lines, and *RowBandingInterval* and *ColumnBandingInterval* specify the banding behavior. (Banding causes groups of rows and columns to be displayed with alternating background colors.)

- **Fetch Size**  ADF tables and trees do on-demand fetching of rows, bringing only enough rows from the server to fill the available view space. The Fetch Size attribute specifies how many rows are fetched in one time; a small value for this setting may result in multiple fetches.

- **Editing Mode**  A close look at Figure 8-24 reveals a subtle difference between the table used for Section and the one used for Paragraph. In the first, only the selected row (the third row) has input components enabled and the other rows are displaying as read-only, whereas in the second, all rows have input components enabled. This is specified by the *editingMode* attribute; setting it to *clickToEdit* makes only the selected row editable.

  **TIP**  
  The *clickToEdit* mode is recommended due to performance reasons; the author also finds it aesthetically more pleasing.

- **Column**  A table component wraps one or more column components. The components nested within the column components are displayed within the cells of the table. The header for a column can be specified using the *HeaderText* attribute. If needed, the *header* facet of the column can be used for a richer column header. The *sortable* and *filterable* attributes of the column specify the sorting and filtering behavior.

  Columns can nest other columns to create spreadsheet-like grouped columns.
Adding and Deleting Rows  

Typically, a data control will provide operations to add and delete rows to a collection. These operations can be bound to buttons on the table. A Human Task data control includes Create and Delete operations for every collection in the data control. For example, the add and delete button for the sections table in Figure 8-24 have their action listener attributes set to #{bindings.CreateInsert3.execute} and #{bindings.Delete3.execute}, respectively, where CreateInsert3 and Delete3 are action bindings bound to the Create and Delete operation of the Section iterator as shown in the following XML fragment:

```xml
<action IterBinding="sectionIterator" id="CreateInsert3" RequiresUpdateModel="true" Action="createInsertRow"/>
<action IterBinding="sectionIterator" id="Delete3" RequiresUpdateModel="false" Action="removeCurrentRow"/>
```

The partial-trigger for the table should include the buttons, and the buttons should have the partialSubmit property set to true. If the buttons are added by dragging and dropping the previously mentioned operations from the data control, the drop handler sets up everything correctly including the partial-triggers.

The preceding discussion assumes a single selection table. Deleting multiple rows in a multiselection table can be achieved in backing bean Java code. One possible implementation is to set each selected row as the current row in the iterator and then invoke the Delete operation.

Selection  
The simplest configuration to handle single selection correctly is to set the Selection Listener attribute to an expression of the form #{bindings.section.collectionModel.makeCurrent} and Selected Row Key to an expression of the form #{bindings.section.collectionModel.selectedRow}, where section is the name of the tree binding.

Some scenarios may require some custom processing in the selection listener. If needed, the following code excerpt shows how to handle this in backing Java code:

```java
public void onTreeSelect(SelectionEvent selectionEvent) {
    /* custom pre processing goes here */
    // If needed the added rows can be retrieved using
    RowKeySet rks = selectionEvent.getAddedSet();
}
```
Chapter 8: Developing Rich User Interfaces for BPM with ADF

/* Default Make Current */
RichTree tree1 = (RichTree) selectionEvent.getSource();
JUCtrlHierBinding treeBinding =
(JUCtrlHierBinding)((CollectionModel)tree1.getValue()).getWrappedData();
String exp = "#{bindings." + treeBinding.getName() + ".makeCurrent}";
executeExpression (selectionEvent, exp);
}

private void executeExpression (SelectionEvent selectionEvent, String expr) {
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory exprFactory = fctx.getApplication().
        .getExpressionFactory();
    MethodExpression me = exprFactory.createMethodExpression(elctx,
        expr, Object.class, new Class[] { SelectionEvent.class });
    me.invoke(elctx, new Object[] { selectionEvent });
}

Also, some scenarios may require working with the table’s selected row data outside of the table. For example, the Information Panel in Figure 8-24 displays information about the selected row. The selected row of a table can be worked with in backing bean Java, as shown in the following example code fragment:

private JUCtrlHierBinding treeBinding;
private String[] _attrNames = null;

public String getSelectedRow() {
    Row row = treeBinding.getCurrentRow();
    String ret = "";
    for (int i = 0; i < _attrNames.length; i++) {
        ret += _attrNames[i] + ": " + row.getAttribute(_attrNames[i]) + 
    };
    return ret;
}
Master-detail  The master-detail pattern is commonly used when dealing with hierarchically nested collections. For example, in the page shown in Figure 8-24, the Section table and Paragraph table, as well as the Paragraph table and the Text rich-text-editor have a master-detail relationship. What that means is that selecting a row in the Section table filters the rows in the Paragraph table to only those paragraphs that belong to the selected section, and selecting a row in the Section table changes the text in Text to the text element of the selected section.

A master-detail pair can be easily added by dragging and dropping the detail element from the data control palette and selecting the appropriate master-detail option from the drop handler choices. Otherwise, it is trivial to establish a master-detail relationship between components bound to hierarchically related data source elements. It requires:

- Ensuring that the master table is configured to invoke the collectionModel.makeCurrent on selection as discussed earlier.
- Ensuring that the detail component (table or otherwise) has the master in its partial-trigger, so that it automatically refreshes on selection change in the master.

Tree
While master-detail tables can be used to render hierarchical data, the approach starts getting cumbersome if the depth of hierarchy is more than two levels. ADF Faces also includes a tree component that is ideally suited for displaying hierarchical data in a tree form. The example shown in Figure 8-24 was redone using the ADF Faces Tree component as shown in Figure 8-25.

The .jspx fragment for the tree in Figure 8-25 is shown next, and in the following subsections we will discuss its elements, including the node stamp and path stamp. In the following .jspx fragment, note that, similar to tables, the value for a tree is bound to a tree model (collection model for tables), there is a property var to define a variable that will hold the data specific to each node in the tree (row in table), and that the selection listener is set to #{bindings.elem.treeModel.makeCurrent}.

```
<af:tree value="#{bindings.DocElement.treeModel}" var="node"
   id="t4" partialTriggers="::cb3 ::cb4 ::cb5 ::cb6"
   rowSelection="single"
   selectionListener="#{bindings.DocElement.treeModel.makeCurrent}">
```

```
```
FIGURE 8-25.  A tree example
Node Stamp  The components defined inside the *Node Stamp* facet of the tree are stamped (that is, rendered) for each element in the tree. Only certain types of components are supported, including all components with no behavior and most components that implement the *EditableValueHolder* or *ActionSource* interfaces—that is, input components and command components. The *af:switcher* component can be used to render different node levels with different UI components (in the preceding code fragment, the value is switched inside the expression for *outputText*). The use of the *af:switcher* component is illustrated in the following section “Detail Form” for this example.

