Oracle Fusion Middleware 12*c*: Build Rich Client Applications with ADF

Student Guide – Volume II

D76564GC10

Edition 1.0

July 2014

D87571

ORACLE

Authors

DeDe Morton
Lynn Munsinger
Gary Williams

Technical Contributors and Reviewers

Matthew Cooper
Martin Deh
Steven Davelaar
Frédéric Desbiens
Susan Duncan
Brian Fry
Jeff Gallus
Joe Greenwald
Taj-ul Islam
Peter Laseau
Duncan Mills
Chris Muir
Frank Nimphius
Gary Williams
Lynn Munsinger
Katarina Obradovic
Grant Ronald
Shay Shmeltzer
Richard Wright

Editors

Daniel Milne
Richard Wallis
Aju Kumar
Vijayalakshmi Narasimhan

Graphic Designers

Divya Thallap
Maheshwari Krishnamurthy

Publishers

Pavithran Adka
Jayanthy Keshavamurthy
Joseph Fernandez

Contents

11

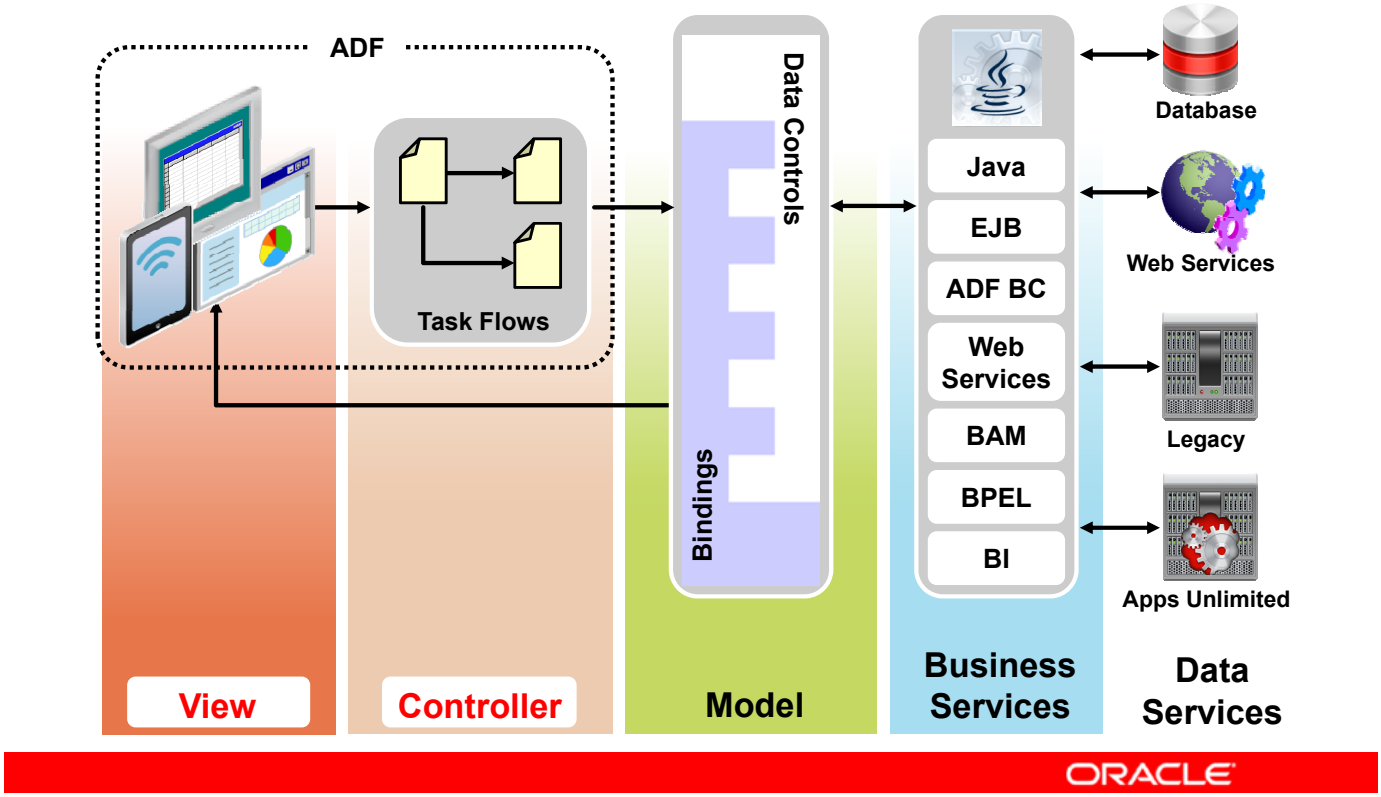
Passing Values Between UI Elements

Objectives

After completing this lesson, you should be able to:

- Define the data model to reduce the need to pass values
- Use a managed bean to hold values
- Store values in memory-scoped attributes
- Use parameters to pass values

Passing Values Between UI Elements



In this lesson, you learn how to better encapsulate UI functionality by passing values between UI elements such as pages, regions, task flows, and page fragments. This lesson lays the foundation for a later lesson where you learn about the reusability features that are available in Oracle ADF.

Holding Values in the Data Model (Business Components)

- Define transient attributes to save additional state information.
- When possible, rely on row currency to coordinate multiple pages without passing parameters.
- Define view links in the data model. This will:
  - Coordinate master and detail pages
  - Re-execute queries as needed
  - Reduce the need to pass parameters between UI pages and regions

ORACLE

You can reduce the need to pass parameters between pages and regions by properly constructing a data model in ADF Business Components.

Examples:

- You can define transient attributes in business components to save additional state information (such as a soft delete flag or information that is not part of the entity).
- ADF BC maintains the state of the currently selected row, so a query form that displays results on a different page does not have to pass a parameter to that page but instead relies on row currency.
- When you define view links between related view objects, you can rely on row currency to coordinate pages that contain master and detail results, without explicitly passing parameters or writing code to re-execute queries.

Development of the user interface is, therefore, much easier if you take time to define a data model that accurately reflects the relationships in your data.

Holding Values in Managed Beans

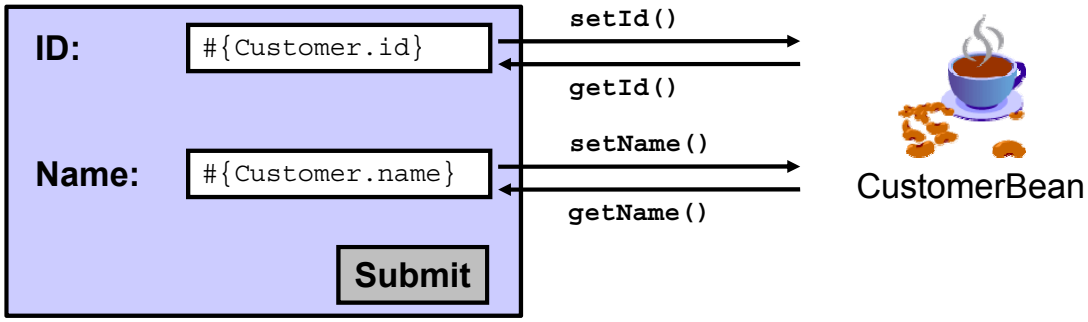
- Managed beans are optional and can be used to:
  - Hold values
  - Store state
- Using managed beans:
  - Bean properties
  - Memory-scoped attributes  **Not recommended**

You can pass values between pages by using managed beans to hold the values. You can store values in the properties of managed beans that you create and register, or you can store values in memory-scoped attributes without explicitly defining a managed bean. However, to protect your application from null pointer exceptions, it is recommended that you avoid using memory-scoped attributes. Instead, use managed beans. Unlike memory-scoped attributes, properties in managed beans are discoverable using EL Builder, can be documented using Javadoc, and can be initialized with default values.

The JSF run time manages instantiating the managed beans on demand when any EL expression references them for the first time. When displaying a value, the JSF run time evaluates the EL expression and pulls the value from the managed bean to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF run time pushes the value back to the corresponding managed bean by using the same EL expression.

Storing Values in Bean Properties

- Store values as properties in managed beans.
- Expose the bean properties through getter and setter methods.
- Use EL to get and set property values.
- Use managed properties to set values for bean initialization.



As you learned earlier, you can store values as properties in managed beans and expose the properties through getter and setter methods. You then use Expression Language to get and set the property values, as shown in the image in the slide.

Managed beans also have managed bean properties that you can configure on the Managed Bean tab of the `.xml` configuration file for a task flow. You use managed properties to set values for bean initialization.

One reason to use a managed property in the `adfc-config.xml` file is to have an expression that is always evaluated and ready for use. For example, you can use a managed property to evaluate `#{bindings}`, thus giving you handy access to the binding container.

Standard JSF Memory Scopes

Scope	Bean Lifespan	Use Cases
Application	Available to all instances of an application	Can be used to define static lists of data used by all instances
Session	Available until the user session is invalidated or expires	Use only for beans that carry information for use anywhere in the application (for example, a bean that holds user information).
Request	Available for the duration of a request; stores its internal state until the request completes	Use to pass data between pages.
none	Released immediately; does not store state	Use for helper classes that do not save state.

ORACLE

Managed beans have a *scope*, which is the lifetime between the bean's instantiation by the framework and its dismissal for garbage collection. (You specify the scope when you register the managed bean.) In general, you should always use the smallest possible managed bean scope, which keeps beans in memory only as long as they are required. The available scopes include JSF scopes (covered here) and additional scopes that are added for ADF (covered in the next slide).

Standard JSF memory scopes include:

- **Application:** Application-scoped managed beans are available for an entire application and are shared among users. Application scope information should not have session dependencies because it is visible to all sessions of the application.
- **Session:** Session-scoped managed beans are user interface–specific and live until the user session is invalidated or expires. For example, you can use a session scope bean to store user information that was read from the database or an LDAP server. Storing the user information in a bean avoids unnecessary queries.
- **Request:** Request scope is the smallest available standard JSF scope. It lasts for the time it takes to transition from one page to the next.
- **None:** The managed bean is instantiated each time it is referenced. Configuring a managed bean with a scope of “none” makes the bean accessible through EL.

ADF Memory Scopes

Scope	Bean Lifespan	Use cases
View	Available until the ID for the current view changes	Use to hold values for a specific page.
Page Flow	Available for the duration of the unbounded or bounded task flow	Use to pass values from one page to another in a task flow.
Backing Bean	Available from the time an HTTP request is sent until a response is sent back to the client	Use for managed beans that back page fragments or declarative components.

In addition to the standard JSF memory scopes, ADF Faces provides the following scopes to address the limitations of traditional web application scopes:

- **View:** Stores objects used by a single page and retains the objects as long as the user continues to interact with the page, and then automatically releases the objects when the user leaves the page. The lifetime of a view scope begins and ends with the change of the view ID of a view port (which can be either the root browser window or an ADF region that displays the current view).
- **Page flow:** Exists for the duration of a task flow, after which the allocated memory is released. This scope makes it easier to pass values from one page to another in a task flow. If a task flow calls another bounded task flow, the page flow scope of the caller flow is suspended and activated again upon return from the called task flow
- **Backing bean:** Is used for managed beans for page fragments and declarative components. The bean is available from the time of an HTTP request until a response is sent back to the client. Backing bean scope is a special case of request scope narrowed down to a specific managed bean instance. You use this scope to avoid potential collisions when a page includes a fragment or declarative component. Because there might be more than one page fragment or declarative component on a page, values must be kept in separate scope instances.

Note: Declarative components are an advanced topic that is not covered in this course.

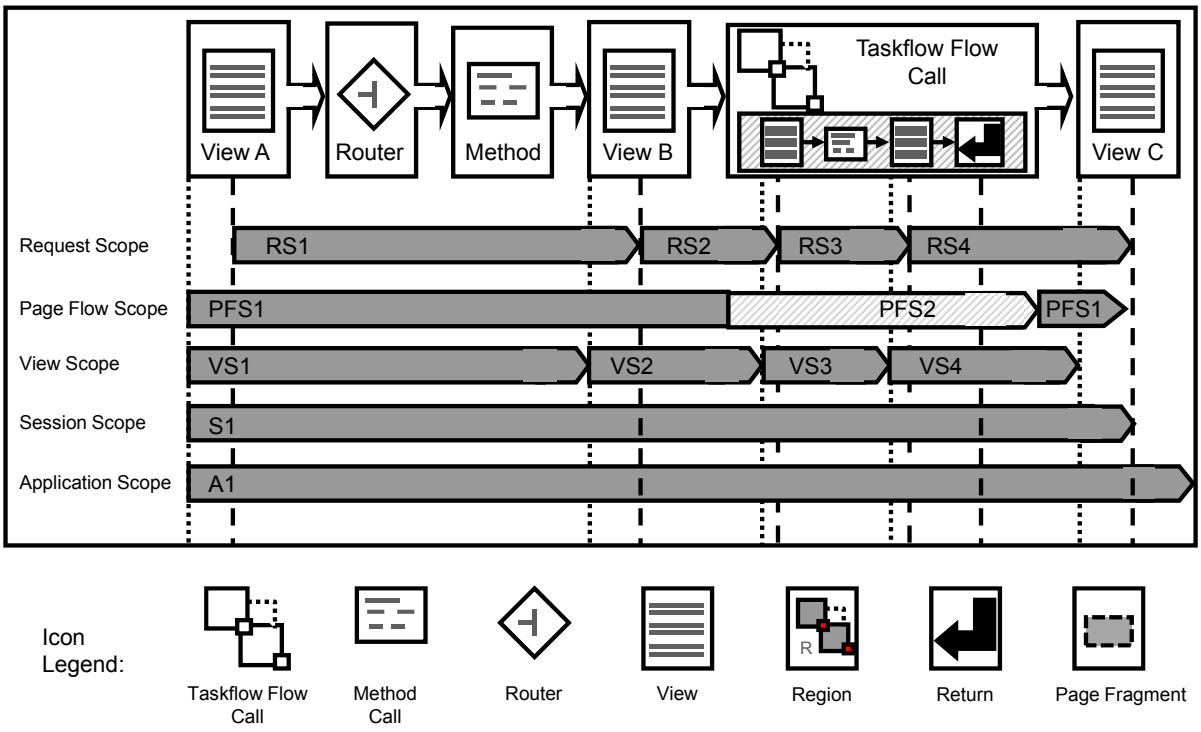
Using Page Flow Scope

For efficient memory usage, you should store values in the scope with the shortest duration that fits your needs. For example, to make a value accessible anywhere in a task flow, you should use page flow scope. Each ADF task flow can specify a page flow scope that is independent of the memory scope for all other task flows.

When one task flow calls another, the calling task flow cannot access the called task flow's page flow scope, so you can use the page flow scope to pass data values only between activities in a task flow. Application and session scopes are also allowed in task flow definition files, but in most cases they are not recommended because they may keep objects in memory longer than needed.

If the user opens a new window and starts navigating, that series of windows has its own page flow scope; values stored in each window remain independent. Clicking the browser's Back button resets page flow scope to its original state.

Memory Scope Duration with a Called Task Flow



As you learned previously, you should store values in the scope with the shortest duration that fits your needs, so it is important to know the duration of memory scopes in an application.

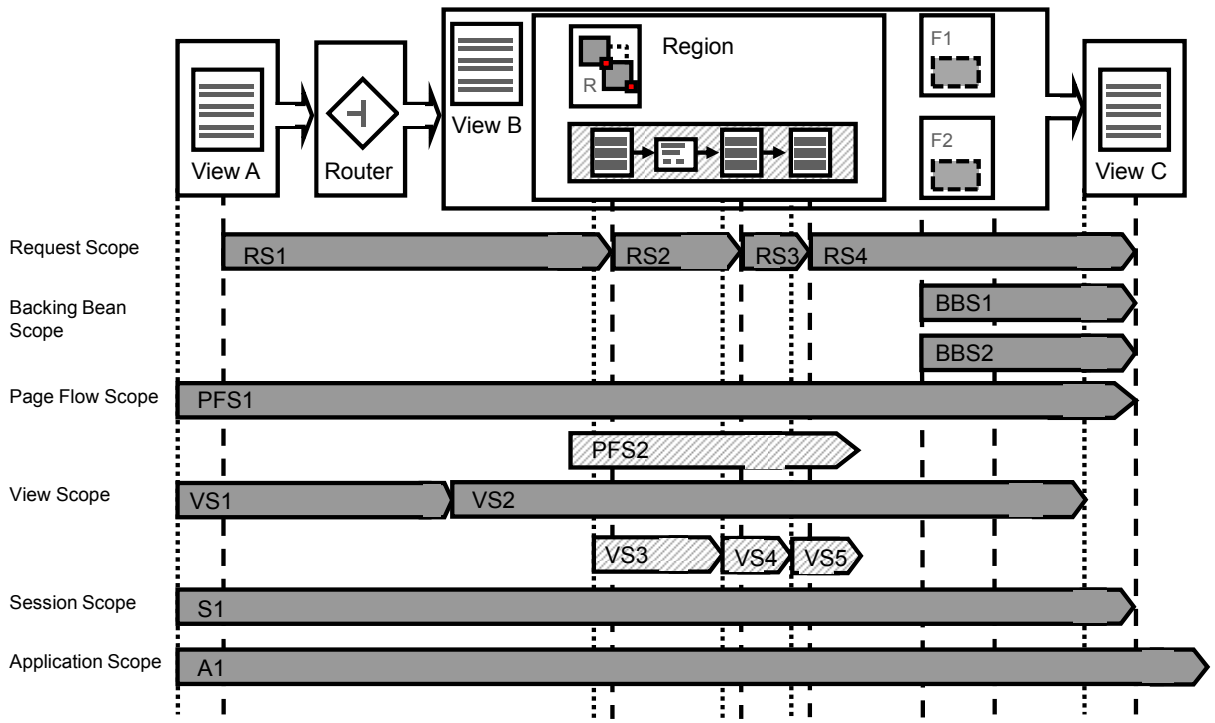
The parent task flow depicted in the slide contains three view activities: a router, a method call, and a task flow call. The called bounded task flow has two view activities: a method call and a task flow return.

The request scope line shows that the duration of a request scope is from the time a request is issued in a view activity until another view activity creates a request. Routers and method calls do not have a request scope of their own, so values that are set in RS1 are available to the router and the method call. If the called task flow had no view activities, RS1 would span the task flow also. However, each of the called task flow's two view activities creates a request, ending the previous request scope and starting a new one. View scope is similar, except that it begins when a view is rendered and ends when a new view is rendered.

The page flow scope attributes are accessible anywhere in a task flow. The called task flow has its own page flow scope, PFS2, that is separate from the parent's PFS1, and the called task flow cannot access PFS1. As a result, parameters (discussed later in this lesson) are needed to pass information from the parent to the called task flow.