Path Stamp  The ADF Tree component has built-in hierarchy management features, including the ability to focus by making a node the root or otherwise simplifying the view on a complex tree. These hierarchy management features are available only if the *Path Stamp* facet of the tree is used. The path stamp facet can only contain read-only components, including images and output text. The components defined inside this facet are stamped (again, rendered) for each node in the path within a pop-up hierarchy browser, which is shown in Figure 8-26 for the example shown in Figure 8-25.

The *Panel Collection* component has built-in toolbar buttons for hierarchy management. For example, the three buttons seen in the top-right corner of the Document Structure panel in Figure 8-25 are built-in panel collection buttons (these buttons are *Go Up*, *Go to Top*, and *Show as Top*). These are enabled only if the *Path Stamp* facet is used and a component in it is bound to the node.

**FIGURE 8-26.** The hierarchy browser for a tree
Determining the Node Type

Since nodes in a tree can have different data definitions, it is important to be able to figure out what is the node type. The `hierarchy.viewDefName` property of the node can be used to determine the type. The values for each level in the tree can be determined from the bindings file, which would have entries such as the following snippet:

```xml
<tree IterBinding="DocElementIterator" id="DocElement">
  <nodeDefinition DefName="TableExUI_EnterDocTask.DocType"
                  Name="DocElement0">
    ...
  </nodeDefinition>

  <nodeDefinition DefName="TableExUI_EnterDocTask.SectionType"
                  Name="DocElement1"
                  TargetIterator="${bindings.sectionIterator}">
    ...
  </nodeDefinition>

  <nodeDefinition DefName="TableExUI_EnterDocTask.ParagraphType"
                  Name="DocElement2"
                  TargetIterator="${bindings.paragraphIterator}">
    ...
  </nodeDefinition>
</tree>
```

The value of the `DefName` attribute of the `nodeDefinition` attribute is the value that needs to be compared with `hierarchy.viewDefName`, for example, as in the following snippet:

```xml
#{node.hierarchy.viewDefName == "TableExUI_EnterDocTask.ParagraphType" ? node.header : node.title}
```

Tree Binding

Similar to table, the easiest way to add a tree is to drag and drop an element from a data control to a page. This launches the same tree binding editor as in the case of the tables; however, in the case of trees the hierarchy needs to be configured in the tree binding editor by using the add button on the `Tree Level Rules`, as shown in Figure 8-27. The attributes at each level selected as display attributes will be the only attributes available through the stamped node variable.

In the case of achieving master-detail with separate tables, each table was tied to its own iterator. However, in the case of tree, one tree iterator for the root element handles the whole hierarchy. The individual levels may also have their own iterators—for example, if a detail form is bound to the individual level. To keep the tree iterator and the individual iterator in sync, use the
Target Data Source specification. For example, in Figure 8-27, for the tree level sectionType, the Target Data Source is set as $[bindings.sectionIterator]. This will make sure that when the current section is changed from the tree, all components bound to section (through sectionIterator) will automatically switch to the new current section.
Detail Forms When using a tree to display the hierarchy, it is typically desired to provide a detail form where the selected element in the tree can be edited. A Switcher component can be used to switch between different forms based on the selection in the tree. Assuming that there is a property selectedNode exposed by a managed bean treeHelper (that refers to the selected node in the tree), the switcher’s facetName attribute needs to be set to #{treeHelper.selectedNode.hierTypeBinding.viewDefName}. Within the switcher, a facet needs to be added for each node type; the facet’s name attribute needs to be set to the viewDefName value for the node type, which can be determined from the bindings file as discussed earlier. Within the facets, the forms for each node type can be added by dragging and dropping the corresponding element from the data control. The .jspx snippet for the Details panel in Figure 8-25 is shown next:

```xml
<af:switcher id="s9" facetName="#{treeHelper.selectedNode.hierTypeBinding.viewDefName}" defaultFacet="none">
    <f:facet name="none">
        <af:outputText value="Select a node in the tree." id="ot17"/>
    </f:facet>
    <f:facet name="TableExUI_EnterDocTask.ParagraphType">
        <af:panelFormLayout id="pfl5">
            ...
        </af:panelFormLayout>
    </f:facet>
    <f:facet name="TableExUI_EnterDocTask.SectionType">
        <af:panelFormLayout id="pfl6">
            ...
        </af:panelFormLayout>
    </f:facet>
    <f:facet name="TableExUI_EnterDocTask.DocType">
        <af:panelFormLayout id="pfl7">
            ...
        </af:panelFormLayout>
    </f:facet>
</af:switcher>
```
To programmatically find the selected node in a tree, the selected row keys need to be determined using the `getSelectedRowKeys` method of the `RichTree` component and then the corresponding nodes must be found using `findNodeByKeyPath` on the tree binding. The code to expose the `selectedNode` property used earlier is shown below:

```java
public JUCtrlHierNodeBinding getSelectedNode() {
    //return _selectedNode;
    JUCtrlHierNodeBinding sel = null;
    if (_treeBinding == null) return sel;

    RowKeySet rks = _richTree.getSelectedRowKeys();
    if (rks == null) return sel;
    //iterate over the contained keys. Though for a single selection
    use case we only expect one entry in here
    Iterator rksIterator = rks.iterator();
    if (rksIterator.hasNext()) {
        //get the tree node key, which is a List of path entries
        describing the location of the node in the tree including its parents
        nodes
        List key = (List)rksIterator.next();
        sel = _treeBinding.findNodeByKeyPath(key);
    }
    return sel;
}
```

As discussed earlier, if the tree binding’s `Target Data Source` is set correctly and if the tree’s selection listener is set to make the selected element current (as in the selection listener setting used earlier), the synchronization of the selection in the tree and the form element happens automatically. Of course the detail section must have its partial trigger set to the tree.

**Add and Delete** Just as in the case of tables, adding or deleting new nodes is as simple as dragging and dropping the appropriate operation from the
By default, the delete command button’s `disabled` attribute is set to `#{bindings.Delete3.enabled}`. The logic of this method provided by the data control is that `enabled` is true if a node of the appropriate node type is available up and down the hierarchy. In the example shown in Figure 8-25, if a user selects the document node, then delete paragraph will be enabled if there are any paragraphs, and trying to delete a paragraph will delete the paragraph that is current in its iterator. Since what is current will not be obvious to the user, the behavior will be confusing. It may be better to enable the delete buttons only if a selected node type is of the appropriate type. For example, the delete paragraph button’s disabled property will be as shown in the following .jspx fragment:

```jsp
<af:commandButton actionListener="#{bindings.Delete3.execute}"
  text="Delete Section"
  disabled='#{treeHelper.selectedNode.hierTypeBinding .viewDefName != "TableExUI_EnterDocTask.SectionType"}'
  id="cb4" partialTriggers="t4"/>
```

By default, adding a node does not select it but makes it current. In the example shown in Figure 8-25, if the user has a section or a paragraph selected, then adds a section, and then adds a paragraph, the new paragraph gets added to the new section, while the old selection continues. This can be confusing to the user. It can be resolved by making the selection change on add. The way to accomplish this is described next.
To automatically select the newly added node, the operation exposed by data control to add a node should be wrapped in Java, and the Java code should force the selection of the newly added node. The newly added node can be determined from the underlying iterator binding as it is its current row. The Java code snippet to accomplish this is shown next:

```java
public void addNewSection(ActionEvent actionEvent) {
    // Add event code here...
    _addSection.execute();
    setNodeSelected (getNewSectionNode ());
}

private void setNodeSelected (JUCtrlHierNodeBinding node) {
    if (node == null) return;
    RowKeySet selected = _richTree.getSelectedRowKeys();
    selected.clear();
    selected.add (node.getKeyPath());
    _richTree.setSelectedRowKeys(selected);
}

private JUCtrlHierNodeBinding getNewSectionNode () {
    JUCtrlHierNodeBinding ret = null;
    if (_docIter == null || _sectionIter == null) {
        System.out.println("an iter is null");
        return ret;
    }
    Key docKey = _docIter.getCurrentRow().getKey();
    Key sectionKey = _sectionIter.getCurrentRow().getKey();
    List<Key> keyPath = new ArrayList<Key> ();
    keyPath.add (docKey);
    keyPath.add (sectionKey);
    ret = _treeBinding.findNodeByKeyPath(keyPath);
    return ret;
}
```

**NOTE**

The preceding code fragment is hard-coded to work only for the section nodes, which are the second level of hierarchy. This can be easily generalized.
In the code, `_addSection` is exposed as a property from the managed Java bean using the code snippet below:

```java
public void setAddSection(OperationBinding _addSection) {
    this._addSection = _addSection;
}
public OperationBinding getAddSection() {
    return _addSection;
}
```

The value of `_addSection` is passed to the managed bean by adding a managed property with the name `addSection`, property-class `oracle.binding.OperationBinding`, and value `#{bindings.CreateInsert3}` (the name of the binding is indicative and may vary).

**Selection**  
As seen earlier, just like in the case of tables, handling selections correctly is a matter of using `#{bindings.elem.treeModel.makeCurrent}` as the selection listener, and if needed the selected node can be found using Java the code shown earlier. In some scenarios, custom selection handling may be needed. The following code snippet shows how selection can be handled correctly in Java code:

```java
public void onTreeSelect(SelectionEvent selectionEvent) {
    /* custom pre processing goes here */

    /* Execute default make current */
    RichTree treel = (RichTree)selectionEvent.getSource();
    JUCtrlHierBinding treeBinding = null;
    treeBinding =
        (JUCtrlHierBinding)((CollectionModel)treel.getValue()).getWrappedData();

    String exp = "#{bindings." + treeBinding.getName() + ".treeModel.makeCurrent}";
    executeExpression (selectionEvent, exp);

    /* Sample snippet to show how to retrieve selected node
    RowKeySet rks2 = selectionEvent.getAddedSet();
    Iterator rksIterator = rks2.iterator();
    if (rksIterator.hasNext()) {
        List key = (List)rksIterator.next();
        _selectedNode = treeBinding.findNodeByKeyPath(key);
    }
}
```
private void executeExpression (SelectionEvent selectionEvent, String expr) {
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory exprFactory = fctx.getApplication().getExpressionFactory();
    MethodExpression me = exprFactory.createMethodExpression(elctx, expr, Object.class, new Class[]{SelectionEvent.class});
    me.invoke(elctx, new Object[] { selectionEvent });
}

Tree Table

ADF Faces also has a tree-table component that displays hierarchical data in the form of a table. The first column of this component is the tree. This component can display additional columns of data for each tree node in the hierarchy. The tree-table component can be understood as a combination of tree and table, and therefore is not covered in detail here; earlier discussions should apply. Some key aspects of the tree-table component are:

- **Node Stamp**  Similar to tree component, tree-table must have a node-stamp facet. However, in the case of tree-table, the node-stamp facet must contain a column. This column is rendered as the first column and as a tree inside the column. The stamping behavior of tree-table is the same as trees.

- **Path Stamp**  Same as tree component.

- **Columns**  Same as table components. However, in the case of tree-table, different levels of hierarchy may have different data definitions, so care should be taken in displaying the column data. Similar to the tree component discussion earlier, the hierType can be used to conditionally display elements or switch components to display.
Data Visualization Components

ADF Data Visualization components provide significant graphical and tabular capabilities for displaying and analyzing data. Like any other ADF Faces components, they can be created and bound to data sources via dragging and dropping from data control elements and their behavior is controlled by properties and facets. ADF Data Visualization components also provide the design time preview of live data. ADF Data visualization components include:

- **Graphs**  
  More than 50 types of graphs are supported, including a variety of area, bar, bubble, combination, funnel, line, Pareto, pie, radar, scatter, spark-chart, and stock graphs.

- **Gauges**  
  Typically, gauges identify problems in data by plotting one data point with an indication of whether that point falls in an acceptable or an unacceptable range. The types of gauges supported include dial, status meter (horizontal and vertical), and LED (bulbs, arrows, and others).

- **Pivot Table**  
  The ADF pivot table component displays a grid of data with rows and columns similar to spreadsheets and provides the option of automatically generating subtotals and totals for grid data. A pivot table allows pivoting—that is, moving data labels and the associated data layer from one row or column edge to another to obtain different views of the underlying data, supporting interactive analysis.

- **Geographic Map**  
  The ADF geographic map component supports representing business data on a geographic map and superimposing multiple layers of information (known as themes) on a single map, leveraging Oracle MapViewer and potentially the Spatial option of the Oracle database.

- **Gantt Chart**  
  The ADF Gantt chart component supports Gantt charts, a type of horizontal bar graph in conjunction with a table, typically used for project tracking. Types of Gantt charts supported include project Gantt chart, resource utilization Gantt chart, and scheduling Gantt chart.
Hierarchy Viewer  The ADF hierarchy viewer component visually displays hierarchical data, typically with master-detail relationships. The most common use of hierarchy viewer may be to display an organization chart, but it can be used to visualize relationships between any other data—for example, the BPM Composer uses the hierarchy viewer to display the catalog of projects available in Composer.