Session scope lasts throughout the user session, while application scope lasts beyond the user session and can share values among all instances of an application.

Memory Scope Duration with a Region



This example is similar to the previous one, except that the bounded task flow is contained in a region on the page that is represented as a view activity in the main task flow. It shows the following differences from the previous slide:

- The backing bean scope is available for the page fragments and lasts as long as the page fragment is in focus. Each fragment notifies the controller when it starts and ends being in focus. There is a separate backing bean scope for each fragment.
- The page flow scope works differently in this example because the second task flow is in a region on a view in the main task flow. So the duration of the page flow scope of the main task flow is throughout its entire flow, including the region that contains the other task flow. The bounded task flow that is represented as a region has its own page flow scope as well. Each task flow's scope is independent of (and cannot access objects in) the other task flow's scope.
- The view scope for the page fragments in the bounded task flow (in the region on the page represented by the view activity B) is separate from the view scope for the containing page. The views in the region cannot access the view scope of the containing page, and the containing page cannot access the view scopes of the page fragments in the region.

Accessing ADF Memory Scopes

You can use EL to access the following ADF memory scopes:

- `#{pageFlowScope}`
- `#{viewScope}`
- `#{backingBeanScope}`

Example:

```
<af:inputText label="Label 1" id="it1"
value="#{pageFlowScope.customerBean.discountCode}"/>
```

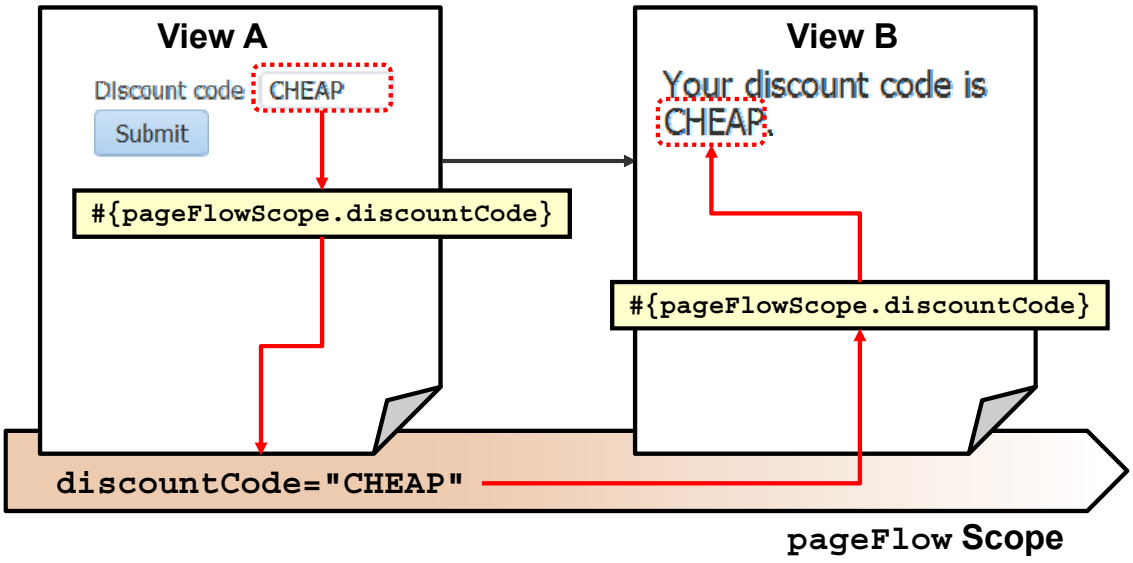

After placing a managed bean in a shared ADF memory scope, you can access the bean by using an EL expression.

For example, you might have a managed bean called `customerBean` in `pageFlowScope`. The example in the slide shows an Input Text component that stores the input value in the `customerBean`. To access the bean, use the `#{pageFlowScope.customerBean}` expression. To get and set the value of the `discountCode` attribute in `customerBean`, use the `#{pageFlowScope.customerBean.discountCode}` expression.

You can also use Java code to access the methods and properties of managed beans in memory scope. You learn how to do this later in the course.

Storing Values in Memory Scoped Attributes

Store values in memory-scoped attributes without explicitly defining a managed bean. **Not recommended**

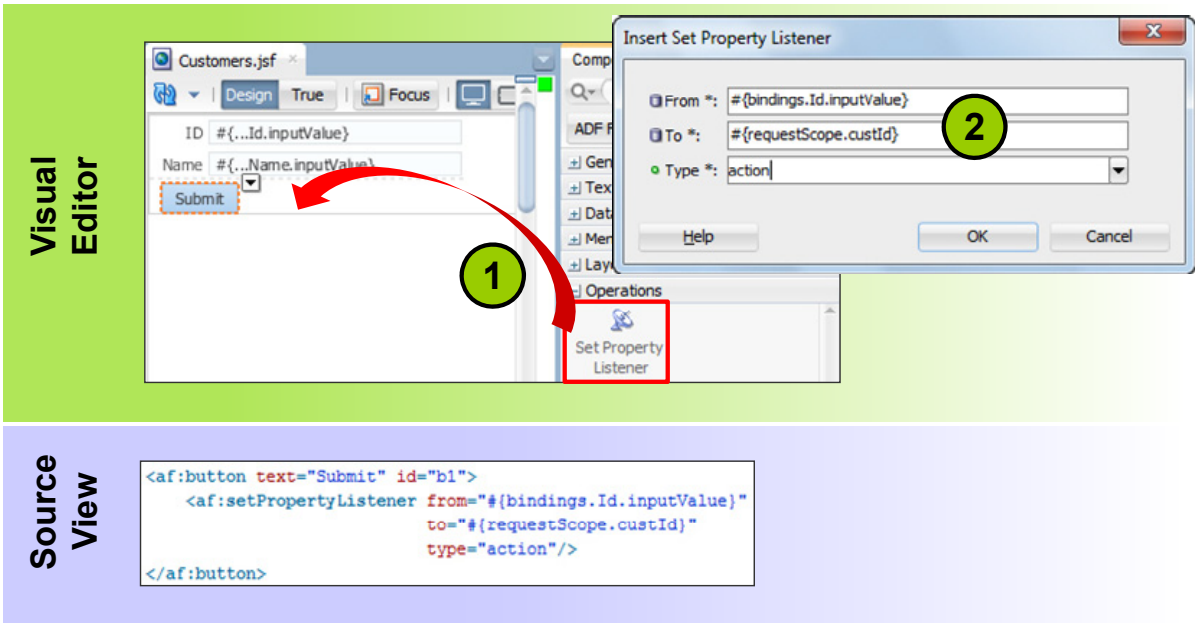


You can store values in memory-scoped attributes without explicitly defining a managed bean. To do this, you simply specify an EL expression that sets the value of the attribute, and the attribute is added to the specified scope. The attribute can then be referenced from wherever it is in scope.

For example, instead of using a managed bean to hold the value of the `discountCode` attribute (as you saw in the previous example), you can store the value directly in `pageFlow` scope and then reference the variable from any page in the same task flow. In the slide, both View A and View B use the following expression to set and get the value of the discountCode attribute: `#{pageFlowScope.discountCode}`.

As mentioned earlier, although it's technically feasible to store values in memory-scoped attributes, it is not a best practice. To protect your application from null pointer exceptions, it is recommended that you avoid using memory-scoped attributes. Instead, use managed beans. Unlike memory-scoped attributes, properties in managed beans are discoverable using EL Builder, can be documented using Javadoc, and can be initialized with default values.

Setting the Value of a Memory-Scoped Attribute by Using a Set Property Listener



You can pass values from one page to the next by using the Set Property Listener component to set the value of a memory-scoped attribute. The value is set when an action, such as a button click, is performed. The example shows how you can use a Set Property Listener component to take the value of the ID field in the binding container (specified in the From field) and store the value in the `custId` attribute in `requestScope` (specified in the To field) when the user clicks the Submit button.

It is important to note that ADF Business Components provides a smart layer that remembers the selected row when navigating between pages that use the same view object through the ADF binding layer. Therefore, if you are using the same view object through the ADF binding layer, you do not need to store values from the binding layer in memory-scoped attributes.

Using Parameters to Pass Values

- You can use:
  - Page parameters
  - Task flow binding parameters

  Stored in the binding container; defined in the page definition file

  - Task flow parameters
  - View activity parameters
  - Task flow call activity parameters

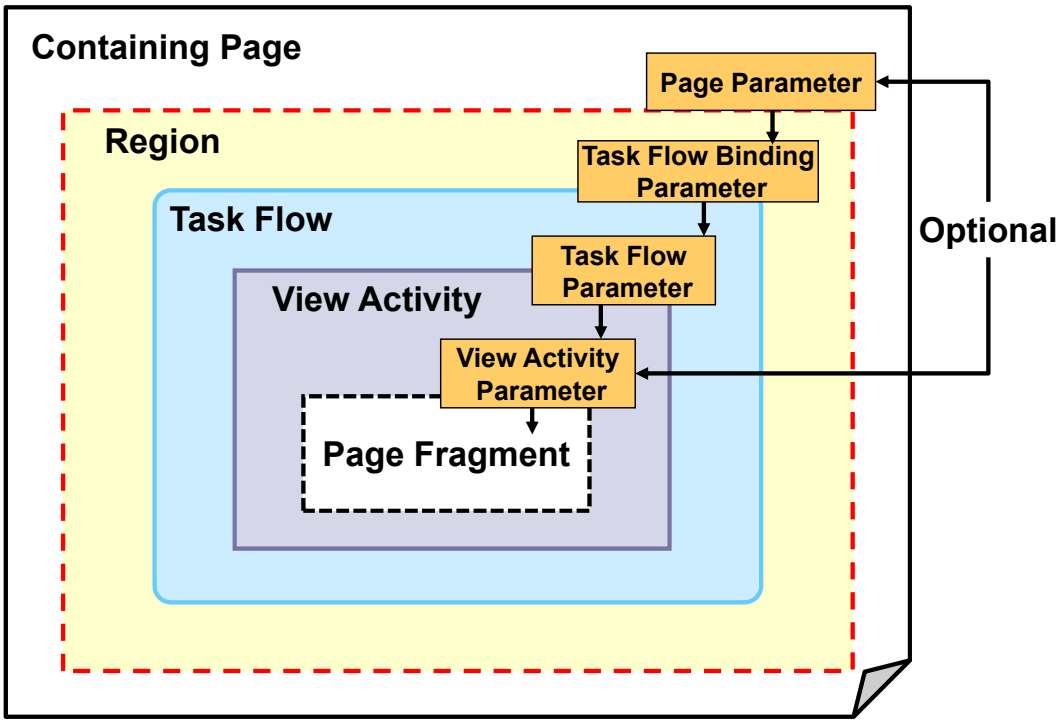
  Stored in the task flow `.xml` file; defined in the Properties window
- You set the parameter values by using expressions (not by just entering a value).
- The goal is increased reusability of pages and task flows.

Subsequent slides discuss the five parameter types listed in the slide, which you use to pass information to different parts of the application.

The examples discussed in the following slides show how to use these parameters to pass a value from a containing page to a task flow represented as a region on that page. You can actually accomplish this task without using all the parameters. However, using more types of parameters enables you to make your pages and task flows more reusable, as shown in the examples.

After we present these examples, the remainder of the lesson will cover how to pass values to a called task flow and return values to the calling task flow.

Passing Values from a Containing Page to a Reusable Page Fragment in a Region



The page fragment in this diagram is represented as a view activity in a task flow that is used as a region on a containing page. Because parameters are used at each level, the page can be developed independently of all other components.

- The containing page uses a page parameter to pass its value to the region.
- The region uses a task flow binding parameter to pass the value to the task flow, so the task flow can be developed independently of the page and reused on multiple pages.
- The task flow uses a task flow parameter to pass the value to its view activity.
- The view activity uses a view activity parameter to store the value in a managed bean that is used by the page fragment.

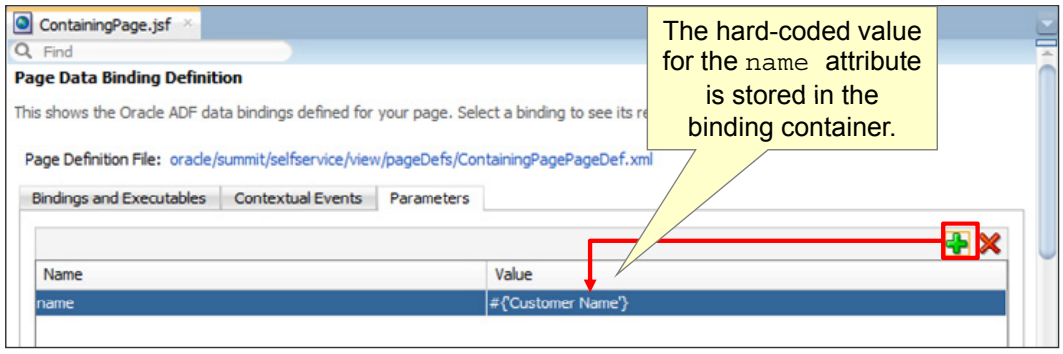
All of these parameters can be named differently, as long as the output parameter of one level has the same name as the input parameter of the next level (you will shortly learn about input and output parameters in this lesson). For example, the task flow binding parameter would have the same name as the task flow's input parameter.

The slide shows the parameters that you can define at every possible level to ensure maximum reusability. For most applications, however, this level of reusability is not required. You can simply omit the page parameters and view activity parameters, and instead you can use task flow parameters to pass values directly to a task flow that is used in the region.

Using Page Parameters

Page parameters are:

- Output parameters to pass a value *from* a page
- Stored in the binding container for the page
- Defined in the Structure window or Properties window for a page definition file



You define page parameters in the page definition of the page, and the parameters are added to the binding container at run time. The example in the slide shows a page parameter called `name` that stores a hard-coded value, `Customer Name`. Notice that the parameter expects the value to be an EL expression even for literal values. (Of course, in a real-world application, you would not hard-code the value. It appears hard-coded in this example to make it easier for you to see how the value is passed through various parameters in the application.) The value of the page parameter can be contained in a managed bean, a value in the binding container, a memory-scoped attribute, and so on.

Example:

```
<parameters>
   <parameter id="myDeptId" value="#{DepartmentBean.deptId}"/>
   <parameter id= "fname" value="#{bindings.FirstName.InputValue}"/>
   <parameter id= "lname" value="#{pageFlowScope.lastName}"/>
</parameters>
```

You can access the page parameter value in EL as `#{bindings.<param name>}`.

You can use page parameters to pass values to a page template, enabling the use of the same template for a variety of different pages. Another example in this lesson describes the use of a page parameter to pass a value to a task flow that is contained in a region on the page.

Role of the Page Parameter

Type	Description	Name	Value
Page Parameter (output)	Receives a value and stores it in the binding container.	`name` ←	`#{'My Customer'}`

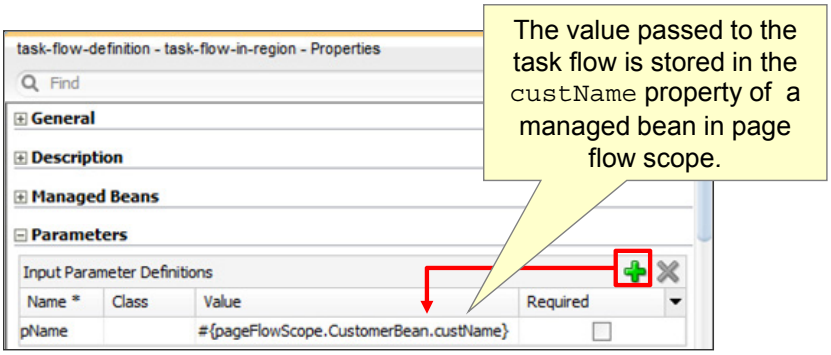
In the example scenario, there is a bounded task flow that is represented as a region on a page. The task flow contains a view activity that represents a page fragment, and the requirement is to pass a value from the containing page to the page fragment. The role of the page parameter is to receive a value and store it in the binding container. The example uses a page parameter called `name` to receive the value of the `#{'My Customer'}` expression and store it in the binding container of the page. (Keep in mind that page parameters are defined in the page definition file.)

Subsequent slides build this table to show the various parameters that are used to pass values between different kinds of UI elements.

Using Task Flow Parameters

Task flow parameters are:

- Input parameters to pass a value *to* a task flow
- Defined in the `.xml` file for the task flow (You can use the Properties window for the task flow.)



Building well-defined, parameterized task flows that can be used as regions on a page is a good way to provide reusable and highly maintainable segments of functionality across your application.

Regions on a page use parameters and page events to communicate with each other and pass context to each other without being hard-wired together and without even knowing if the other one is present on the page. For example, a region that shows a graph of types of goods sold for a particular country can take a parameter into the region that specifies the country. The same region can also take in a parameter that specifies whether or not it should display the results as a graph or as a pie chart. Using parameters enables that same region to be reused in a variety of different situations.

You define task flow parameters in the task flow's `.xml` file. You can define a task flow parameter declaratively as shown in the slide, where you see an input parameter with the name `pName` and the value `#{pageFlowScope.CustomerBean.custName}`.

The parameter works as follows:

1. When the `pName` parameter is passed to the task flow, its value is stored in the `custName` property of `CustomerBean` (a managed bean in page flow scope).
2. In the task flow, activities and pages can obtain the value of the parameter by using the `#{pageFlowScope.CustomerBean.custName}` expression.

Role of the Task Flow Parameter

Type	Description	Name	Value
Page Parameter (output)	Receives a value and stores it in the binding container	`name` ←	`#{'My Customer'}` →
Task Flow Binding Parameter (output)	(Covered later)		
Task Flow Parameter (input)	Stores the input value in a memory-scoped attribute or a managed bean	`pName` →	`#{pageFlowScope.CustomerBean.custName}`

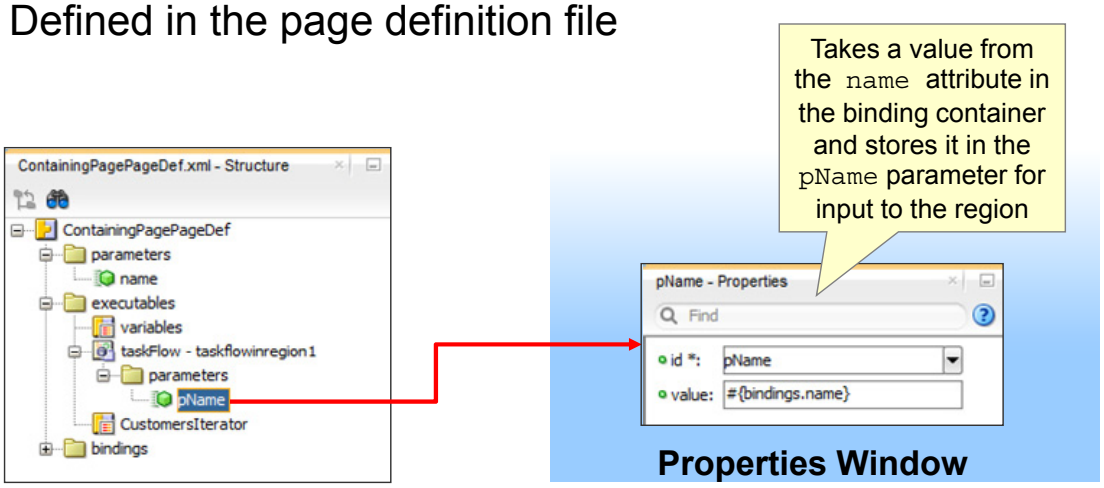
The role of the task flow parameter is to receive an input value and store it in a managed bean property (or in a memory-scoped attribute). The example in the slide shows that the value of the `pName` parameter is stored in the custName property of `CustomerBean`. `CustomerBean` is held in page flow scope so that it can be accessed from anywhere in the task flow.