Page Templates
ADF page templates provide a mechanism to ensure that pages are always consistent in structure and layout across an application. Page templates are reusable ADF Faces pages that define the page layout and common components and page fragments. Page templates contain placeholders called facets for authors of pages using the template to insert content specific to the page. When a page is based on a page template, content can only be inserted in the facets defined in the template. Page templates are created in JDeveloper from the New Gallery by selecting the JSF Page Template within the JSF subcategory of Web Tier category. Page templates are edited pretty much the same as ADF pages. In addition to facets, a page template can accept parameters from the page using the template. A page template can also have its own bindings in a file.

Page templates are interpreted, allowing the template to change the layout of a page without changing the page or the application itself.

BPM Form Templates
The BPM wizard-driven form generation leverages page templates. Templates included out-of-box are available inside JDEV_HOME\soa\modules\oracle\soa.worklist_11.1.1\adflibWorklistComponents.jar. Customers can use their own templates too; any custom template must expose the following:

- **Facets**  *Action, header, body, comment, attachment, history;* the names of the facets are clearly indicative of the content that will be generated within them.

- **Attribute**  *Title of type java.lang.String;* the title of the Task will be mapped to this attribute.
Custom templates should be added to an ADF Library JAR, which is created by adding a deployment profile of type *ADF Library Jar File* to the project containing the templates. The jar file containing the templates should be added to the consuming project’s libraries, and then the jar file and the template can be chosen when using the wizard.

**Drag and Drop**

ADF Faces includes a rich drag-and-drop framework. Some drag-and-drop scenarios can be achieved without requiring any code—for example, an attribute of one component can be dragged and dropped on another component simply by adding *Attribute Drag* and *Attribute Drop Target* tags on the source and target component (assuming both attributes are of the same type).

To do something other than copy attributes from one component to another, the *Drop Target* tag needs to be added to the target. Additionally, *Data Flavor* needs to be added to the *Drop Target* to specify the type of object being dropped and the drop handling logic must be implemented in Java as a *Drop Listener*. The object of the drop event is called the *transferable*, which contains the payload of the drop. A drop listener must access the transferable object, and from there use the DataFlavor object to verify that the object can be dropped. It can then use the drop event to get the target component and update the property with the dropped object.

To drag things other than attribute values, the *Drag Source* tag may be used on the source. *Discriminant* property on *Drag Source* and *Drop Target* may be used to pair a drag and a drop.

**NOTE**

ADF supports additional drag-and-drop tags including *Collection Drop Target*, *Component Drag Source*, and *Calendar Drop Target*, which are not discussed here. Interested readers can refer to Chapter 32—“Adding Drag and Drop Functionality” in the “Oracle Fusion Middleware Web User Interface Developer’s Guide for Oracle Application Development Framework.”
The drag-and-drop framework supports three drag-and-drop actions based on the keyboard modifiers used:

- **Copy**  No modifier or CTRL
- **Move**  SHIFT
- **Link**  CTRL+SHIFT

To illustrate drag-and-drop handling, we will modify the example seen in Figure 8-24 to use drag and drop. As shown in Figure 8-28, we will add:

- Drag and drop from the Section table to the Information panel. On drop, we display the drag-and-drop action as well as the attributes and attribute values for the row dragged and dropped.
- Drag and drop from the Information panel to the Section table. A new row in the Section table is added based on the values in the input text box in the Information panel. (This is the scenario shown in Figure 8-28).

**FIGURE 8-28.** A drag-and-drop sample
Drag-and-Drop Sample: From Table to Input Text

As mentioned earlier, in this scenario we drag and drop a row from Section table to the input text within the Information panel.

First, we need to specify that the section table is a drag source. Since we are not dragging a simple attribute, we need to use Drag Source. This can be accomplished by selecting the table in the structure window, right-clicking, selecting Insert Inside, and then within ADF Faces selecting Drag Source. Afterward, select the Drag Source in the structure window and using the Property Inspector window specify Actions as Copy Move, Default Action as COPY, and Discriminant as section. The resultant .jspx snippet is shown next:

```xml
<af:dragSource defaultAction="COPY" actions="COPY MOVE" discriminant="section"/>
```

Next, we need to specify that the input text is a drop target. First, a managed bean to hold the drop handler code should be added, if needed. The Drop Target can be added similar to Drag Source described earlier. In the dialog prompting for Drop Listener, use edit and specify a method name in the managed bean (in this case, dropHandler in dragdropHandler). In the dialog prompting for Data Flavor, org.apache.myfaces.trinidad.model.RowKeySet should be specified. This is the data type that the ADF drag-and-drop framework will use to drop one or more rows. Next, a discriminant should be specified on the Data Flavor to match the discriminant in the drag source. The resultant .jspx snippet is shown next:

```xml
<af:dropTarget dropListener="#{dragdropHandler.dropHandler}"
               actions="COPY MOVE">
  <af:dataFlavor flavorClass="org.apache.myfaces.trinidad.model.RowKeySet" discriminant="section"/>
</af:dropTarget>
```

In the code for the drop handler, first the RowKeySet object is retrieved from the transferable. Next, the collection model for the table is retrieved from the transferable and from the collection model the tree binding; the row corresponding to the key-path in the row-key-set is found using...
findNodeByKeyPath method of the tree binding. In case of the move drag-and-drop action, the row is removed and the table is programmatically added as a partial target so it refreshes to reflect the deletion of row. The code is shown next:

```java
public DnDAction dropHandler(DropEvent dropEvent) {
    DnDAction ret = DnDAction.NONE;
    String discriminant = "section";
    Transferable transferable = dropEvent.getTransferable();
    // The data in the transferable is the row key for the dragged component.
    DataFlavor<RowKeySet> rowKeySetFlavor =
        DataFlavor.getDataFlavor(RowKeySet.class, discriminant);
    RowKeySet rowKeySet = transferable.getData(rowKeySetFlavor);
    if (rowKeySet != null) {
        // Get the model for the dragged component.
        CollectionModel dragModel = transferable.getData
            (CollectionModel.class);
        if (dragModel != null) {
            JUCtrlHierBinding treeBinding =
                (JUCtrlHierBinding)dragModel.getWrappedData();
            // For simplicity assuming single row drop; for multi-row
            iterate over iterator
            List currKey = (List)rowKeySet.iterator().next();
            JUCtrlHierNodeBinding treeNode =
                treeBinding.findNodeByKeyPath(currKey);
            Row row = treeNode.getRow();
            info = dropEvent.getProposedAction().name() + "\n" +
                getRowData(row);
            RichInputText dropTarget =
                (RichInputText)dropEvent.getDropComponent();
            dropTarget.setValue(info);
            if (dropEvent.getProposedAction() == DnDAction.MOVE) {
                row.remove();
                AdfFacesContext.getCurrentInstance().addPartialTarget(dropEvent.getDropComponent());
            }
            ret = dropEvent.getProposedAction();
        }
    }
    return ret;
}
```
Drag-and-Drop Sample: From Input Text to Table

In this scenario, as mentioned earlier, we drag and drop values from the input text to the table and add new rows populated with the dropped data. We expect the dropped data to be a set of attribute name:attribute value separated by newlines. While the earlier scenario showed how to retrieve rows from a drop, this scenario shows how to add rows on a drop.

Since we are simply dragging the value attribute for the input text, we can just use Attribute Drag Source, which can be added similar to Drag Source as described earlier. The resultant .jsp snippet is:

```
<af:attributeDragSource attribute="value"/>
```

Next, we need to add drop target on the tree, which can be added as described earlier. The data flavor should be java.lang.String, and no discriminant is needed. The resultant .jspx snippet is:

```
<af:dropTarget dropListener="#{dragdropHandler.drop2TableHandler}"
    <af:dataFlavor flavorClass="java.lang.String"/>
</af:dropTarget>
```

In the code for the drop handler, the table component is retrieved from the drop event and from the table the tree binding is retrieved via the collection model. The row-set-iterator is retrieved from the tree binding and is used to create a new row. This new row is then populated with the dropped value, which is a plain string. The code is shown next:

```java
public DnDAction drop2TableHandler(DropEvent dropEvent) {
    // *** Create a new row
    RichTable table = (RichTable)dropEvent.getDropComponent();
    // the Collection Model is the object that provides the structured
    // data for the table to render
    CollectionModel tableModel = (CollectionModel)table.getValue();
    // the ADF object that implements the CollectionModel is
    JUCtrlHierBinding. It is wrapped by the CollectionModel API
    JUCtrlHierBinding tableBinding =
        (JUCtrlHierBinding)tableModel.getWrappedData();
    // Access the ADF iterator binding that is used with ADF table
    binding
    DCIteratorBinding tableIteratorBinding = tableBinding
        .getDCIteratorBinding();
    RowSetIterator rowIter = tableIteratorBinding.getRowSetIterator();
    Row row = rowIter.createRow();
    rowIter.insertRow(row);
```
// *** Now get the data being dropped and add to just created row
Transferable transferable = dropEvent.getTransferable();
String data = transferable.getData(String.class);
ArrayList parsedAttrs = parseStringForAttrs(data);
addRowData(row, parsedAttrs);
return dropEvent.getProposedAction();

private void addRowData(Row row, ArrayList data) {
    for (int i = 0; i < data.size(); i++) {
        String[] pair = (String[])data.get(i);
        String attrName = pair[0];
        String attrVal = pair[1];
        try {
            row.setAttribute(attrName, attrVal);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
    }
}

private ArrayList parseStringForAttrs(String str) {
    ArrayList ret = new ArrayList();
    String[] tokens = str.split("\[\s*[^,;][\s]*\]" Component Default screen

ADF Life Cycle

In this section, we will discuss concepts related to the life cycle management of ADF applications, including built-in AJAX behavior, implications of the life cycle on validations, and some other ADF optimizations.
Partial Page Refreshes (AJAX)
AJAX (Asynchronous JavaScript and XML) is a popular web development technique where only portions of a page are re-rendered. This enables highly interactive and responsive web interfaces by minimizing the amount of data being exchanged with the server, as well as providing a better visual appeal by avoiding unnecessary redrawing.

AJAX is built into the ADF Faces framework and is exposed as a feature called partial page rendering (PPR), which enables certain components to re-render without re-rendering the whole page. PPR manifests itself in the following two flavors.

Single Component
Many ADF Faces components include built-in partial refresh functionality. For example, a Table component allows scrolling, sorting, column reordering, and so on, without refreshing components outside of the table. These components are recognized as event boundaries and any events happening within them don’t refresh other components outside of them. Another example of such a component is the pop-up dialog component—any events triggered inside a dialog do not refresh components outside the dialog.

Also, certain events like disclosure and sort events indicate a component as a root. In response to such events, only the identified root component and components contained within it are refreshed.

NOTE
We have simplified the discussion here by using “refreshed.” To be more accurate, the life cycle is run only for those components. (The concept of the life cycle is discussed in the section “ADF and JSF Life Cycle.”)
Cross Component—Partial Triggers

While the preceding flavor is built in, this flavor is interesting to understand because it needs to be specified as part of the page design. It can be specified that a component, referred to as the target component, refresh (including all components contained within it) when any event occurs on another component, referred to as the trigger component.

A trigger component can initiate a cross-component partial page event to be triggered if its autoSubmit (for input components) or partialSubmit (for command components) is set to true.

A component is specified as a target for a trigger component by including the relative ID of the trigger component in its partialTrigger attribute. Since some components are naming containers and others are not, getting the ID right may be tricky and so the partial trigger editor shown in Figure 8-29 should be used.

FIGURE 8-29. The partial trigger editor
PPR behavior can also be programmatically specified from backing Java code using code like the following snippet:

```java
AdfFacesContext adfFacesContext = AdfFacesContext.getCurrentInstance();
adfFacesContext.addPartialTargets(comp);
```

### ADF and JSF Life Cycle

While it is possible to write ADF applications, especially in the context of BPM UI forms, without understanding the details of the underlying technology, having some understanding of the underlying life cycle will help developers. Particularly when designing forms with validations, including required fields, an understanding of the life cycle and its impact on validations will eliminate unnecessary frustration.

The stages in the JSF life cycle are:

- **Restore View**  
The component tree is established; this phase is not interesting from the perspective of the web UI developer.

- **Apply Request Value**  
A component’s values are retrieved from the request parameters and stored locally (the model is not updated yet). If the immediate attribute is set for the component, then conversions, validations, and events associated with the component are processed in this phase.

- **Process Validations**  
First, the local values are converted from the input type to the underlying data type. If conversion succeeds, required test is performed on the component—that is, if the required attribute is set on it, then testing that it has a value. If the required check succeeds and the value is not empty, all associated validation rules for the component are run (even if some fail).

If any of those just listed—conversion, required-check, validations—fail, the component is marked as invalid and an error message is added. However, this phase runs to completion—that is, all components are tested. At completion, if any component is found to be invalid, the life cycle jumps to the **Render Response** phase.
At the end of this phase, converted versions of the local values are set (no updates are made to the model yet), any validation or conversion error messages and events are queued on the FacesContext object, and any value change events are delivered.

- **Update Model Values**  
  It is only in this phase that the model is updated with the new (local) values.

- **Invoke Application**  
  The application is invoked—that is, all actions and listeners such as action listeners and disclosure listeners are executed. If an action has a navigation associated with it and if the `immediate` property for it is not set to `true`, the navigation is performed in this phase.

- **Render Response**  
  The page (or view) is rendered.

Some aspects that web developers must understand about the life cycle are:

- Model values are updated only during the update model, whereas validations are run before. This means that if validation rules for a component depend on values of other components, by default the updated values are not available during validation.