But where does that value for the `pName` parameter come from? There needs to be a way to pass the value from the page output parameter, `name`, to the task flow input parameter, `pName`. To do this, you use a task flow binding parameter, which you learn about next.

Using Task Flow Binding Parameters

Task flow binding parameters are:

- Input parameters to pass a value *to* a task flow that exists as a region on a page
- Stored in the binding container for the page
- Defined in the page definition file



To pass a parameter from a containing page to a task flow in a region, you can use a task flow binding parameter, which you define as follows:

1. Define a page parameter and a task flow parameter (as shown in previous slides).
2. Pass the page parameter's value to the input parameter of the task flow. You do this by defining a task flow binding parameter in the page definition file of the containing page, either in the Structure window or in the Properties window (with the region selected in the Executables section of the Properties window).
3. Set the parameter's value to the EL value of the page parameter.

It is simpler to define a task flow binding parameter when you create the region. If the task flow has input parameters defined on it, when you drag a task flow to a page to create a region, you must define task flow binding parameters for any task flow parameters.

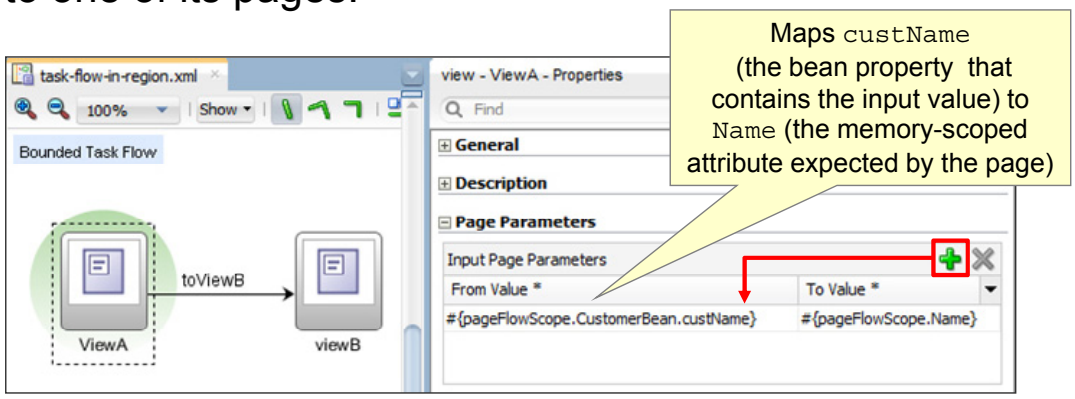
Role of the Task Flow Binding Parameter

Type	Description	Name	Value
Page Parameter (output)	Receives a value and stores it in the binding container	`name`	`#{'My Customer'}`
Task Flow Binding Parameter (output)	Stores the binding value in a parameter for input to a region	`pName`	`#{bindings.name}`
Task Flow Parameter (input)	Stores the input value in a memory-scoped attribute or a managed bean	`pName`	`#{pageFlowScope.CustomerBean.custName}`

The task flow binding parameter acts as a bridge from the page parameter to the task flow. In this example, the task flow binding parameter takes the value stored in the page parameter and stores it in an output parameter called `pName` (which has the same name as the input parameter of the task flow). The task flow is able to receive the value from the parameter and store it in a managed bean in page flow scope so that the value can be accessed from anywhere in the task flow.

Using View Activity Parameters

Use the view activity parameters to pass values from a task flow to one of its pages:



Referenced on the page:

The customer name is #{pageFlowScope.Name}.

Instead of using task flow parameters directly on the pages that are in the task flow, you can make pages more reusable by using the view activity input page parameters to map the value that is passed to the task flow to a managed bean property (or memory-scoped attribute) that is accessible by the page.

To define a view activity parameter:

1. Select the view activity in the task flow diagram. Then, under Page Parameters in the Properties window, click the Add (plus sign) button.
2. In the From Value field, enter an EL expression to specify where the parameter value is being passed from, such as `#{pageFlowScope.CustomerBean.custName}`. If the value comes from the task flow input parameter, the value should match the value specified for the task flow input parameter.
3. In the To Value field, enter an EL expression to specify where the value of the input page parameter will be stored, such as `#{pageFlowScope.Name}`. The page that is associated with the view activity can use this expression to retrieve the value.

This approach enables you to use a page in multiple task flows as the page for different views activities. The view activity acts as the mechanism for passing parameters from its task flow to its page.

Role of the View Activity Parameter

Type	Description	Name	Value
Page Parameter (output)	Receives a value and stores it in the binding container	`name`	`#{'My Customer'}`
Task Flow Binding Parameter (output)	Stores the binding value in a parameter for input to a region	`pName`	`#{bindings.name}`
Task Flow Parameter (input)	Stores the input value in a memory-scoped attribute or a managed bean	`pName`	`#{pageFlowScope.CustomerBean.custName}`
View Activity Parameter	Maps the value passed in from the task flow to a memory-scoped attribute	`#{pageFlowScope.Name}`	

Used on the page: The customer name is #{pageFlowScope.Name}.

The view activity parameter in the example in the slide simply takes the value stored in the `custName` property of `CustomerBean` and stores it in the memory-scoped attribute `Name`. The `Name` attribute is referenced on the page in the following expression: `#{pageFlowScope.Name}`

You might wonder why you would do this because `CustomerBean` is already in page flow scope and can therefore be accessed from all pages in the task flow. You do this if you need to make the page (or page fragment) reusable by using a parameter to define the page's dependency on a specific bean or memory-scoped attribute (rather than defining the dependency on the page itself).

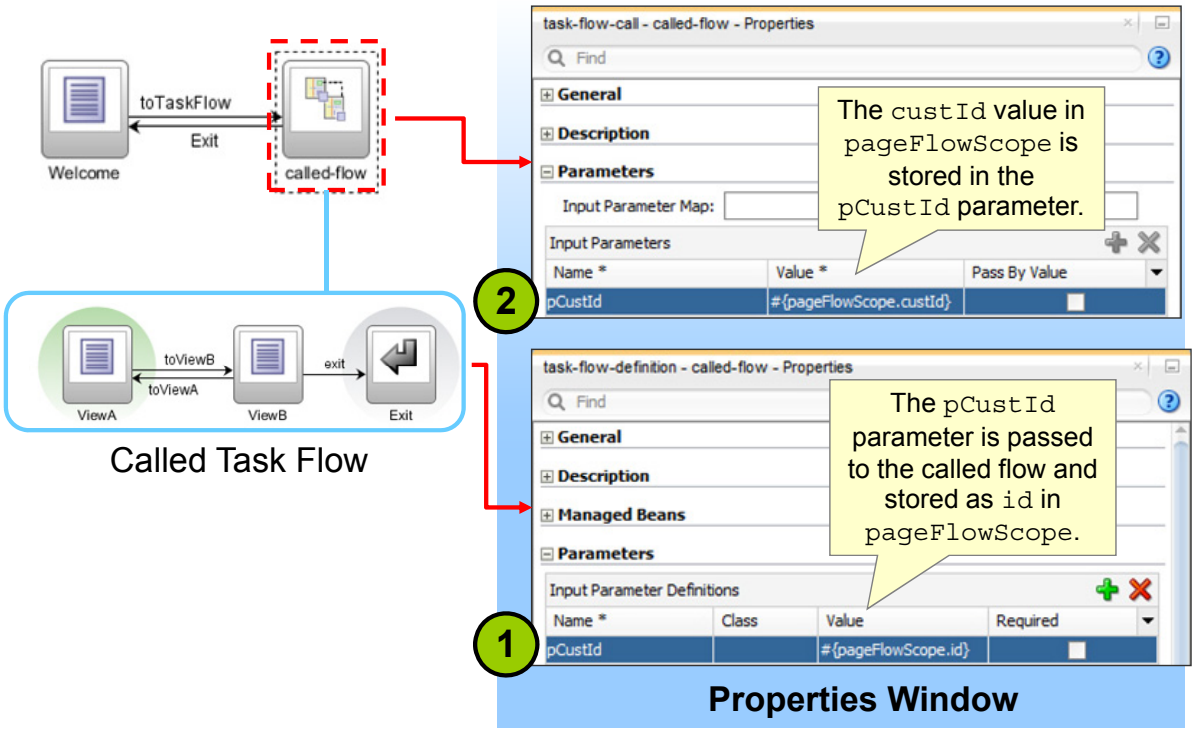
To summarize, the example uses the following parameters to ensure maximum reusability across pages and task flows:

- **Page parameter (defined on the page definition file for the containing page):** Receives a value and stores it in the binding container. In the example, the value of the `#{'My Customer'}` expression is stored in a parameter called `name` in the binding container.

- **Task flow binding parameter (defined on the page definition file for the containing page):** Acts as a bridge from the page parameter to the task flow. In the example, the task flow binding parameter takes a value from the `name` attribute in the binding container and stores it in the `pName` parameter for input to the region. The task flow binding parameter must have the same name as the input parameter of the task flow.
- **Task flow parameter (defined on the task flow):** Receives an input value and stores it in either a memory-scoped attribute or a managed bean. The example shows a task flow parameter that stores the value of the `pName` parameter in a managed bean called `CustomerBean` in page flow scope so that the value can be used anywhere in the task flow.
- **View activity parameter (defined on the view activity in the task flow):** Takes a value (stored in a memory-scoped attribute or managed bean) and stores it in a memory-scoped attribute or managed bean that is accessible to the page. In the example, the value of the `pName` parameter is stored in an attribute called `Name` in page flow scope. You can then use the following EL expression to use the value on a page:

  `#{pageFlowScope.Name}`

Passing Values to a Task Flow from a Task Flow Call Activity



Another situation in which you need to pass values to a task flow is when you use a task flow call activity to call one task flow from another. A called bounded task flow can accept input parameters from another unbounded or bounded task flow and can pass return values to the caller upon exit.

To pass an input parameter to a bounded task flow:

1. Open the called task flow. Under Parameters in the Properties window, specify a name for the input parameter. Under Value, specify an EL expression for the location where the value of the parameter will be stored when it is passed in from the calling task flow. Pages in the bounded task flow will retrieve the value from this location.
2. Go to the calling task flow and select the task flow call activity (`called-flow` in the example in the slide). Notice that the `pCustId` parameter (which is expected by the called task flow) already appears under Input Parameters. Under Value, specify an EL expression that resolves to the value that you want to pass to the called task flow. For this example, suppose that you want to pass the value of a `custId` attribute stored in `pageFlowScope` for the calling task flow, so you set the value of the `pCustId` parameter to `#{pageFlowScope.custId}`. Notice that you can also specify whether the input parameter is required.

The names of the two input parameters (defined on the task flow and the task flow call activity) must match.

In the input parameter definition for the called bounded task flow, you can specify whether the parameter is required. If a required input parameter is not received, a run-time error occurs. An input parameter definition that is identified as “not required” can be ignored during the task flow call activity creation.

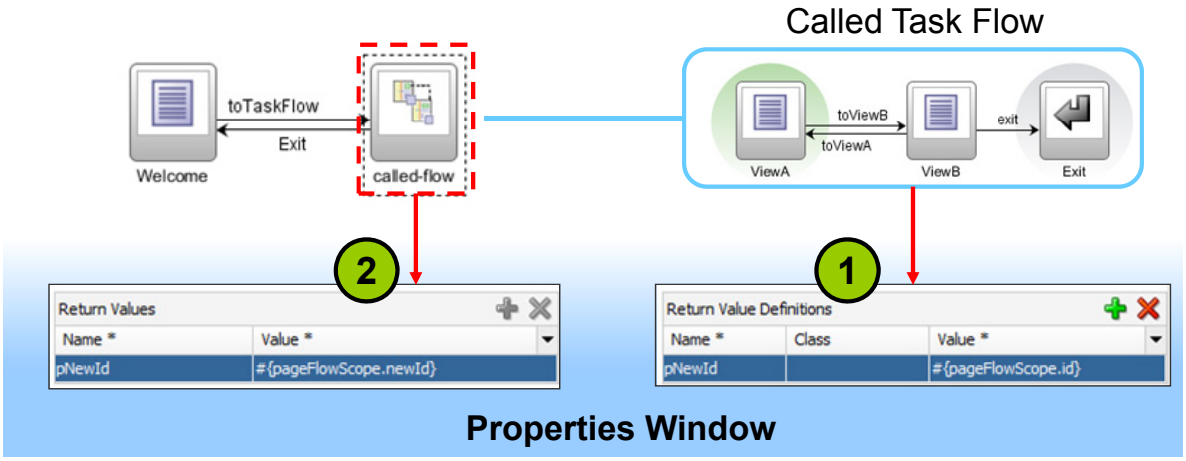
By default, input parameters are passed by reference, although you can select the Pass By Value check box in the Properties window for the task flow call activity to pass the parameter by value. Mixing pass-by-value and pass-by-reference parameters, however, can lead to unexpected behavior in cases where parameters reference each other.

Task flow call activity input parameters can be passed by reference only if managed bean objects (and not individual values) are passed. So the Pass By Value check box applies only to managed bean objects and is used to override the default setting of passing by reference. Individual values are only passed by value.

If you call a bounded task flow by using a URL rather than a task flow call activity, you pass parameters and values on the URL itself.

Returning Values to a Calling Task Flow

- To return a value, you must specify:
  - Return value definitions on the called task flow
  - Return values on the task flow call activity in the calling task flow
- Parameter names must match.



A task flow return activity causes the bounded task flow to return to the task flow that called it. The called bounded task flow can pass return values back to the calling task flow.

To return a value, you must specify:

1. Return value definitions on the called bounded task flow. These specify where the return value is to be taken from upon exit of the called bounded task flow.
2. Return values on the task flow call activity in the calling task flow. These specify where the calling task flow can find return values.

In this example, the called task flow uses the `pNewId` parameter to return the value of the `id` attribute stored in `pageFlowScope`, and the calling task flow stores the value passed as `pNewId` in an attribute called `newId` in `pageFlowScope`.

The caller of the bounded task flow can choose to ignore the return value definition values by not identifying any task flow call activity return values back to the caller.

Return values on the task flow call activity are passed back by reference. Nothing in the bounded task flow still references them, so there is no need to pass by value and make a copy.

Deciding Which Kinds of Parameters to Use

- Parameters are designed to:
  - Improve reusability
  - Encapsulate functionality
- You can use more than one type of parameter to achieve these goals.

As you have seen in this lesson, you can use a variety of parameters to pass values between UI elements in an application. The parameters that you use depend on your specific needs.

However, you should keep in mind that defining parameters is all about providing reusability. Defining parameters on a task flow improves the reusability of that task flow. Defining parameters on a page or page fragment improves the page's reusability. Parameters also make an object much more encapsulated (self-contained).

For example, if you have a bounded task flow that updates customers, you can have it take in `customerId` as a parameter. The task flow can then make sure that the correct customer is queried to edit—and there is no reliance on the consumer of that bounded task flow setting the current row in a view object in some application module that this bounded task flow uses. This ensures that the interface between consumer and producer is much better documented and separated from the internal implementation.

If you want to make the page a more reusable component, you can provide a parameter on the page for exactly the same reasons.