- If any component fails validation, model values are not updated for all.

- The default behavior can be overridden using the `immediate` attribute, which is discussed in more detail later.

**The ADF Optimized Life Cycle**

ADF improves upon the JSF life cycle in the following ways:

- **Partial Page Refreshes**  
  As discussed earlier.

- **Client-side Life Cycle**  
  Built-in and custom Javascript-based conversion and validation that runs on the client without requiring a roundtrip to a server.
Subform Component
A subform component submits a group of form entries without impacting others, enabling parts of a page to be validated and submitted independently, while maintaining a single page state.

Immediate
As noted earlier, the immediate attribute can be used to change the JSF Life Cycle. There is a difference in behavior of this attribute between command components (such as buttons) and input components (such as input text).

The Immediate attribute on command components cause application events, such as action listener, to be delivered in the Apply Request Value phase instead of the Invoke Application phase; essentially all other phases are skipped. Therefore, this is useful in scenarios, such as cancel, where the validation or updating of other values is not needed. This can also lead to better performance by avoiding unnecessary validations, such as when a command component would navigate to another page. A command action that is only meant to navigate to another view also benefits from immediate=true because validation is omitted (as is the full life cycle). Only when navigation should occur after a model update, then immediate=false makes sense for such command components.

An immediate attribute on input components causes only the component event handling and validation to move up to the Apply Request Value phase instead of the Process Validation phase, so that they execute before components that don’t have immediate set to true. Unless renderResponse is called explicitly (from an event handler), the life cycle continues normally. Immediate on input components is needed in scenarios where some components need to process ahead of validation running on others, including where the validation of the other components depend on this component. Such a scenario is discussed in the next section.

If an immediate action needs values from an input component, the input component should also be made immediate; however, since actions are processed before the model is updated, either the value should be retrieved from the component or the component’s event handler should update the model (as in the example in the next section). While immediate input components process before immediate actions, validation failures do not prevent the action from firing.
NOTE
Setting immediate=true on a component does not mean that it is processed alone; if no errors are detected in its processing all other components follow their normal life cycle. To avoid processing of life cycle for other components, for example when dependent list boxes need to be set without validating other form fields, the FacesContext.getCurrentInstance().renderResponse() method needs to be called from the value change listener.

Validation Example
To illustrate the preceding concepts, consider an example where there is a comment field that is required only if the required checkbox field is checked, as shown in Figure 8-30.

To start with, autoSubmit needs to be set on the required checkbox, and the required checkbox’s ID must be added to the partialTrigger attribute of the comment field to ensure that as the checkbox is changed, the comment field is updated. Also, the required attribute on the comment field is set to #{bindings.required.inputValue} (required is the binding for the checkbox).

At this point, the behavior will be that, once the required checkbox is checked, it cannot be unchecked without specifying some value in the
comment field. This happens because the validation for the comment field happens in the Process Validation Phase ahead of the value for required getting updated, and once validation fails, model updates are skipped.

To get this scenario working correctly, two things are needed:

- Since the model update for required cannot wait for the Update Model Values phase, a value change event listener is needed and within that listener the model value needs to be updated explicitly, as shown in the following code snippet:

```java
public void reqChanged(ValueChangeEvent valueChangeEvent) {
    BindingContainer bc = BindingContext.getCurrent().getCurrentBindingsEntry();
    AttributeBinding cb = (AttributeBinding) bc.getControlBinding("required");
    cb.setInputValue(valueChangeEvent.getNewValue());
}
```

- The event listener needs to be run ahead of the Process Validation Phase. This is achieved by setting immediate on the required checkbox to true.

**ADF Task Flows**

ADF Task Flows define flows of pages and methods. Typically, task flows are interesting to application developers because any application involves navigation between pages. In the context of BPM forms, while many scenarios may have traditionally been addressed by a single page form, the ability to break down a page into multiple pages and provide some guided navigation has the potential of simplifying the user experience, especially when dealing with complex forms.

ADF has two flavors of task flows: unbounded task flows and bounded task flows. An unbounded task flow is useful for building application navigation and does not apply to BPM forms; therefore, we will not cover unbounded task flows in this section.
Bounded Task Flows

A bounded task flow has a well-defined entry point and can be invoked with parameters. A BPM ADF UI is always wrapped in a bounded task flow, even when the UI is a single page; therefore, BPM UI elements are typically referred to as BPM Task Flows (although in this chapter we have been loosely referring to them as BPM forms). In addition to being used for BPM UI, some other use cases for bounded task flows are:

- Creating regions that are contained within other pages
- Encapsulating reusable page flow logic that is invoked as a subtask flow
- Creating regions to be included in Web Center

An example bounded task flow is shown in Figure 8-31. The activity with a shaded circle around it is the entry point for the task flow. The rest of the elements in the task flow are discussed in the following sections.

Transitions or Control Flow

Control Flows (similar to transitions in BPMN) define navigation between activities in a task flow. The attributes From Action and From Outcome, if specified, limit the applicability of the control flow to only when the navigation is caused by the specified action or outcome. In bounded task flows, the From Outcome is the more commonly used option.

Wildcard Control Flow

In Figure 8-31, a big star, which is the source of transition labeled closeTaskFlow can be seen. This is a wildcard control flow, which means that regardless of the source, if the outcome from the navigation is closeTaskFlow, follow the transition (assuming that there is not an overriding local definition on the source activity). In addition to a pure wildcard, a trailing wildcard such as pre* can be specified, in which case the control flow is limited to navigations whose source has an activity-id starting with pre. This is useful for grouping together pages with similar control flow requirements and simplifying the specification of control flows.
View Activity

The view activity, the most commonly used activity in a task flow, represents a page or a page fragment within a task flow. A new page for a view activity can be created by double-clicking the activity, or it can be associated with an existing page by specifying the page attribute accordingly. Also, an existing page can be dragged and dropped into the task flow to add a new view activity representing the page.

FIGURE 8-31.  A sample bounded task flow
Router Activity
The conditional navigation provided by control flow is limited to actions and outcomes and doesn’t support BPMN like data-based expressions. The router activity addresses this limitation by mapping data-based conditional expressions to outcomes that can then be used in control flows. In the example shown in Figure 8-31, a router activity is used to determine which page to navigate to, depending on the return value of the previous activity. An example of router activity was also discussed earlier in the section “Forms for E-mail.”

Method Activity
A method activity allows invocation of a method or a service between page navigations. A method activity may be useful to fetch the data needed to populate a page, do processing between page navigations, and to evaluate rules on what page to show next. A method activity can be bound to a method or operation exposed by a data control as well as to backing bean methods.

Task Flow Entry Point—Default Activity
As noted earlier, every bounded task flow must have a well-defined entry point. The default activity in a task flow is the entry point. An activity can be marked as default by right-clicking it and then selecting Default Activity within Mark Activity.