Summary

In this lesson, you should have learned how to:

- Define the data model to reduce the need to pass values
- Use a managed bean to hold values
- Store values in memory-scoped attributes
- Use parameters to pass values

Practice 11 Overview: Passing Values Between UI Elements

This practice covers the following topics:

- Passing parameters between task flows
- Examining attributes at different memory scopes to see what their value is at various points: in a different page, in a different page fragment, in a different region, and so on

12

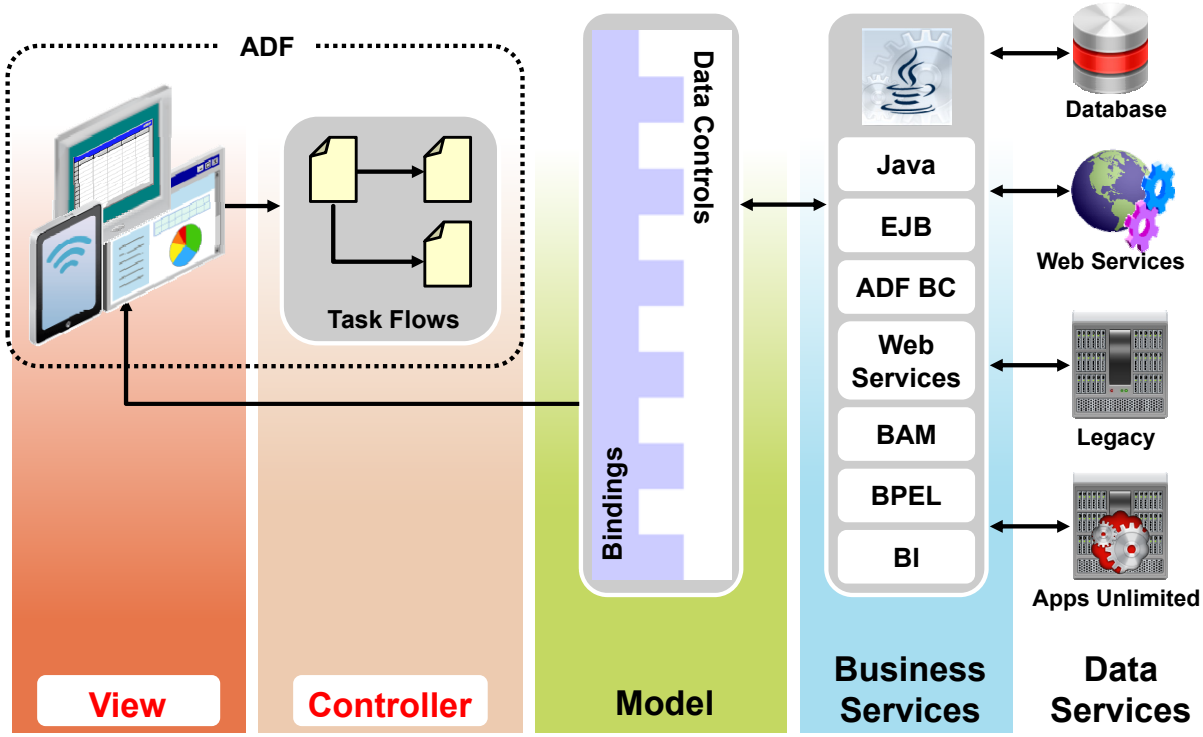
Responding to Application Events

Objectives

After completing this lesson, you should be able to:

- Describe the phases of the JSF life cycle
- Describe the different types of events and the contexts in which they are used
- Describe advanced event handling in ADF Faces
- List other types of server events that are used by ADF Faces components
- Use partial page rendering (PPR)

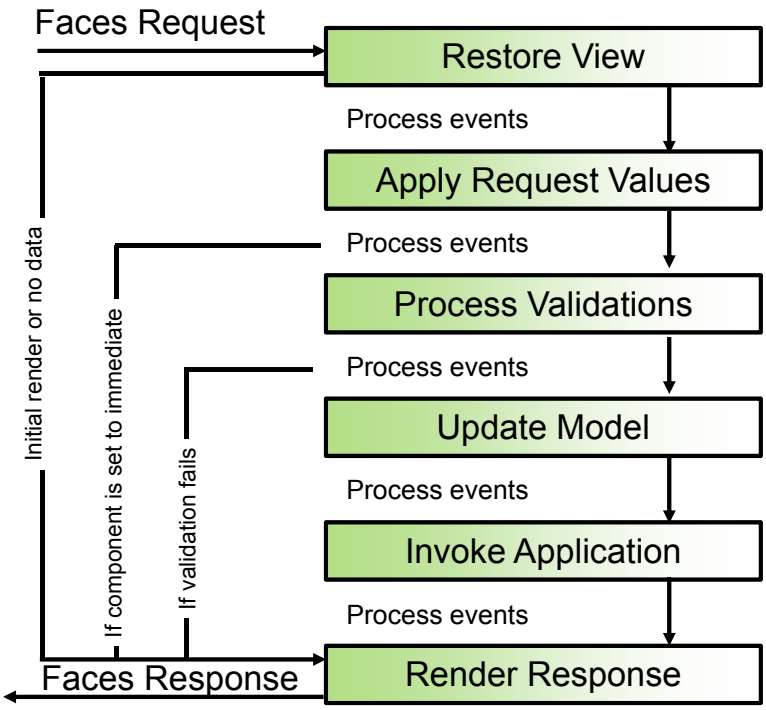
Responding to Application Events



In previous lessons, you learned that JSF is a component-based and event-based architecture. You learned about the various kinds of UI components that are available for building JSF pages, and you learned how to build command components (such as buttons and links) that respond to simple events such as mouse clicks.

In this lesson, you learn more about event handling in ADF. You learn about the phases of the JSF life cycle and about how ADF extends the JSF life cycle by providing phases and events that developers can use to augment standard ADF behavior. You also learn how to implement partial page rendering to render part of a page in response to an event.

JSF Lifecycle Phases



When a JSF page is submitted and a new page is requested, the JSF page request life cycle is invoked. This life cycle handles the submission of values on the page, validation for components on the current page, navigation to and display of the components on the resulting page, and saving and restoring state. The JSF lifecycle phases use a UI component tree to manage the display of the components.

This tree is a runtime representation of a JSF page: Each UI component tag in a page corresponds to a UI component instance in the tree. The FacesServlet object manages the page request life cycle in JSF applications. The FacesServlet object creates an object called FacesContext, which contains the information necessary for request processing, and invokes an object that executes the life cycle.

The diagram in the slide shows the following phases of the JSF life cycle:

Restore View

The component tree is established. If this is not the initial rendering (that is, the page was submitted back to the server), the tree is restored with the appropriate state. If this is the initial rendering, the component tree is created and the life cycle jumps to the Render Response phase.

Apply Request Values

Each component in the tree extracts new values from the request parameters (using its decode method) and stores the values locally. Most associated events are queued for later processing. If a component has its `immediate` attribute set to `true`, the validation, conversion, and events associated with the component are processed during this phase.

Process Validations

Local values of components are converted from the input type to the underlying data type. If the converter fails, this phase continues to completion (all remaining converters, validators, and required checks are run), but at completion, the life cycle jumps to the Render Response phase.

If there are no converter failures, the required attribute on the component is checked. If the value is `true` and the associated field contains a value, any associated validators are run. If the value is `true` and there is no field value, this phase completes (all remaining validators are executed), but the life cycle jumps to the Render Response phase.

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

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

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

Update Model Values

The component's validated local values are moved to the model and the local copies are discarded.

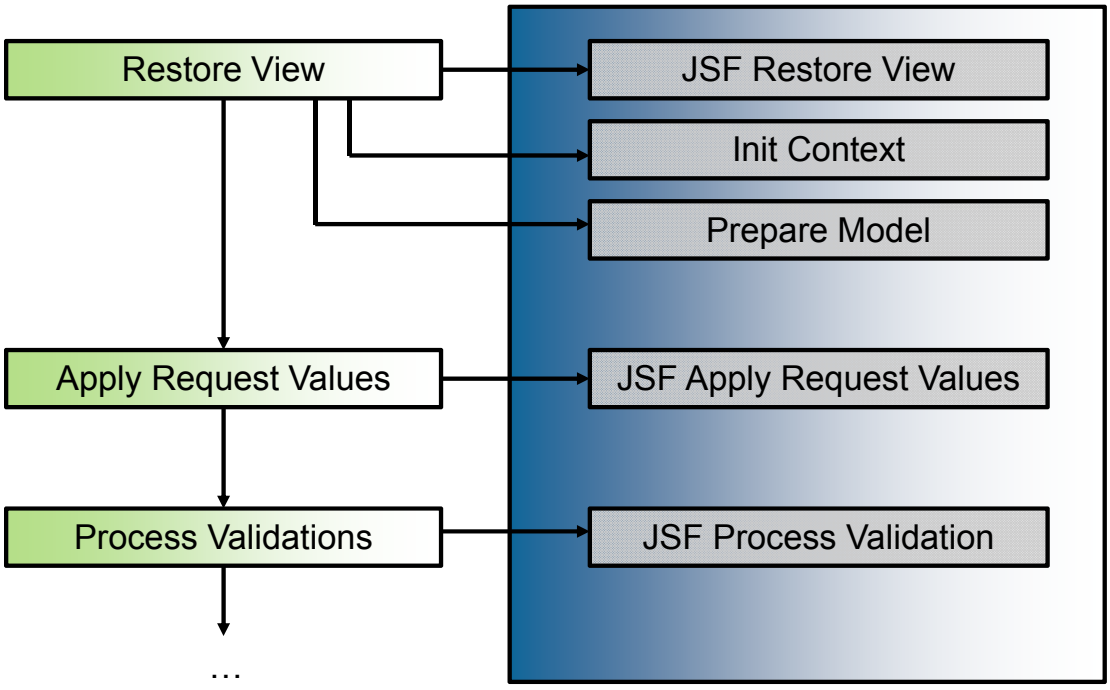
Invoke Application

Application-level logic (such as event handlers) is executed.

Render Response

The components in the tree are rendered. State information is saved for subsequent requests and for the Restore View phase.

Phases of the ADF Life Cycle



As you saw on the previous page, the ADF Faces life cycle extends the JSF life cycle, providing additional functionality, such as a client-side value life cycle, a subform component that allows you to create independent submittable sections on a page without the drawbacks of using multiple forms on a single page, and additional scopes. The ADF page life cycle handles preparing and updating the data model, validating the data at the model layer, and executing methods on the business layer. The ADF page life cycle uses the binding container to make data available for easy referencing by the page during the current page request.

The phases listed on the left of the diagram show standard JSF lifecycle phases; the boxes on the right show the ADF lifecycle phases.

Phases of the Lifecycle

Restore View: The URL for the requested page is passed to the bindingContext object, which finds the page definition file that matches the URL. The component tree of the requested page is either newly built or restored. If the component tree is empty (that is, if there is no data from the submitted page), the page life cycle proceeds directly to the Render Response phase.

If any discrepancies between the request state and the server-side state are detected, an error is thrown and the page life cycle jumps to the Render Response phase.

JSF Restore View: Provides before and after phase events for the Restore View phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener is registered with the Restore View phase. The Initialize Context phase of the ADF Model page life cycle listens for the after (JSF Restore View) event and then executes. ADF Controller uses listeners for the before and after events of this phase to synchronize the server-side state with the request. For example, it is in this phase that browser back button detection and bookmark reference are handled. After the before and after listeners are executed, the page flow scope is available.

Initialize Context: The page definition file is used to create the bindingContainer object, which is the runtime representation of the page definition file for the requested page. The LifecycleContext class that is used to persist information throughout the ADF page lifecycle phases is instantiated and initialized with values for the associated request, binding container, and life cycle.

Prepare Model: The ADF page life cycle enters the Prepare Model phase by calling the BindingContainer.refresh(PREPARE_MODEL) method. During the Prepare Model phase, BindingContainer page parameters are prepared and then evaluated. If parameters for a task flow exist, they are passed into the flow. If the incoming request contains no POST data or query parameters, the life cycle forwards to the Render Response phase. If the page was created using a template, and if that template contains bindings using ADF Model, the template's page definition file is used to create the binding container for the template. The container is then added to the binding context. If any taskFlow executable bindings exist (for example, if the page contains a region), the taskFlow binding creates an ADF ControllerViewPortContext object for the task flow, and any nested binding containers for pages in the flow are then executed.

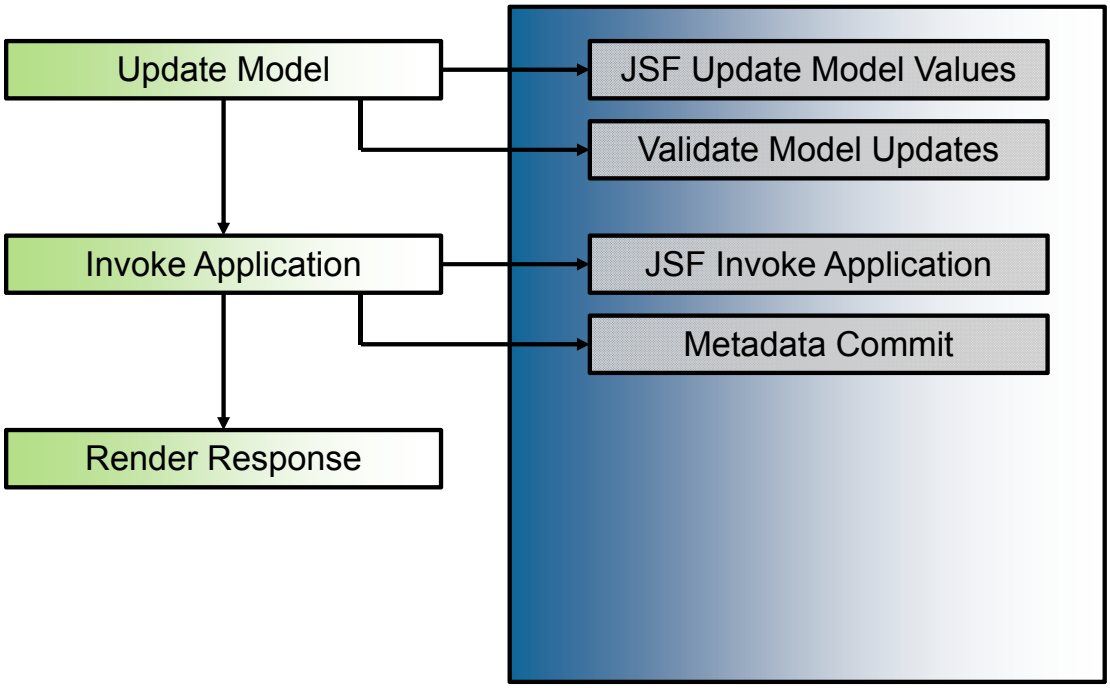
Apply Request Values: Each component in the tree extracts new values from the request parameters (using its decode method) and stores those values locally. Most associated events are queued for later processing. If you have set a component's immediate attribute to true, the validation, conversion, and events that are associated with the component are processed during this phase and the life cycle skips the Process Validations, Update Model Values, and Invoke Application phases.

JSF Apply Request Values: Provides before and after phase events for the Apply Request Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener is registered with the Apply Request Values phase.

Process Validations: Local values of components are converted and validated on the client. If there are errors, the life cycle jumps to the Render Response phase. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued onFacesContext, and any value change events are delivered. Exceptions are also caught by the binding container and cached.

JSF Process Validations: Provides before and after phase events for the Process Validations phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener is registered with the Process Validations phase.

Phases of the ADF Life Cycle



Update Model Values: The component's validated local values are moved to the model and the local copies are discarded. For any updateable components (such as an inputText component), corresponding iterators are refreshed if the refresh condition is set to the default (deferred) and the refresh condition (if any) evaluates to true.

JSF Update Model Values: Provides before and after phase events for the Update Model Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener is registered with the Update Model Values phase.

Validate Model Updates: The updated model is now validated against any validation routines that are set on the model. Exceptions are caught by the binding container and then cached.

Invoke Application: Any action bindings for command components or events are invoked.

JSF Invoke Application: Provides before and after phase events for the Invoke Application phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener is registered with the Invoke Application phase.

Metadata Commit: Changes to runtime metadata are committed. This phase stores any runtime changes that are made to the application using the Metadata Service (MDS).

Partial Page Rendering (PPR)

PPR:

- Is enabled by ADF Faces
- Enables the updating of only a portion of a page
- Requires server round-trip:
  - Re-renders only a portion of the server-side component tree
  - Downloads only the appropriate fragment of HTML
- Implements certain ADF Faces patterns
  - Single-component refresh
  - Cross-component refresh
- Can be enabled declaratively or programmatically

Like standard JSF components, ADF Faces command components deliver ActionEvent events when the components are activated, and ADF Faces input and select components deliver ValueChangeEvent events when the component local values change. Although ADF Faces adheres to standard JSF event-handling techniques, it also enhances event handling. Unlike standard JSF events, ADF Faces events support AJAX-style partial postbacks to enable partial page rendering (PPR). Instead of full page rendering, ADF Faces events and components can trigger partial page rendering, that is, only portions of a page refresh upon request. In ADF Faces components, this is implemented by a hidden `IFrame`, which is automatically added to a webpage when one of the following ADF Faces elements is used: `af:document`, `afh:body`, or `af:panelPartialRoot`.

Two main AJAX patterns are implemented with PPR:

- **Single-component refresh:** Implemented natively. For example, the ADF Faces table component comes with a built-in functionality that enables you to scroll through the table, sort the table by clicking a column header, mark a line or several lines for selection, and expand specific rows in the table—all through declarative property settings with no coding needed
- **Cross-component refresh:** Implemented declaratively or programmatically by the application developer by defining ADF Faces UI components to act either as a trigger for a partial update or as a partial listener to be updated

Guidelines for Using PPR

- Purposes of PPR:
  - Improves application performance
  - Improves the user experience
  - Does not require a full page refresh for changes
  - Sends smaller payloads across internet connections
- PPR should not be used in the following situations:
  - When navigating to another page
  - When response times may be long
  - When multiple sections of the page need to be redrawn
- PPR might cause accessibility issues.

ORACLE

When PPR is implemented correctly, it improves application performance as follows:

- Rendering performance is improved by generating and re-rendering only a subset of the page.
- Network traffic is reduced by sending only a subset of the page's contents to the browser.
- User perception of performance is improved because of not spending time looking at a blank page.

When performance improvement is not possible with PPR, it should not be implemented.

PPR should not be invoked in the following contexts:

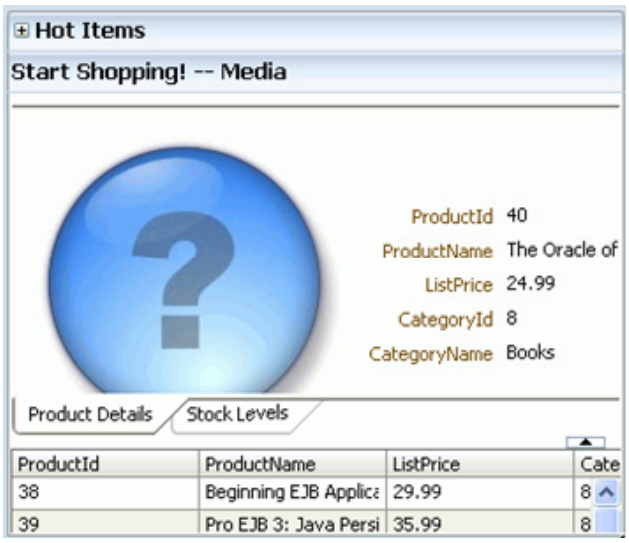
- When navigating to another page, because some page elements, such as page titles, do not change during PPR
- When response times may be long (user is blocked during a partial page submit), as in the following:
  - Database queries or database maintenance operations
  - Processes that demand significant middle-tier processing

- When multiple sections of the page need to be redrawn, such as:
  - Action or choices that affect more than half the content of the page
  - Inline messaging, which features a message box at the top of the page and may insert inline messages below multiple fields

Using PPR may cause some accessibility issues because accessibility readers read from the top down.
If there are changes in the page, they may not be picked up by the screen reader.

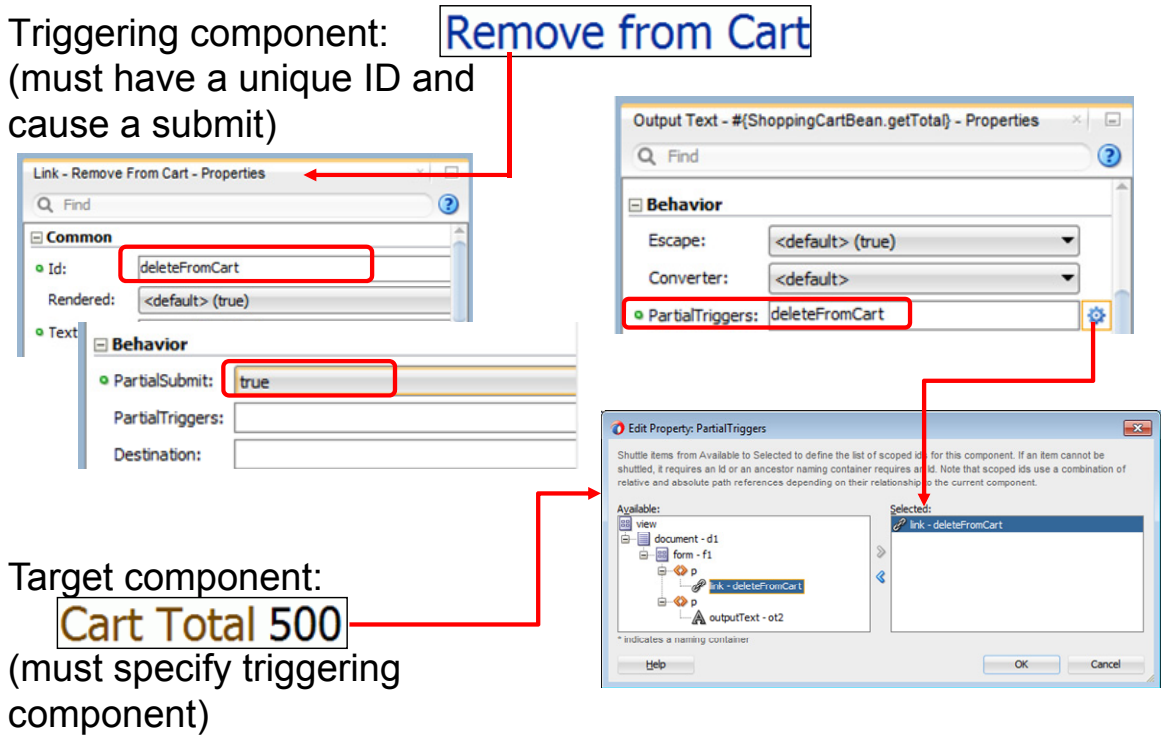
Native PPR

Example: Native PPR is built into some components.



As an example, the slide shows a page with two `af:showDetailItem` components in a tabbed panel near the bottom of the display: one for Product Details and one for Stock Levels. When the user clicks one of the tabs, the page renders only that portion of the page and not the whole page. This behavior is built into some components; you do not need to add any code.

Enabling PPR Declaratively



For a component to trigger another component to refresh, the triggering component must have a unique ID (which is a valid XML name) and must cause a submit when an appropriate action takes place. For a component to be refreshed as triggered by another component, it must declare which other components are the triggers.

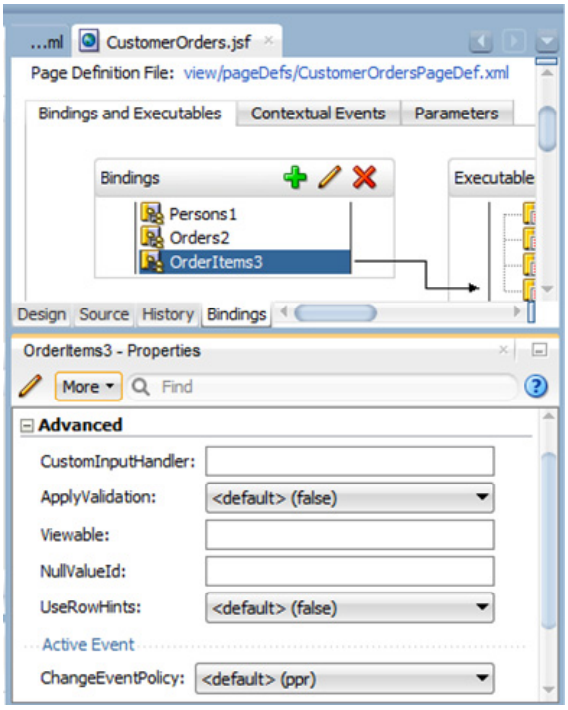
The following are the three main component attributes that are used in PPR:

- **`autoSubmit`:** When set to `true`, and when an appropriate action takes place (such as a value change), the component automatically submits the enclosing form. For PPR, you can use this with a listener attribute bound to a method that performs some logic when an event that is based on the submit is launched.
- **`partialSubmit`:** When set to `true`, the page partially submits when the button or link is clicked. You can use this with an `actionListener` method that performs some logic when the button or link is clicked.
- **`partialTriggers`:** Use this attribute to list the IDs of components whose change events are to trigger this component to be refreshed. Use a space between multiple IDs. When any of those triggering components is submitted or updated (for example, through an `autoSubmit`), this component is also updated. All rendered components support the `partialTriggers` attribute.

Enabling Automatic PPR

To enable automatic PPR:

1. Select a binding in the page definition file.
2. Set ChangeEventPolicy to `ppr` (default).
3. The binding layer is notified of value changes.
4. Adds components to the PPR target list.



Configuring PPR manually can be tedious and error-prone, especially when back-end business logic is added to the mix. The complexity rises when page developers use back-end components that are developed by another team that might not be aware of how the back-end logic changes values.

To resolve this difficulty, you can enable automatic partial page refresh on the bindings of any page. This causes the automatic repainting of components whose values change as a result of back-end business logic. At run time, the binding layer is notified of value changes and appropriate components are added to the PPR targets list.

This feature enables you to avoid much of the manual PPR configuration so that you can focus on building the UI or business logic.

Automatic PPR applies to any UI component that has an ADF binding. If there are components that do not have data bindings, they must be manually enabled.

Using the `immediate` Attribute

- Action source components skip:
  - Process Validation
  - Update Model
  - Invoke Application
- The `immediate` attribute is useful for buttons, like af:command, that result in navigation (for example, a Cancel button).
- Buttons that do not provide navigation will still skip the same phases.
- Use the `immediate` property on an input component when it needs to be validated before other components.

You can use the `immediate` attribute to allow the processing of components to move up to the Apply Request Values phase of the life cycle. When action source components (such as a button) are set to `immediate`, events are delivered in the Apply Request Values phase instead of in the Invoke Application phase. The actionListener handler then calls the Render Response phase. For example, you might want to configure a Cancel button to be `immediate` and have the action return a string that is used to navigate back to the previous page.

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

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

Value Change Events

- Input components raise value change events.
- Code for the value change listener can be in the backing bean.
- Stub code is generated by JDeveloper.
- The value change event is registered in the page source.
- Value change events are processed in the Invoke Application phase of the life cycle.
- Value change events fire before action events.

Value change events are attached to input and select components. When there is a change to the value of a component, a value change event is added to the queue of application events to be executed. Just like action events, the code can be either in the backing bean or in a separate class. JDeveloper creates the method stub in the backing bean and registers the listener in the page source by using the `valueChangeListener` property of the component tag.

The registered value change listener is added to the queue of listeners to be processed before any action events. Action events, which are linked to navigation, are executed last.

Using Value Change Event Listeners

- Select the input component in the visual editor.
- Edit the `valueChangeListener` property and select a bean and method, or create a new one.
- JDeveloper:
  - Creates the method in the managed bean if it does not already exist
  - Adds the method to the `valueChangeListener` property in the page source
- Add your code.

You create value change events by entering the name of the method in the `valueChangeListener` property of the component. If you are creating a new method, JDeveloper adds the method stub to the Java class. You then add your custom code.

Action Events

- Command components raise action events.
- Code to respond to the action event can be in the backing bean or external class.
- Methods called by action events return an outcome string that is used by the controller for navigation.
- Stub code is generated by JDeveloper on demand.
- The action is registered in the page source.
- Action events are called in the Invoke Application phase of the life cycle.
- Action methods are the last to execute after other listeners, such as `valueChange` events.

ORACLE

Action events are events that are associated with command components such as buttons, links, and menu items.

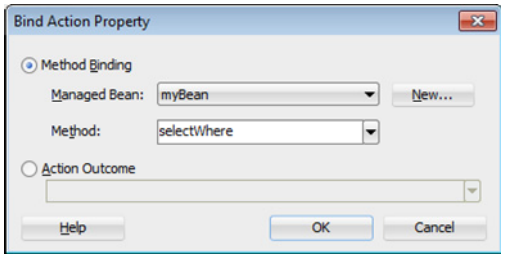
An action event enables you to define an action method that returns an outcome to be used for navigation. Instead of entering a control flow outcome directly in the `action` property of a command component, you refer to a bean method that returns a string that corresponds to the control flow outcome. An example is shown in the next slide.

Action method code can be either in the page-backing bean or in a separate class. The benefit of having the code in the backing bean is that the code that is associated with a specific page is all local to the page (single source). However, the benefit of a separate class is reusability. For example, if you have standard code that should be executed regardless of the page, the separate class approach makes more sense. If you use the backing bean approach, JDeveloper creates a method stub for you; all you have to do is add your custom code. JDeveloper also registers the action in the page source.

When the page is submitted, the action method is added to the queue of application events to be executed. Action methods are the last to fire in the application phase of the life cycle. For this reason, they are generally used to determine navigation.

Creating Action Methods

- To create an action method:
  1. Invoke the action binding editor.
  2. Select an existing bean and a method (or create a new one).



- JDeveloper does the following:
  - (If the bean does not exist) Creates a Java class and registers it as a managed bean
  - (If the method does not exist) Creates a method stub in the managed bean class
  - Adds the method to the `action` property of component
- Add your own code.
- Action methods return a `String` outcome.

To create an action method in JDeveloper:

1. Invoke the action-binding editor by either double-clicking the component in the editor or right-clicking it in the Structure window. Then select “Create Method Binding for Action.”
2. In the action-binding editor, you can accept the proposed method name or change it to a name of your choice. You can select to create a new bean and method if desired.

You then add whatever code you need to the action method. Remember that action methods can be used for controlling the UI, modifying values, or navigation; however, action methods return a `String` outcome.

For example, the `myBean.selectWhere` method shown in the slide can have code that is similar to the following:

```
public String selectWhere() {
  if <some condition> {
    return (“cart”); /* navigate to shopping cart */
  }
  else {
    return(“prod”); /* navigate to Product page */
  }}
```

Using Action Event Listeners

Action listeners differ from action methods.

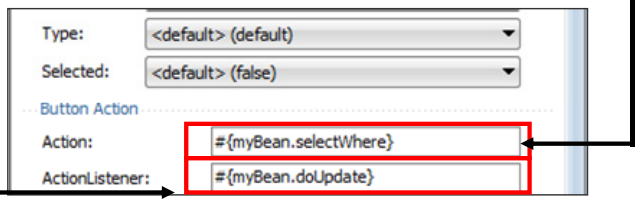
Action listeners

- Contain code that responds to an action
- Do not return a value
- Execute after value change listeners

Both are initiated by the same action (such as a button click).

Action methods

- Contain code that responds to an action
- Return a value
- Execute after value change listeners

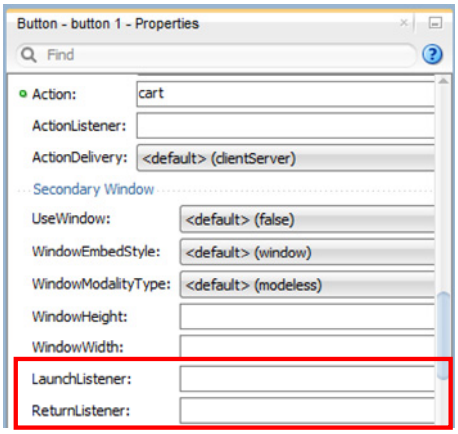


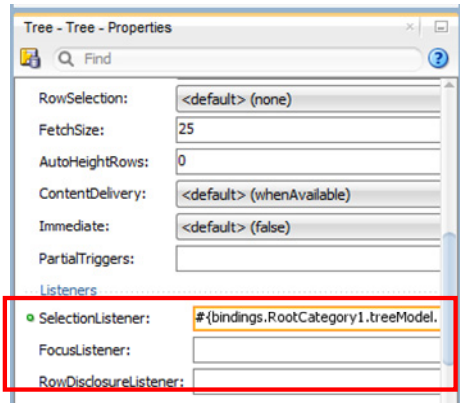
ORACLE

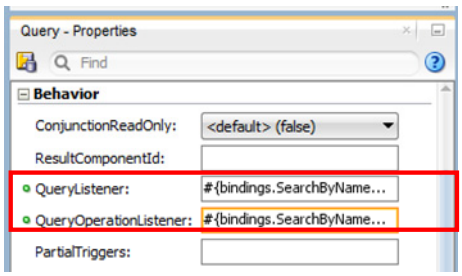
A command component, such as a button or a link, generates an action when clicked. An action listener is an event listener that defines code to execute in response to that action. The action listener contains code that processes the action event object that is passed to it by the command component.

Other ADF Faces Server Events

The Properties window of ADF Faces components displays available event listeners.







In addition to server-side action and value change events, ADF Faces components also fire other kinds of server events. ADF Faces server events, and the components that generate them, include the following:

- **`ActionEvent`:** All command components
- **`DialogEvent`:** `dialog`
- **`DisclosureEvent`:** `showDetail`, `showDetailHeader`, `showDetailItem`
- **`FocusEvent`** (generated when focusing in on a specific subtree, which is not the same as a client-side keyboard focus event): `tree`, `treeTable`
- **`LaunchEvent`:** All command components
- **`LaunchPopupEvent`:** `inputListOfValues`, `inputComboboxListOfValues`
- **`PollEvent`:** `poll`
- **`QueryEvent`:** `query`, `quickQuery`, `table`
- **`QueryOperationEvent`:** `query`, `queryCriteria`, `quickQuery`
- **`RangeChangeEvent`:** `table`
- **`RegionNavigationEvent`:** `region`

Using Tree Model Methods in Selection Listeners

You can do the following programmatically:

- Determine whether a row has children: `isContainer()`
- Access children of current row: `enterContainer()`
- Revert to parent collection: `exitContainer()`

Overview Package Class Use Tree Deprecated Index Help

Oracle Fusion Middleware Java API Reference for Oracle ADF Faces 12c (12.1.2) E23185-01

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

oracle.adf.view.rich.model

Class TreeModel

java.lang.Object
  └ javax.faces.model.DataModel
    └ org.apache.myfaces.trinidad.model.CollectionModel
      └ org.apache.myfaces.trinidad.model.TreeModel
        └ oracle.adf.view.rich.model.TreeModel

All Implemented Interfaces:

Iterable, org.apache.myfaces.trinidad.model.LocalRowKeyIndex, org.apache.myfaces.trinidad.model.RowKeyIndex, org.apache.myfaces.trinidad.model.TreeLocalRowKeyIndex

ORACLE

The ADF Faces tree components use a model to access the data in the underlying hierarchy. The specific model class is `oracle.adf.view.rich.model.TreeModel`, which extends `CollectionModel`. You can use code from this class in a managed bean method and call it from a tree's selection listener.

The `TreeModel` is a collection of rows. It has an `isContainer()` method that returns `true` if the current row contains child rows. To access the children of the current row, call the `enterContainer()` method. Calling this method results in the `TreeModel` instance changing to become a collection of the child rows. To revert to the parent collection, call the `exitContainer()` method.

The `oracle.adf.view.faces.model.ChildPropertyTreeModel` class can be useful when you are constructing a `TreeModel`.

Summary

In this lesson, you should have learned how to:

- Describe the phases of the JSF life cycle
- Describe the different types of events and the contexts in which they are used
- Describe advanced event handling in ADF Faces
- List other types of server events that are used by ADF Faces components
- Use partial page rendering (PPR)

Practice 12 Overview: Responding to Application Events

This practice covers the following topics:

- Using EL and PPR to set the conditional display of certain attributes
- Implementing PPR
- Implementing event listeners

13

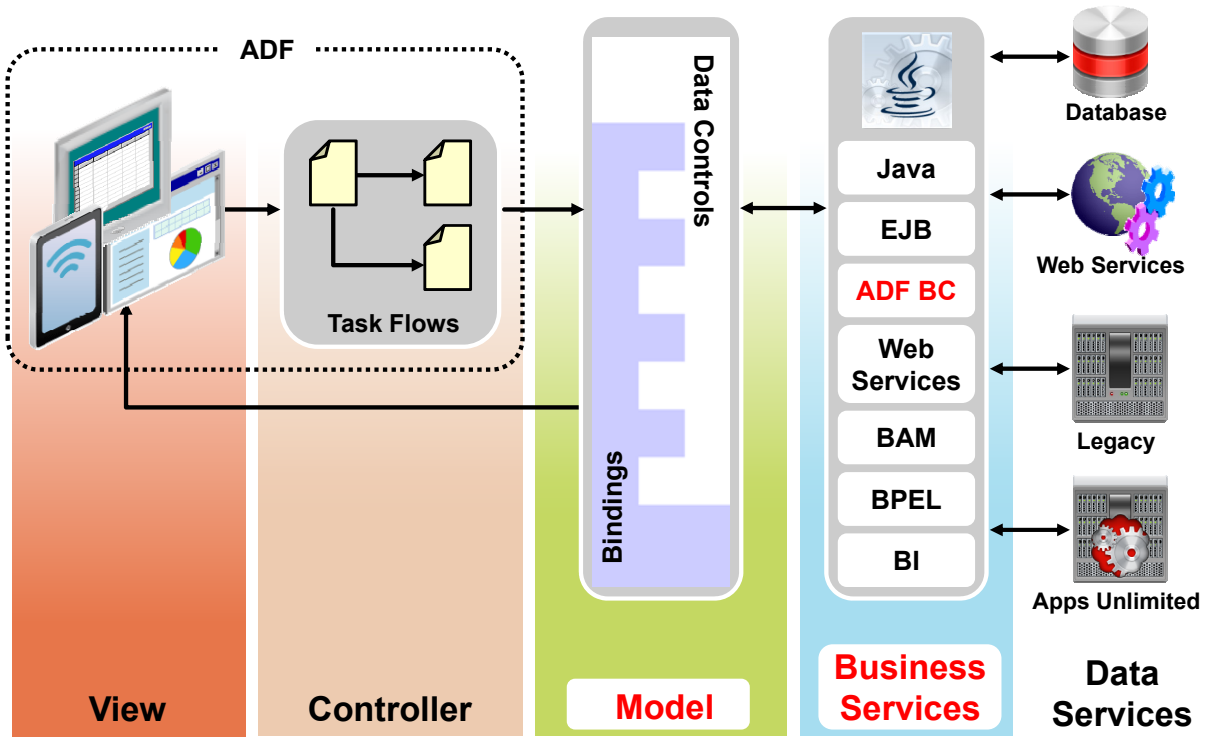
Programmatically Implementing Business Service Functionality

Objectives

After completing this lesson, you should be able to:

- Generate Java classes for business components
- Override class methods
- Implement programmatic modifications
- Add service methods to an application module
- Use business component client APIs
- Access ADF bindings programmatically

Programmatically Customizing the Data Model



In earlier lessons, you learned about the declarative features of Oracle ADF and how to modify those features by using the editors and wizards available in JDeveloper. This lesson does not explore all parts of an application that you can modify programmatically. Instead, it focuses on adding code to the business service layer, and on accessing data and operations from the view layer through ADF bindings in the model layer.