Task Flow Return Activity
A bounded task flow should end with a task flow return activity to ensure that resources are appropriately released. Also, the outcome property of the task flow is returned to the calling task flow.

NOTE
To ensure that the task flow return activity is always invoked on task close, BPM-generated task flows include a wildcard transition to a task flow return activity.
Task Flow Trains

As mentioned earlier, a big value-add of providing multipage task flows as BPM UI is to navigate the user through bite-sized pieces of information. A common metaphor to facilitate such navigation is to provide the user with a navigational tool that shows them where they are in the context of the overall task, as well as to navigate easily between pages. ADF Trains provide a built-in framework for adding such navigational context to bounded task flows. Figure 8-32 shows a BPM ADF Task Flow using a train to navigate a sales representative through the various steps in creating a quote: enter header, select products, request discounts, request terms, and review (and submit).

A train can be added to a bounded task flow by right-clicking the task flow canvas and then selecting the Create Train submenu item within the Train menu item. The ordering of activities within a train can be changed by right-clicking the activity, selecting the Train menu item and then selecting the appropriate ordering option (such as move forward).

FIGURE 8-32. A sample task flow train
While the default navigation for task flow trains is that each train stop is visited in sequence, it can be modified using the following attributes on the activity:

- **Sequential** If the sequential attribute evaluates to false, the activity can be performed in any order; otherwise, it can be visited only after the preceding stop.

- **Skip** If the skip attribute evaluates to true, the train-stop is skipped.

The label and tooltip to display for a train stop can be specified by selecting the `train-stop` child of the activity in the `structure` window and adding `Display Name` and `Description` to it. For the navigation bar to be included in a page, the ADF Faces `Train` or `Train Button Bar` component should be added on the page.

**ADF Business Components**

In many BPM scenarios, data stored in a database needs to be accessed. Some use cases include:

- BPM is working with entity data and, since the process does not own the data and others may want to access and modify the data while the process is in flight, the data is mastered outside of BPM in a database and BPM accesses it.

- Supporting data is needed for users to perform their tasks intelligently—for example, a list of products for a user creating a quote, or credit card transactions for a user approving a credit change request.

- A list of values used for fields in a task form is stored in a database to facilitate easy changes to the list.

Oracle ADF includes a business services framework—ADF Business Components (BC)—that not only enables the development of database-backed applications but is also very useful in the context of BPM for the
scenarios mentioned earlier. The ADF-BC framework provides Object-Relational (O/R) mapping, the definition of application-specific views possibly with custom validation rules and other business logic, default operations including CRUD (create, read, update, delete), the caching of data, management of transactions, and the coordination of master-detail behavior.

ADF-BC components can be added to the model project in a Fusion Web Application (essentially a project that has ADF Business Components in its technology scope) by using the New Gallery and selecting the Business Components from Table option from the ADF Business Components subcategory of the Business Tier category. ADF-BC is a declarative framework and metadata for all components is stored in XML files. While most use cases, especially in the context of BPM, can be addressed with just the declarative framework, ADF-BC provides hooks to add Java classes to extend or modify the behavior of the framework.

If the database schema is set up correctly with the right foreign key and other constraints, creating ADF-BC components is a matter of walking through the wizard. The wizard creates the ADF-BC components with the right relationships based on the foreign keys, as shown in a sample ADF-BC diagram in Figure 8-33 (this diagram is generated by JDeveloper). Most users will need to make very few changes. Although more advanced usage such as employing Groovy for business logic or using Java extension classes is possible, it is out of the scope of this book. Since the simple usage of ADF-BC is trivial and advanced usage is out of our scope, this section only gives a high-level overview of ADF-BC.

**Entity Objects**

The entity object provides O/R mapping, exposing columns in the underlying database table as entity object attributes. The entity object holds the data retrieved from the database and is responsible for manipulating it. Business and validation logic is implemented in the entity object. An entity object can be created against a database table, view, or synonym, as well as against a Web Service data source.
Associations
As can be seen in Figure 8-33, entity objects can have relationships among themselves. These relationships are defined by associations. As mentioned earlier, default associations are generated by the wizard based on foreign key relationships. The default associations should suffice in most scenarios.

Attributes
Attributes represent columns in the underlying table. They define the column data type, primary key, unique key settings, and more. Optionally, entity attributes can be created as transient, in which case they don’t have a related database column and are based on calculations. Also, display hints can be specified. While the hints can be overridden by the view objects or within the page, providing hints at the entity level leads to consistency and less work when the entity is used by multiple views.

FIGURE 8-33. A sample ADF-BC diagram
Validations

One of the benefits of using ADF-BC is the ability to define business logic in a declarative way. Validations can be specified on the entity object attributes as well as the entity object itself. Validations on the entity object attributes are triggered when the attribute value is changed, while those on the entity object are triggered when rows are changed or during commit.

The following types of validations are supported:

- **Compare**  Compare the attribute against a literal value, a Groovy expression, or another attribute value. *View Accessors* can be used to get a value from a view definition.

- **List**  Test if the attribute value is in a list. The list can be a static list of values, values from a view object, or specified by a SQL query.

- **Key Exists**  Test if the value already exists as a key.

- **Length**  Test the value’s length either in characters or bytes for a specified condition.

- **Range**  Test if the value is within or outside the specified range.

- **Regular Expression**  Test if the value matches a regular expression pattern. Regular expression patterns for U.S. phone numbers and some other common scenarios are built in.

- **Script and Method**  Execute a Groovy script or a Java method returning *true* or *false* to indicate success or failure.

In addition to the preceding validations that are available for attributes, the following validations are available at the entity object level:

- **Collection**  Test against aggregated value—sum, average, min, max—of an attribute across all records.

- **Unique Key**  Test if the primary key or an alternate key is unique across all records.
Validations can be conditional—that is, only when the specified condition is true the validation is executed. Also, validations allow for the specification of failure messages, which can be internationalized.

Java Extension
JDeveloper provides an option to generate an implementation class for the entity object. The getters and setters for the attributes can be modified to add desired behavior. Also, this class allows for extending the logic when creating and removing rows, as well as executing data operations.

Events
From within the Business Events tab of the entity object editor, business events and their publication can be defined; the editors to create business event definition and to set up a business event publication are shown in Figure 8-34. The event definition consists of an event name and a list of attributes; attributes may be either always included in the event or only if the attribute value changes. For example, in Figure 8-34 we see an event named CardEvent that includes two attributes CardNumber and AvgMonthlyBalance; AvgMonthlyBalance is included in the event only if its value is changed. Events can be set up to be published on the entity object create, update,
or delete; also, publication can be filtered on conditions that test for an
attribute value change or an attribute value. For example, in Figure 8-34 we
see that the CardEvent is published whenever the underlying entity object is
updated and either attribute AvgMonthlyBalance is changed or the attribute
Brand equals Master Card.