In this lesson, you learn how to how to implement business logic programmatically. You extend the behavior of the framework by adding custom logic to your application code. You add business logic to entity and view object classes and add service methods to an application module. You also use ADF BC client APIs to change the behavior of business components, and then you test the modified functionality. Finally, you add logic to perform validation programmatically.

Deciding Where to Add Custom Code

Common use cases for adding custom code:

- Business components
  - Override default behavior.
  - Declare custom methods.
  - Define validation logic.
- Model
  - Access data and operations through ADF bindings.
- View
  - Add UI-specific code.
- Controller (not covered in this lesson)
  - Define logic for navigating programmatically.

Developing ADF applications is largely a declarative experience. As with any framework, however, you might occasionally want to have the flexibility and power of driving the framework programmatically. You need the ability to add logic and functionality to your application beyond what the framework provides. In ADF, you do this by writing Java code. In fact, ADF provides a rich set of APIs that you can use to add logic to your application at every layer.

In the business services layer, you can add custom code to business components to do the following:

- **Override default behavior:** Override default behavior by overriding framework methods and providing your own implementation. For example, you can override the `doDML()` method on the entity object to augment or change the standard `INSERT`, `UPDATE`, or `DELETE` DML operation that the framework performs for the entity object. You can add logging logic, or check to see if the user has the correct privileges.
- **Declare custom methods:** Add custom application logic by declaring custom methods. For example, rather than simply using built-in operations to create or delete employees, you can create custom methods, such as `hireEmployee()` and `fireEmployee()`, that implement additional application logic beyond simply creating or deleting a record.

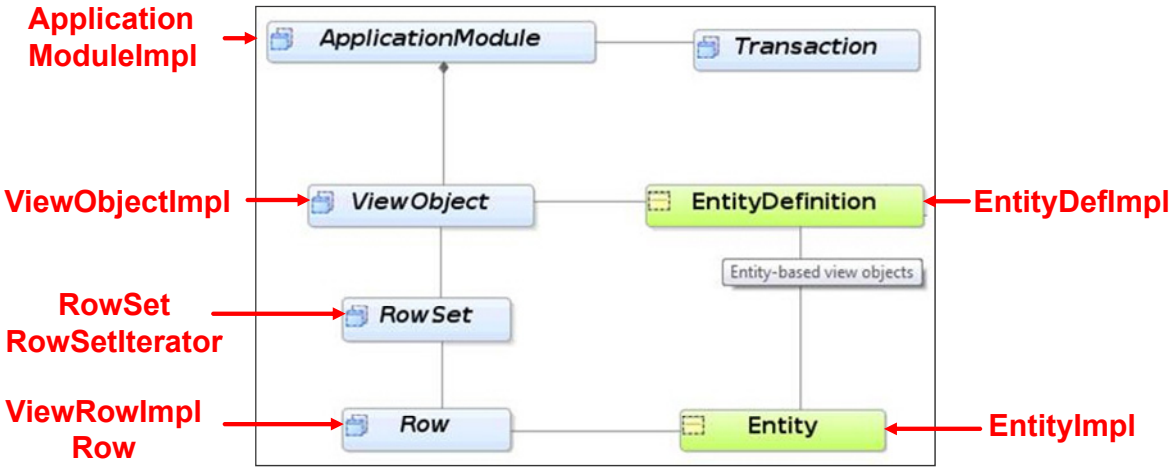
- **Define validation logic:** Define validation rules that call custom methods. Or override the framework method that performs validation.

You can also write code that uses ADF binding classes to access data and operations defined in the business services layer. For example, in the UI layer of an application, you might want to provide a control that triggers actions defined in the business services layer. To do this, you code against the model.

As you learned in an earlier lesson, you can add UI-specific code in the view layer by creating backing beans.

Another common use case is programmatic navigation, but that topic is beyond the scope of this course.

Overview of the Framework Classes for ADF Business Components



Classes that implement ADF BC

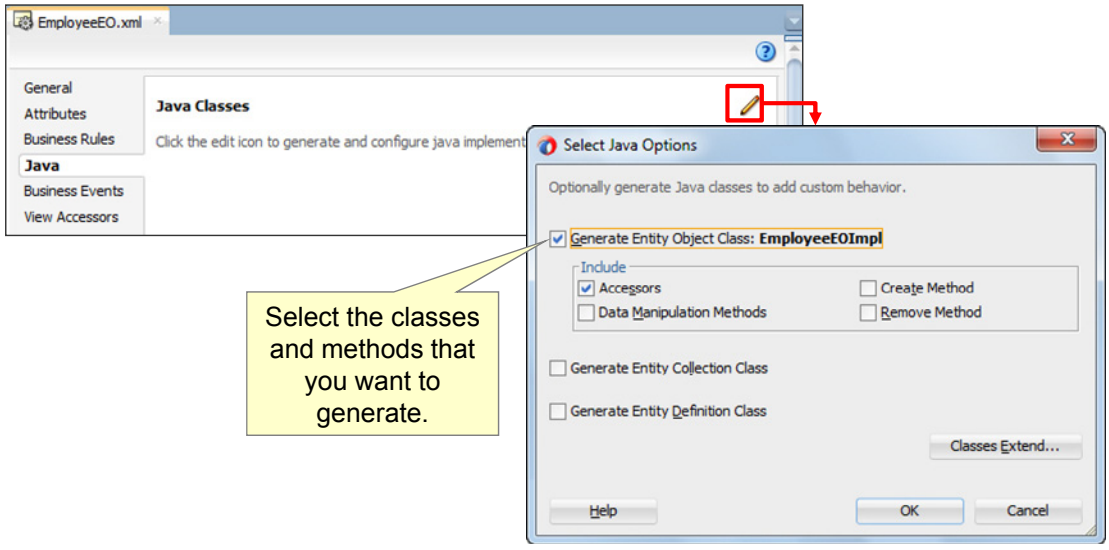
Behind every declarative feature in Oracle ADF, there is a Java framework class that implements the feature. Knowing how a feature maps to the core framework classes is key to understanding where to add custom logic to your application. The slide shows how some of the business component objects that you have used declaratively map to implementation classes (shown in red) in the framework. To modify the behavior of business components, you can generate subclasses of the implementation classes and add your own logic, either by overriding existing methods or adding your own custom methods.

The framework classes that you are likely to modify are described in more detail in the slides that follow.

Generating Java Classes to Add Custom Code

To generate a Java class for a business component:

1. On the Java tab of the overview editor, click Edit.
2. Select the classes to generate.



By default, the only files generated for business components are the metadata files in XML. If you need to add Java code, you can generate Java classes on the Java tab of the overview editor for a business component:

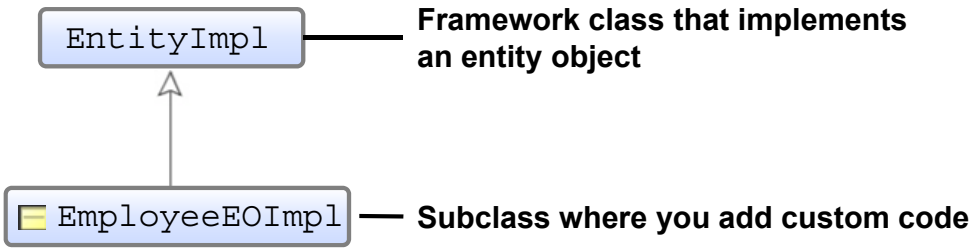
1. Click the Edit (pencil) icon.
2. In the Select Java Options dialog box, select the check boxes for the classes and methods that you want to generate.
3. Click OK to generate the class.

Just as with the XML definition file, JDeveloper keeps the generated code in your Java classes up to date with any changes you make in the editor.

To delete a Java class that has already been generated, follow the same process, but deselect the classes that you want to delete. Any code that you may have added to a class is lost when you delete the class.

Customizing Entity Objects Programmatically

Generate a subclass of `EntityImpl`:



Common use cases:

- Get and set attribute values.
- Override data manipulation methods.
- Declare custom methods (for example, to traverse associations and to perform custom validation).

The `EntityImpl` class defines the generic behavior of all entity objects. At run time, one entity object is instantiated for each row of data. To customize an entity object, you generate a subclass of the `EntityImpl` class (using the procedure you learned earlier). The subclass extends the base class and inherits methods for inserting, updating, deleting, and locking rows. When you generate the entity object class, you can also choose to include:

- **Accessors:** For generating type-safe accessors that get and set attribute values
- **Data manipulation methods:** For overriding DML methods, such as `lock()` and `doDML()`
- **Create method:** For overriding the `create()` method to modify or add initialization features to the create logic
- **Remove method:** For overriding the `remove()` method to modify or add clean-up code to the remove logic

In the subclass of `EntityImpl`, you can also declare custom methods that perform operations that are not available in `EntityImpl`. For example, you can add a custom method that traverses an association to return an attribute value in a different entity object.

The slide shows an example of a subclass called `EmployeeEOImpl` that is generated for an entity object called `EmployeesEO`.

Commonly Used Methods in `EntityImpl`

Methods You Call	Methods You Override	Methods You Add
`get`*`AttributeName`*`()`	`create()`	`validate`*`Something`*`()`
`set`*`AttributeName`*`()`	`remove()`	
`validate()`	`prepareForDML()`	
`refresh()`	`doDML()`	
`getKey()`	`beforeCommit()`	
`getEntityState()`	`afterCommit()`	
`getPostedAttribute()`	`postChanges()`	
`lock()`		
`validateEntity()`		

See the Java API documentation for the full list.

The slide shows some of the methods that are inherited when you create a subclass of `EntityImpl`, including methods that you commonly call and override. It also shows some of the custom methods you might add to your subclass. The list is not complete.

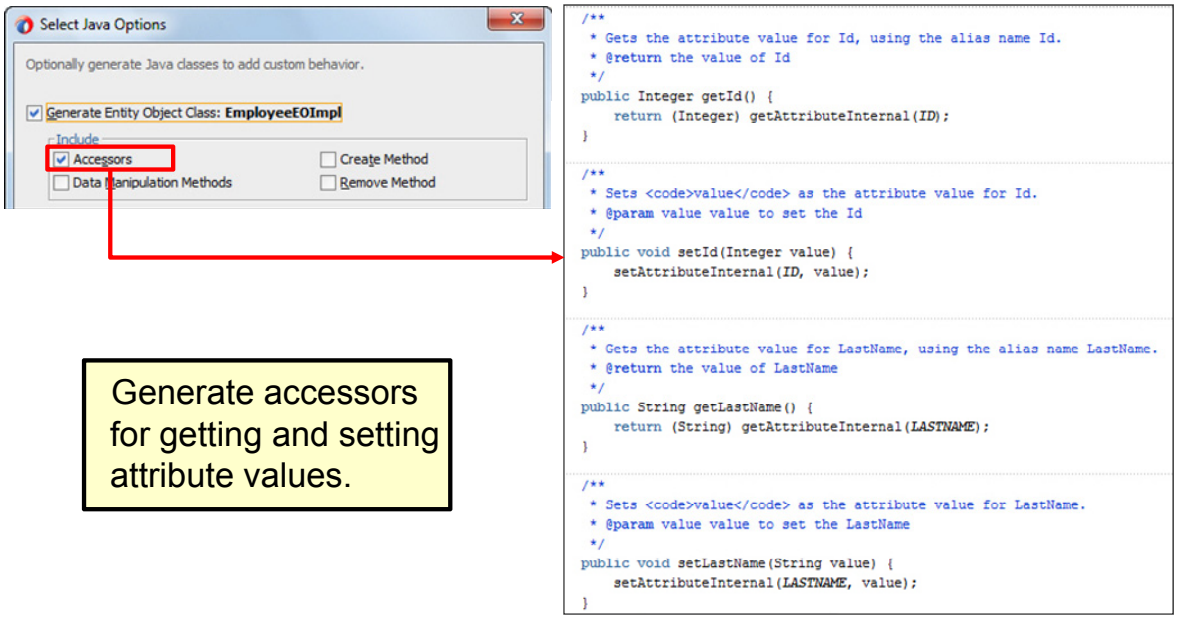
- The **Methods You Call** box lists methods that you call from your code. Notice that these methods enable you to get and set attribute values, perform eager validation, refresh the entity from the database, get the key object for the entity, get the state of the entity (unmodified, initialized, new, and so on), eagerly lock the database row, and validate an entity.
- The **Methods You Override** box lists methods that you can override to modify the default behavior of the entity object. For example, you can override:
  - The `create()` method to supply programmatic default values to specific attributes of a new entity object
  - The `remove()` method to add your own business logic to the remove operation
  - The `prepareForDML()` method to modify attribute values before changes are posted to the database
  - The `doDML()` method to augment or change the standard `INSERT`, `UPDATE`, or `DELETE` DML operation that the framework performs

  - The `beforeCommit()` method to perform complex, SQL-based validation after all entity instances have been posted to the database, but before those changes are committed
  - The `afterCommit()` method to perform custom processing after all entity instances have been committed to the database
  - The `postChanges()` method to ensure that a newly created parent entity gets posted to the database before the child entity on which it depends
- The **Methods You Add** box lists methods that you typically add (for example, custom methods that perform validation).

For a full description of the methods in the `EntityImpl` class, see the *Java API Reference for Oracle ADF Model*:

http://docs.oracle.com/middleware/1212/adf/ADFMR/oracle/jbo/server/EntityImpl.html

Generating Accessors

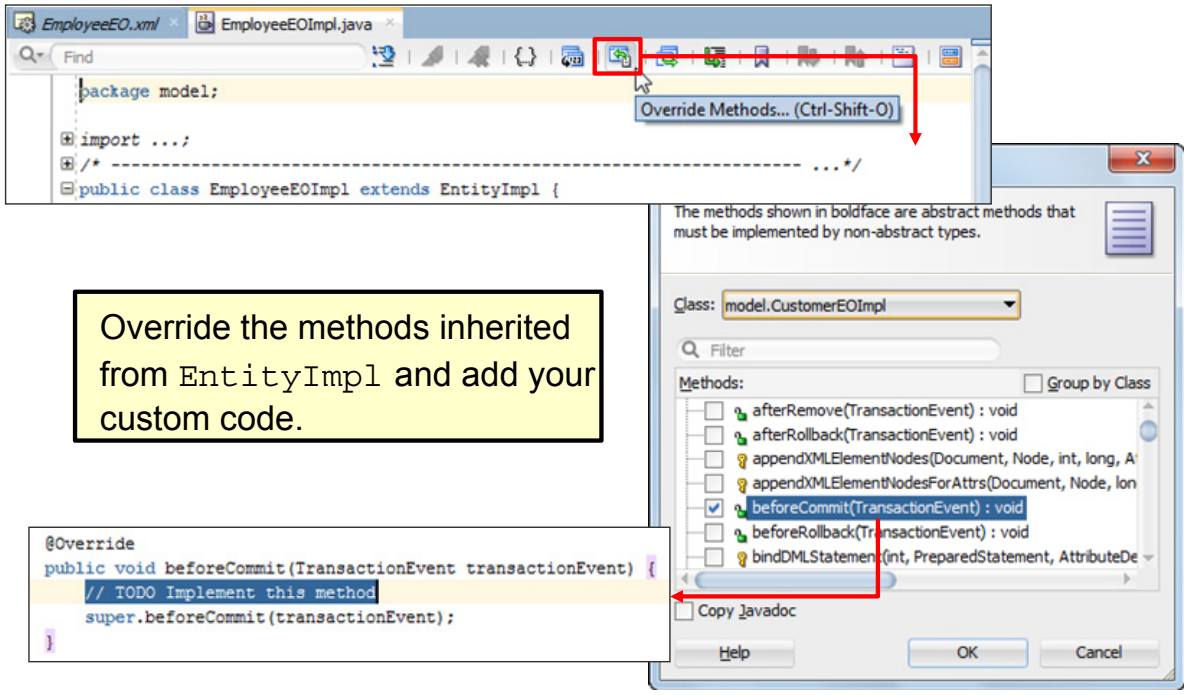


Generate accessors for getting and setting attribute values.

EmployeeEOImpl.java

When you generate the subclass of `EntityImpl`, you can select Generate Accessors to generate methods for getting and setting attribute values.

Overriding Base Class Methods to Modify Behavior



When you generate a subclass of `EntityImpl`, you can choose to generate code for DML methods, the `create()` method, and the `remove()` method. However, you are not limited to overriding those methods. In fact, you can override any method that is inherited by your subclass of `EntityImpl`.

To override methods, open the Java source file (for example, `EmployeeEOImpl.java`) in the editor, and click Override Methods on the editor toolbar (or you can select Source > Override Methods from either the main menu or the context menu). Select the methods that you want to override. You can locate a method by entering its name.

Overriding a method creates skeleton code that performs the default functionality defined for the method. You can then replace or augment the code within the method body, as shown in the example in the next slide.

Example: Overriding `remove()` and `doDML()`

```
// In <Entity name>Impl.java

public void remove() {
  setDeleted("Y");
  super.remove();
}

protected void doDML(int operation,
TransactionEvent e) {
  if (operation == DML_DELETE) {
    operation = DML_UPDATE;
  }
  super.doDML(operation, e);
}
```


Your business requirements might require that rows must never be physically deleted from a table. Instead, the value of a `DELETED` column must be changed from `N` to `Y` when a row is deleted.

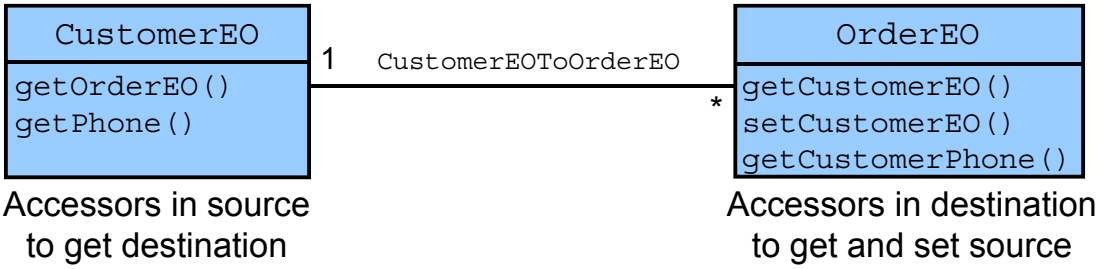
To implement this behavior, you need to override two methods:

- **To update a `DELETED` flag when a row is removed:** Generate a Java class for your entity object and override the `remove()` method to set the deleted flag before calling the `super.remove()` method. The row will still be removed from the row set, but it will have the value of its `DELETED` flag modified to `Y` in the entity cache. It is important to set the attribute *before* calling `super.remove()`, because an attempt to set the attribute of a deleted row causes a `DeadEntityAccessException`.
- **To force an update instead of a delete:** Override the `doDML()` method and write code that conditionally changes the operation flag. When the operation flag equals `DML_DELETE`, your code will change it to `DML_UPDATE` instead.