These business events are published to the Event Delivery Network
(EDN) and can be consumed by BPM processes via signals.

View Objects
The view object is the heart of ADF-BC for UI developers. It defines what
the end-user sees in the application. The view object defines the SQL
statement that selects and orders the required data into the underlying entity
object, and at runtime the view object is a collection of entity objects that
represents the result set of the specified SQL query. Although typically a
view object will be based on one entity object, it can be based on multiple
entity objects (via joins), as well as none (read only view objects).

Query
The view object is essentially a select statement. Group By directives can be
specified or the SQL can be tweaked as needed.

View Criteria
The query can be further filtered by defining view-criteria. These filters can
use bind variables, in which case an Exec with Params operation is exposed
(something we have already encountered in this chapter.) View-criteria can
also be associated with a view instance in an application module to expose
a filtered version of the view. Another use of view-criteria is that it is available
to use in query panels to provide search behavior.

Attributes
Attributes in the view object map to attributes in the underlying entity objects.
Similar to entity objects, transient attributes can be defined. Transient
attributes are attributes that don’t match an entity attribute and are not
persisted on a transaction commit. Also, display hints for attributes can be
specified. Display hints specified on view objects override those specified
on the underlying entity objects.
View Links
As can be seen in Figure 8-33, views can be related to one another. Such relationships are defined by view links. Default view links are generated based on the associations in the underlying entity objects. Additional view links can be added to create custom relationships—for example, an active address link from a customer to filter only those addresses marked as active.

List of Values
Earlier, within the discussion on selection components, we saw how selection components can be bound to a static and dynamic list of values. ADF-BC allows specifying the list of values as part of the view object definition itself. A list of values can be added for a view object attribute by clicking the plus icon in the List of Values section in the Attributes tab. This launches the Create List of Values window where another view object can be selected as the list data source, as shown in Figure 8-35. When the list of...
values is specified on the view object attribute, when using select components, the Model Driven List of Values option must be selected.

In the section on “Selection Components,” we also discussed the dependent lists scenario. For ADF-BC objects, dependent lists can be configured as part of the model definition—view-criteria needs to be associated with the view accessor added as part of the LOV definition. This automatically achieves the desired behavior.

Custom Methods and Other Java Extensions
The view object implementation class for a view object can be generated. This is useful for adding custom methods to the view object beyond the standard operations, as well as to introduce custom behavior such as programmatically managing query and view-criteria. Custom methods exposed on a client interface are available as part of the data control and can be dragged and dropped as a command component onto a view.

A custom method can be added to the view object by implementing the method in the implementation class and then editing the Client Interface within the Java tab to add the method to the interface.

Application Modules
An application module packages a collection of view objects and defines the data model available for client users. A data control is automatically available for every application module and exposes the view objects included in the application module as immediate children of the data control node. If a view object is not added to an application module, it will not be accessible from the data control. In addition to view objects, methods can be added to an application module.

The configuration tab of the application module specifies how the connection to the database is made. By default, a JDBC URL of the form jdbc:oracle:thin:bpmserver:1521:XE is used. Before deployment, the configuration should be changed to use a JDBC data-source of the form jdbc/dataSource, so that the deployed application can leverage JDBC data-sources defined in the application server.
Service Interface

An ADF-BC application module can be exposed as a service by adding a service interface from the Service Interface tab. Operations, methods, and view-criteria of view instances in the application module, as well as methods on the application module, can be selected for inclusion in the service interface, as can be seen in Figure 8-36. Also shown in Figure 8-36 are two Java files in addition to the WSDL and XSD files. The first defines the interface of the service, and the second Java file is the remote server class, which is an EJB 3.0 stateless session bean that implements the interface.

A BPMN process can consume the service exposed using the ADF-BC Service adapter component.

FIGURE 8-36. A Service interface exposed by an application module
BPM and ADF-BC

In addition to being used in BPM UI either to get supplementary data or for a list-of-values, ADF-BC and BPM work together in the following ways:

- **Document-based Routing**  The human workflow component of BPM can provide rich document-based routing when working in conjunction with ADF-BC data. This was discussed in detail in chapter 7.

- **ADF-BC Service**  Services exposed by ADF-BC application modules can be consumed by BPM using the *ADF-BC Service* adapter.

- **Events**  Events published by ADF-BC entities can be subscribed through BPM processes.

**ADF-BC vs. Database Adapters**

BPM also includes a database adapter that can be used to retrieve data from a database into a process. In the context of getting data for displaying as part of a task form or for using as a list-of-values, using ADF-BC is recommended because it provides the following benefits:

- **Data Currency**  If a database adapter is used, then the process will need to retrieve the data. This can result in a time lag between when a user interacts with the data and when the process retrieves it. However, when using ADF-BC, the data can be fetched by the UI itself, ensuring it is current.

- **Business Logic**  ADF-BC is much more than object-to-relational mapping. As discussed earlier, it supports business logic like validations, as well as provides hooks for the inclusion of custom business logic.

In the context of a process accessing data from a database for processing (and not for a user interface), working with a database adapter is simpler. However, using ADF-BC means that business logic is shared between BPM and user interface layers. Therefore, if ADF-BC already exists, or there is a possibility of building user interfaces on top of the data, or there are business logic and validation requirements, ADF-BC should be used. Otherwise, a database adapter should be used.
For More Information

ADF documentation is available under the category “Development Tools” within the Fusion Middleware documentation library.

Anyone working with ADF Faces is recommended to leverage the “ADF Faces Cheat Sheets” within ADF documentation. A runtime demo showcasing these components is also hosted at http://jdevadf.oracle.com/adf-richclient-demo, which is a great way to learn by seeing different components in action. The source code for this demo can also be downloaded from www.oracle.com/technetwork/developer-tools/adf/downloads/index.html by selecting the Oracle ADF Faces Components Demo. The ADF page at Oracle Technology Network (OTN) at www.oracle.com/technetwork/developer-tools/adf/overview/index.html has a wealth of information including articles and step-by-step tutorials.

Two other books available in the documentation library that are particularly useful are “Web User Interface Developer’s Guide” and the “Fusion Developers Guide.” The former is focused on ADF Faces and topics such as working with trees, tables, pop-ups, drag and drop, and so on, while the latter is focused on the full application stack, including ADF-BC, Task Flows, and others.

In addition to the Oracle documentation library, the “Oracle Fusion Developer Guide” by Frank Nimphius and Lynn Munsinger (Oracle Press, McGraw-Hill Professional, 2009) is an excellent resource for readers seeking to go deep into ADF. The Oracle ADF Code Corner (www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html) is a great collection of how-tos and code snippets.