With this code in place, any attempt to remove an entity row through any view object that is based on that entity will cause an update to the flag in the `DELETED` column instead of physically deleting the row. However, to prevent rows that are marked as deleted from appearing in view object query results, you need to include `DELETED = 'N'` as part of the `WHERE` clause of each view object.

Example: Traversing Associations

- The destination entity's `EntityImpl.java` file contains methods to get and set the source entity. For example, `OrderEOImpl.java` contains `getCustomerEO()` and `setCustomerEO()`.
- You can add a method to `OrderEOImpl.java` to get the phone number of the associated customer:




```
public String getCustomerPhone() {
   return getCustomerEO().getPhone(); }
```


By default, JDeveloper generates accessor methods for each association: a getter and a setter in the destination to retrieve the source, and a getter in the source to get the destination. These accessor methods are generated in the entity object classes on either end of the association.

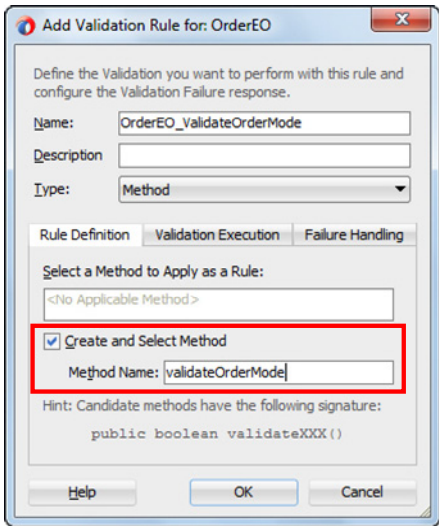
For example, the slide shows two entities, `CustomerEO` and `OrderEO`, that are linked by the `CustomerEOToOrderEO` association. JDeveloper generates two methods in `OrderEOImpl.java` (the destination entity object) for traversing the association from destination to source:

- **`public CustomerEOImpl getCustomerEO()`:** Gets the customer to which the order belongs
- **`public void setCustomerEO(CustomerEOImpl value)`:** Sets the value of the customer to which the order belongs

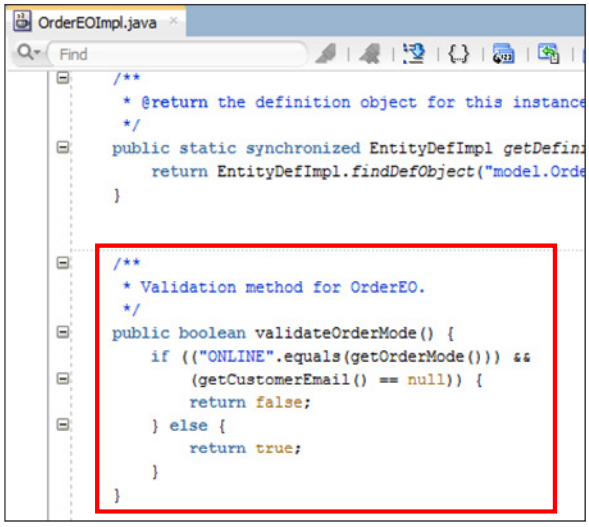
To retrieve an attribute's value from the source entity while in the destination entity, use the `get<Source>()` method plus the `get()` method for the attribute that you want to retrieve, as shown in the example in the slide for retrieving the phone number from the `CustomerEO` entity. The `CustomerEO` entity has only the `getOrderEO()` accessor, with no setter.

Creating a Method Validator for an Entity Object or Attribute

Create a validation rule that calls a custom Java method:



Define a method validator in the entity object.



Add your own code to the `validate`*`XXX`*`()` method.

ORACLE

ADF provides a rich declarative framework for validation. However, you can write complex validation rules for your business layer by developing a custom validation rule. You code the rule in Java, and it is triggered by the method validator at the appropriate point in the entity object validation cycle.

You can create entity-level or attribute-level method validators. You can add any number of validators, provided that each validator triggers a distinct method name in your code. There are many types of validation you can code with a method validator, either on an attribute or on an entity.

The method must be defined as public and return a Boolean value. The name of the method must begin with the keyword `validate`. The code can be as complex or simple as the validation rules require.

To create a method validator for an entity:

1. Either for the entity object or for one of its attributes, invoke the Add Validation Rule editor (as you do for any type of validation rule).
2. Select Method from the Rule Type list.

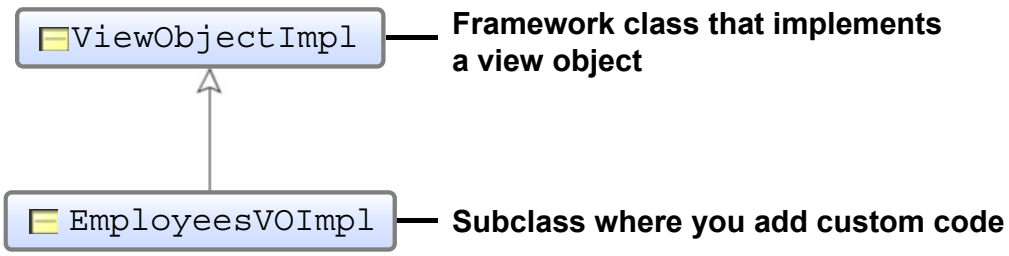
3. Select an existing method or, to create a new one, select the “Create and Select Method” check box and enter a name for the method. Remember that your method name must begin with the word `validate`.
4. Click OK to add the method to your entity object’s Java class. If the class does not exist, you are prompted to create it.
5. In the `<EO>Impl.java` file, edit the new method that you added to define your custom validation logic.

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule.

Note: For more information about setting up the validation rule (such as specifying conditions for executing the rule and setting the error message), see the lesson titled “Validating User Input”.

Customizing View Objects Programmatically

Generate a subclass of `ViewObjectImpl`:



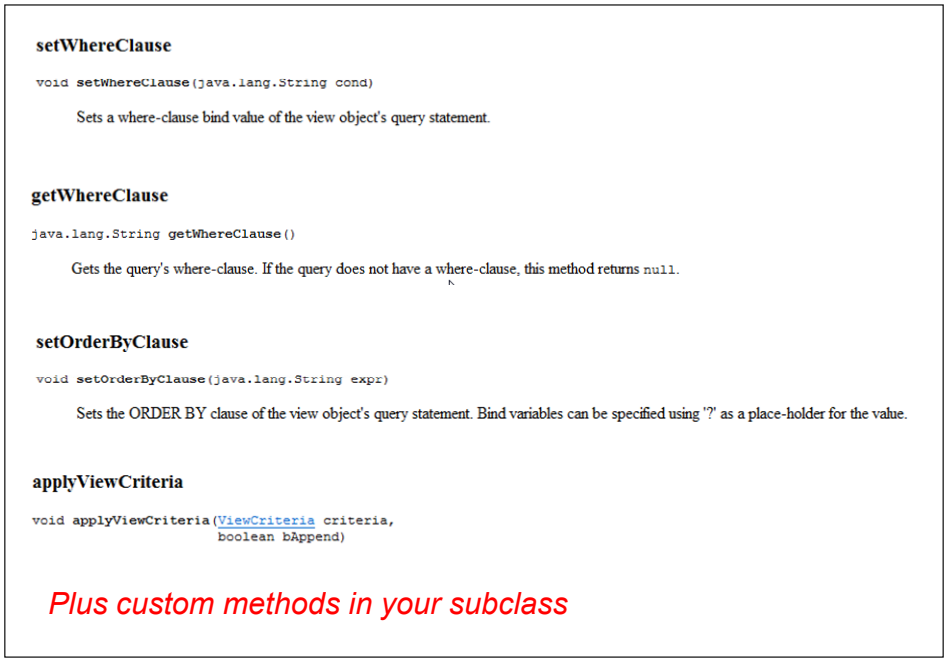
Common use cases:

- Change the `WHERE` clause and re-execute the query.
- Set a bind variable and re-execute the query.
- Set and manage view criteria.

To customize a view object, you use the steps that you learned earlier to generate a subclass of the `ViewObjectImpl` class. The `ViewObjectImpl` class defines the generic behavior of view objects. All view object classes extend this class. The ADF Business Components technology uses this class to manage instances of each view. When you generate the view object class, you can also generate bind variable accessors (type-safe accessors for the view object's named bind variables) and custom Java data source methods.

The `ViewObjectImpl` class also implements some listener interfaces and contains many methods of its own, so there are several opportunities for programmatic manipulation.

Commonly Used Methods in `ViewObjectImpl`



setWhereClause

`void setWhereClause(java.lang.String cond)`

Sets a where-clause bind value of the view object's query statement.

getWhereClause

`java.lang.String getWhereClause()`

Gets the query's where-clause. If the query does not have a where-clause, this method returns `null`.

setOrderByClause

`void setOrderByClause(java.lang.String expr)`

Sets the ORDER BY clause of the view object's query statement. Bind variables can be specified using '?' as a place-holder for the value.

applyViewCriteria

`void applyViewCriteria(ViewCriteria criteria, boolean bAppend)`

Plus custom methods in your subclass

See the Java API documentation for the full list.

`ViewObjectImpl` is the representation of the behavior of a view object instance (the query, the `WHERE` clause, the order, and so on), whereas `ViewRowImpl` is the representation of an instance of a view object row. As you can see, it provides methods for getting and setting the `WHERE` clause, setting the `ORDER BY` clause, and applying view criteria.

Note: For a full description of the methods that you can call on the `ViewObjectImpl` class, see *Java API Reference for Oracle ADF Model*:

http://docs.oracle.com/middleware/1212/adf/ADFMR/oracle/jbo/server/ViewObjectImpl.html

Example: Setting WHERE Clauses Programmatically

Set the WHERE clause by calling the setWhereClause() method in a subclass of ViewObjectImpl:

```
public void onlyAdministration(){
    final Integer DEPTID = 50;
    setWhereClause("DEPT_ID = " + DEPTID);
    executeQuery();
}
```

Method defined in EmployeesVOImpl

Adds the clause:
WHERE DEPT_ID = 50

```
SELECT EmployeeEO.ID,
       EmployeeEO.LAST_NAME,
       EmployeeEO.FIRST_NAME,
       EmployeeEO.USERID,
       EmployeeEO.EMAIL,
       EmployeeEO.START_DATE,
       EmployeeEO.COMMENTS,
       EmployeeEO.MANAGER_ID,
       EmployeeEO.TITLE_ID,
       EmployeeEO.DEPT_ID,
       EmployeeEO.SALARY,
       EmployeeEO.COMMISSION_PCT
FROM  S_EMP EmployeeEO
```

EmployeesVO Query

ORACLE

You can set the WHERE clause of a view object query by calling the setWhereClause() method. The example in the slide shows a custom method that is defined in EmployeesVOImpl.java (a subclass of ViewObjectImpl). The method calls the setWhereClause() method on the current view object to set the query's WHERE clause.

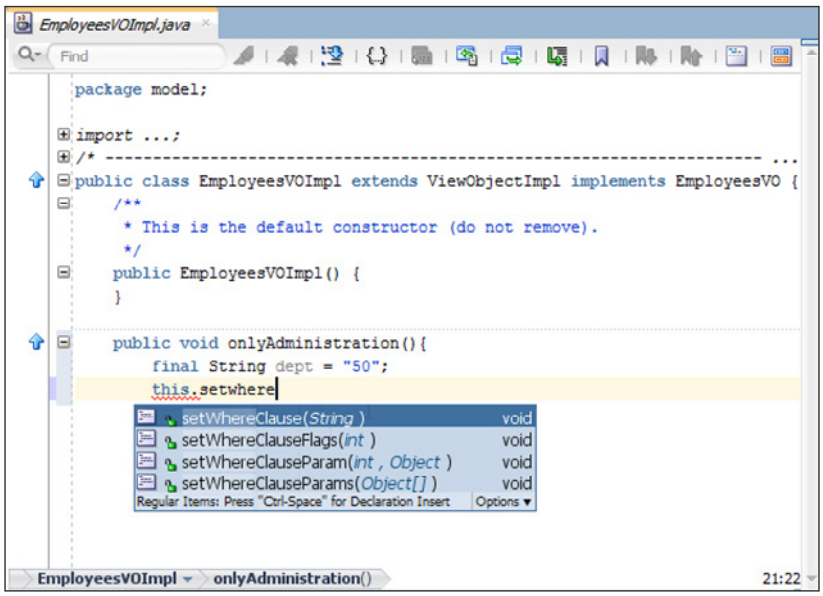
The WHERE clause in the example selects all employees who belong to department 50, the Administration department. Notice that the method also calls the executeQuery() method to re-execute the query with the WHERE clause set.

Of course, you are not required to put this logic in a custom method. However, encapsulating logic in a custom method is a good way to create reusable code. Later, you learn how to expose custom methods in the view object's client interface so that you can call custom methods from the view layer.

Note that hard-coding the criteria (as shown in the example) can result in a fragmented shared SQL area.

Using Code Insight

Use code insight to see the methods that you can call from within a class:



Code insight is built into the JDeveloper code editor. For example, after typing an object reference followed by a period, pause for a moment and JDeveloper then displays a list of available methods. (Or you can press Ctrl + Space at any time to access code insight.) If you begin typing a method name (such as `set` or `get`), code insight will narrow the list of available methods based on what you have typed.

Code insight is a quick way of seeing the methods that are available, and it enables you to add code quickly.

Example: Setting the Value of Named Variables at Run Time

Set the value of a named variable by calling the `setNamedWhereClauseParam()` method:

```
public void filterByDept(Number dept){
    this.setNamedWhereClauseParam("pDeptId", dept);
    this.executeQuery();
}
```

Method defined in `EmployeesVOImpl`

A better approach:
Generate bind variable accessors when you generate the view object class.

Sets the `pDeptId` bind variable

Select:

```
SELECT EmployeeEO.ID,
       EmployeeEO.LAST_NAME,
       EmployeeEO.FIRST_NAME,
       EmployeeEO.USERID,
       EmployeeEO.EMAIL,
       EmployeeEO.START_DATE,
       EmployeeEO.COMMENTS,
       EmployeeEO.MANAGER_ID,
       EmployeeEO.TITLE_ID,
       EmployeeEO.DEPT_ID,
       EmployeeEO.SALARY,
       EmployeeEO.COMMISSION_PCT
FROM  S_EMP EmployeeEO
```

Where:

```
DEPT_ID = :pDeptId
```

EmployeesVO Query

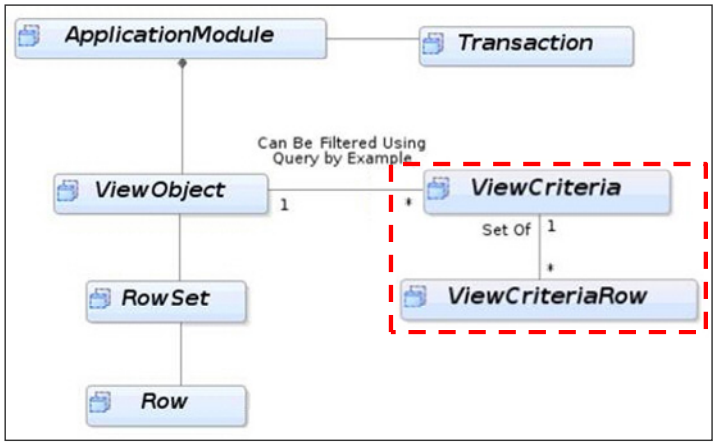
The query for a view object might include named bind variables whose value you want to assign at run time. You can use the `setNamedWhereClauseParam()` method to assign a value to any named bind variable, as shown in the example in the slide.

The example in the slide shows a custom method that is defined in `EmployeesVOImpl.java` (a subclass of `ViewObjectImpl`). The method calls the `setNamedWhereClauseParam()` method on the current view object and passes in the value of the department ID to set the value of the bind variable `pDeptId`. The method also calls the `executeQuery()` method to execute the query using the bind variable value passed in at run time.

The slide shows how to call the `setNamedWhereClauseParam()` method to set the value of bind variables. However, a better approach is to select the “Include bind variable accessors” check box when you generate the view object implementation class. Choosing this option generates type-safe getters and setters for the bind variables.

Remember: You can expose custom methods in the view object's client interface so that you can call custom methods from the view layer. However, to avoid a potential problem with SQL injection, make sure that you safeguard your application against malicious users by restricting a user's ability to pass values directly to the database.

Using View Criteria with View Objects



Interfaces for working with view criteria

Earlier you learned about the framework classes that implement ADF Business Components. In addition to the classes that you saw earlier, the framework includes interfaces for working with view criteria: `ViewCriteria` and `ViewCriteriaRow`. (As you probably remember, a view criteria is a filter that you define at design time and can apply to a view object instance at run time. The view criteria augments the view object's `WHERE` clause.)

Each view object can have one or more view criteria objects, each of which is made up of one or more view criteria rows that represent `AND`ed or `OR`ed elements.

The `ViewCriteria` and `ViewCriteriaRow` interfaces provide methods for creating and populating criteria rows to support query-by-example. However, this lesson does not describe how to create view criteria programmatically. Instead, it focuses on a more common use case: applying a predefined view criteria to a view object programmatically.

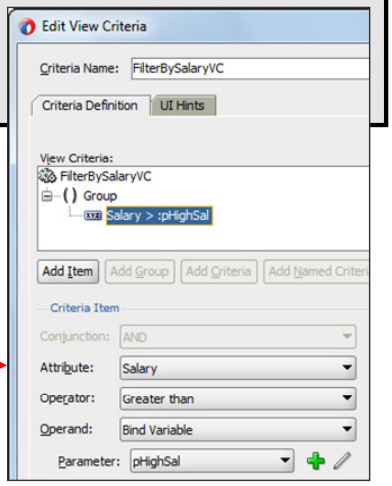
Example: Applying View Criteria Programmatically

Apply the view criteria by calling the `applyViewCriteria()` method:

```
public void applyFilterBySalaryVC(Number sal){
    ViewCritera vc = getViewCriteria("FilterBySalaryVC");
    this.setNamedWhereClauseParam("pHighSal", sal);
    this.applyViewCriteria(vc);
    this.executeQuery();
}
```

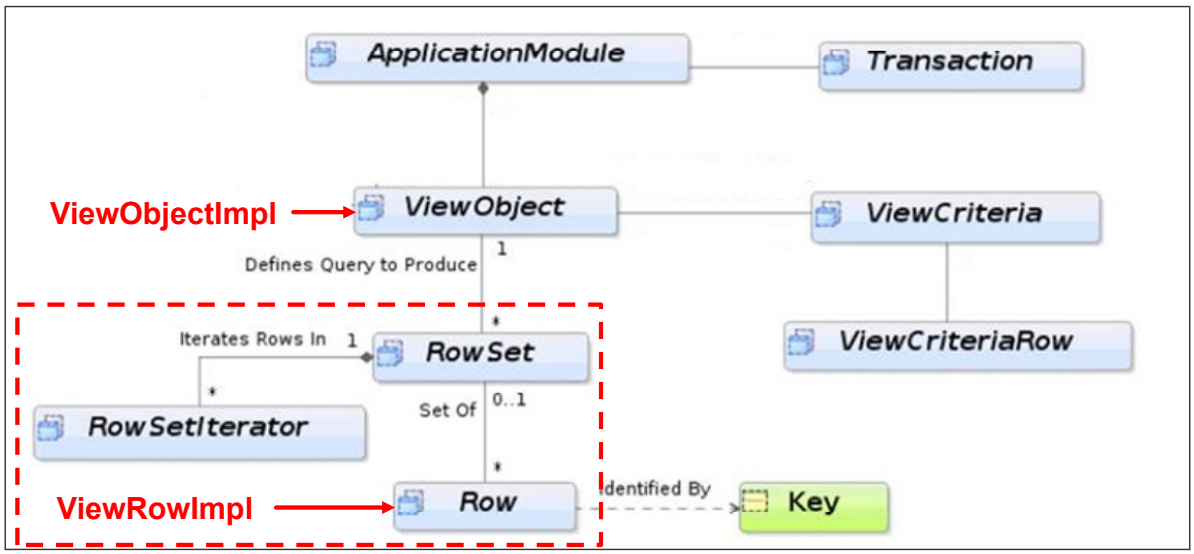
Method defined in `EmployeesVOImpl`

EmployeesVO defines a view criteria that selects all employees with a salary that is greater than `pHighSal`.



The example in the slide shows a custom method that is defined in `EmployeesVOImpl.java` (a subclass of `ViewObjectImpl`). The method calls the `setNamedWhereClauseParam()` method on the current view object and passes in a salary value to set the bind variable `pHighSal`. The method then calls `applyViewCriteria()` to apply the view criteria to the current object, and then it calls `executeQuery()` to execute the query with the view criteria applied.

Working with View Rows



Classes that implement ADF BC

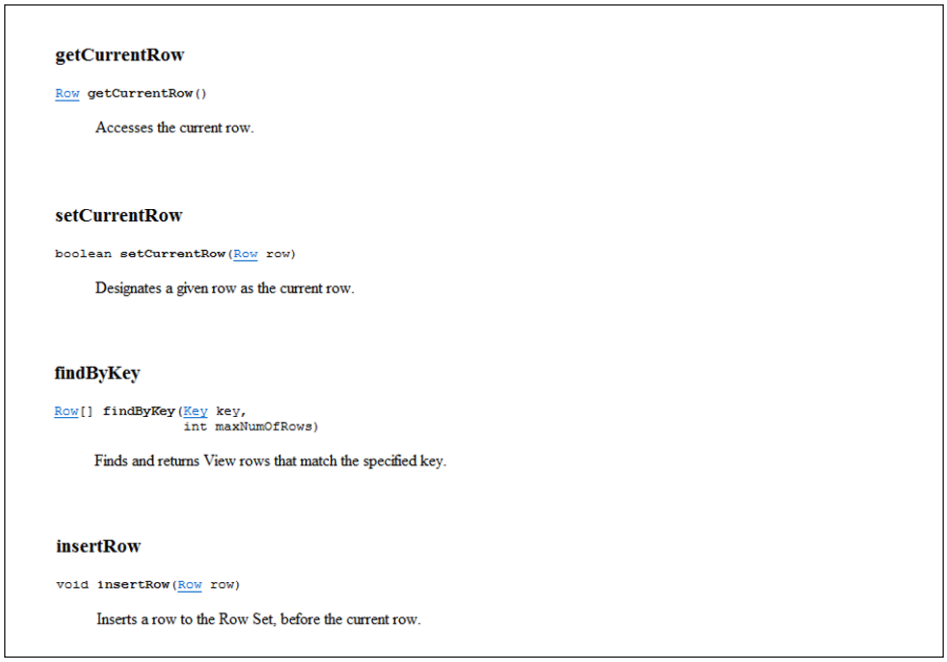
Earlier you learned that the `ViewObjectImpl` class implements the view object. The `ViewObjectImpl` class defines everything about the view object, including the query, bind variables, and view criteria that are applied. You also learned that you can use methods available in the `ViewObjectImpl` class to set the `WHERE` clause, set bind variables, manage view criteria, and execute the query.

The `ViewObjectImpl` class also manages the rowsets that are returned by the query. Each view object uses rowsets to manage separate collections of rows. A row represents a single element that has been queried and is often a pointer to an entity instance. For example, a view object might return 20 rows in a rowset. The rowset would include 20 instances of row objects. Each row object points back to the entity object, which actually holds the data.

`ViewRowImpl` is the framework class that represents a single row of a view object. The `ViewRowImpl` class points to an `EntityImpl` class where the data is stored.

The `ViewObjectImpl` class is able to manage rowsets and iterate over collections of rows because it implements the `RowSet` and `RowSetIterator` interfaces. The methods defined in `RowSet` and `RowSetIterator` enable you to manage a collection of view rows and iterate over the collection.

Commonly Used Methods in `RowIterator` and `RowSet` (or `RowSetImpl`)



getCurrentRow

`Row getCurrentRow()`

Accesses the current row.

setCurrentRow

`boolean setCurrentRow(Row row)`

Designates a given row as the current row.

findByKey

```
Row[] findByKey(Key key,
                int maxNumOfRows)
```

Finds and returns View rows that match the specified key.

insertRow

`void insertRow(Row row)`

Inserts a row to the Row Set, before the current row.

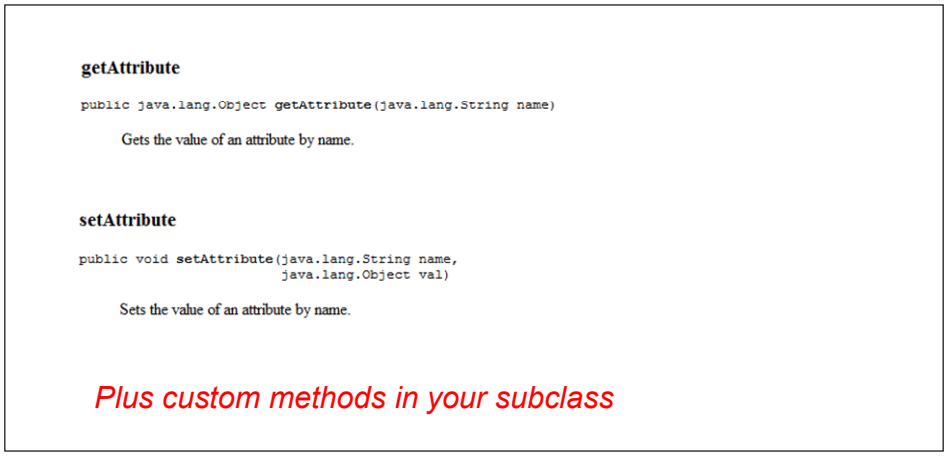
See the Java API documentation for the full list.

A rowset manages the various rows that are returned from a query. Therefore, the classes and interfaces for working with rowsets provide methods for getting and setting the current row, finding a row, inserting a row, and so on. Because of the inheritance structure of these classes, you might need to look at more than one class in the *Java API Reference* to find information about the methods for working with rowsets.

Note: For a full description of these and other methods for working with rowsets, see the *Java API Reference for Oracle ADF Model*:

http://docs.oracle.com/middleware/1212/adf/ADFMR/index.html

Commonly Used Methods in `ViewRowImpl` and `Row`



getAttribute

```
public java.lang.Object getAttribute(java.lang.String name)
```

Gets the value of an attribute by name.

setAttribute

```
public void setAttribute(java.lang.String name,
                         java.lang.Object val)
```

Sets the value of an attribute by name.

Plus custom methods in your subclass

See the Java API documentation for the full list.

`ViewRowImpl` implements the actual row in the rowset and therefore provides methods that you use to get and set an attribute value in the row. You can create a subclass of `ViewRowImpl` and add a custom method.

Example: Finding a Row and Setting It as the Current Row

Find the row by calling the `findByKey()` method. Then set the row as current by calling the `setCurrentRow()` method:

```
public void gotoADepartment(oracle.jbo.domain.Number deptno){
    Key deptKey = new Key(new Object[] {deptno});
    Row[] r = findByKey(deptKey, 1);
    setCurrentRow(r[0]);
}
```

Method defined in `DepartmentsVOImpl`

Note: Remember to check for nulls and handle exceptions (not shown here).

ORACLE

The example in the slide shows the code for a custom method that looks at all the rows in a rowset and, using a key, finds a row and then sets it to the current row. The custom method is defined in a subclass of `ViewObjectImpl` called `DepartmentsVOImpl`. (The steps for generating a subclass are described earlier in this lesson.)

The code within the method body:

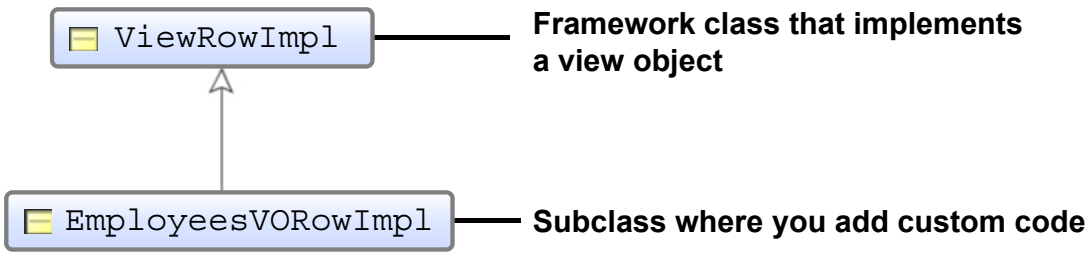
1. Creates a key that will be used to find the row. The department number is wrapped in a key object so that it can be passed into the `findByKey()` method.
2. Calls the `findByKey()` method to find the row. This method takes a key followed by an integer (`1` in the example) to find the first key that matches the department number. This method returns a reference to a Row array.
3. Sets the current row to the found row (that is, the first row in the array of rows returned by the `findByKey()` method). In this example, there is only one row that matches. (In Java, the first element in an array has an index of `0`.)

The `findByKey()` and `setCurrentRow()` methods can be called from `DepartmentsVOImpl` because `ViewObjectImpl` implements the `RowIterator` interface.

To make the code easier to read, the example does not show how to check for null values and handle exceptions, but in a real application, you do need to implement exception handling.

Customizing View Object Rows by Subclassing `ViewRowImpl`

Generate a subclass of `ViewRowImpl`:



Common use cases:

- Get and set row-level attribute values.
- Override the `isAttributeUpdateable()` method.
- Declare custom methods (for example, to iterate over detail rows and update an attribute).

ORACLE

To customize a view object row, use the steps that you learned earlier to generate a subclass of the `ViewRowImpl` class. As you learned, the `ViewRowImpl` class represents a single row of a view object. When you generate this class, you can choose to generate type-safe accessors for the view row's attributes. You can also choose to expose the accessors to the client, thereby turning all accessors into client methods.

Some common uses of the `ViewRowImpl` class are the following:

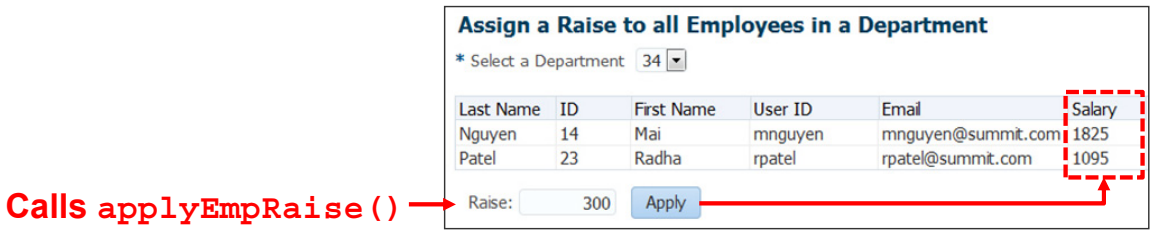
- Getting and setting row-level attribute values
- Overriding the `isAttributeUpdateable()` method to determine the updateability of an attribute in a conditional way
- Declaring custom methods to perform actions such as:
  - Iterating over detail rows in a master-detail relationship to update an attribute
  - Calculating the value of a view object-level transient attribute
  - Performing custom processing to set view row attributes

Example: Iterating through Detail Rows to Update an Attribute

Get the row iterator of the detail view object, and then iterate over the rows to update an attribute:

```
public void applyEmpRaise(BigDecimal raise){
    RowIterator rows = getEmployeesVO();
    while (rows.hasNext()) {
        EmployeesVORowImpl currentrow = (EmployeesVORowImpl)rows.next();
        currentrow.setSalary(currentrow.getSalary().add(raise));
    }
}
```

Method defined in `DepartmentsVORowImpl`



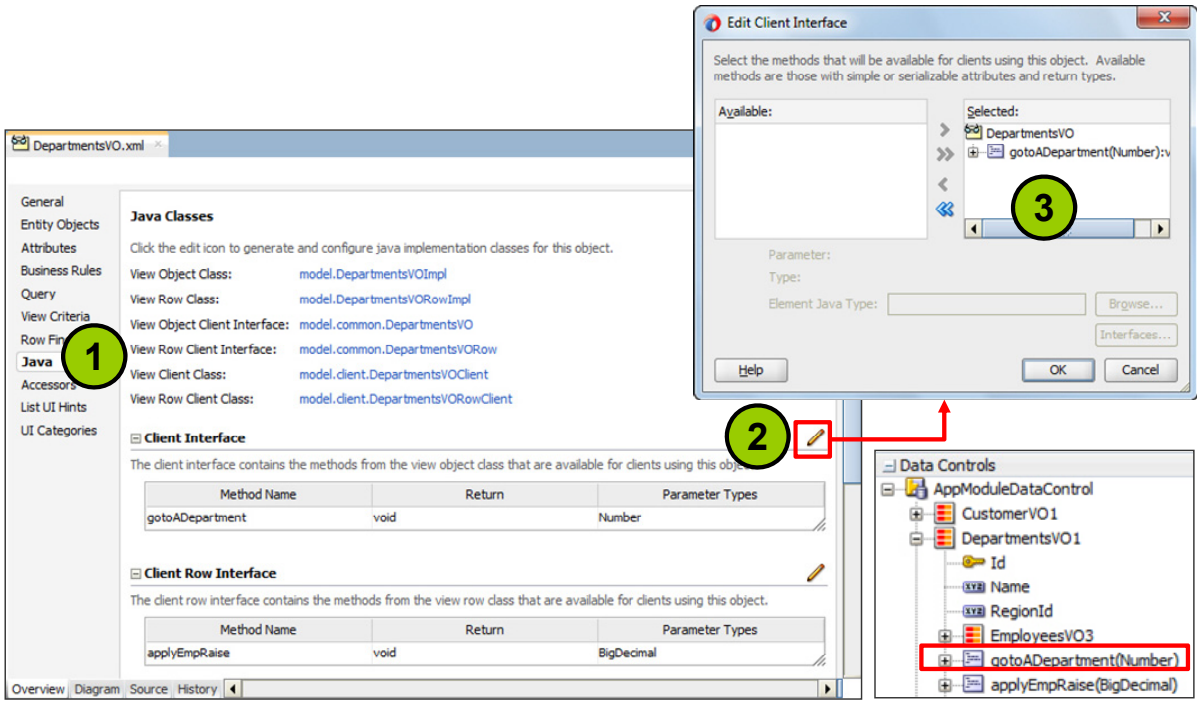
The example in the slide shows a custom method that applies a raise to all the employees in a selected department. The custom method is defined in a subclass of `ViewRowImpl` called `DepartmentsVORowImpl`. (The steps for generating a subclass are described earlier in this lesson.)

The code within the method body:

1. Calls the `getEmployeesVO()` accessor to traverse the master-detail link and get the associated row iterator for `EmployeesVO`
2. Loops through every employee view row (while the iterator has records) and calls the `setSalary()` method in `EmployeesVORowImpl` to apply the raise to the current salary. Note that before calling the `setSalary()` method, you must first generate `EmployeesVORowImpl`.

The slide shows how to call the custom method from the view layer by exposing the method in the client row interface and creating an ADF parameter form that calls the method. In the next slide, you learn how to expose custom methods in the client interface.

Exposing Custom Methods in the View Object and View Object Row Client Interfaces



When you add a public custom method to one of your implementation classes (for example, to `DepartmentsVOImpl` and `DepartmentsVORowImpl`), you need to expose the method in the client interface if you want clients to be able to invoke the method. To do this, perform the following steps:

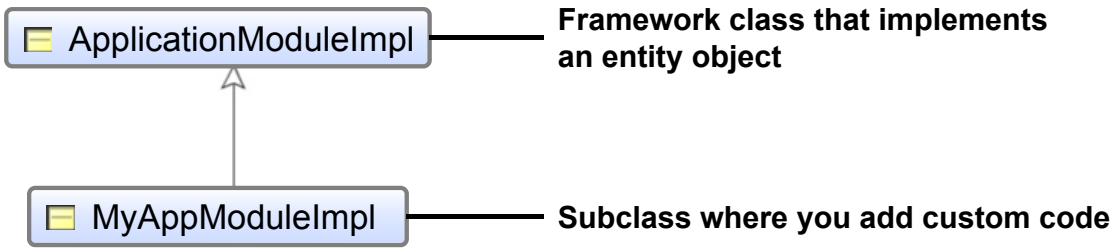
1. In the view object editor, click the Java tab.
2. Click the Edit (pencil) icon in the Client Interface or Client Row Interface section.
3. Shuttle the method into the Selected pane, and then click OK.

JDeveloper creates the client interfaces and classes that are required to expose the method. After you expose methods in the client interface, the methods appear in the Data Controls panel as operations that you can use to build data-bound UI components, such as buttons and ADF parameter forms.

A common use case for exposing custom methods in a client interface is when the built-in operations (such as `Create` and `Delete`) do not meet your business requirements. For example, rather than simply deleting a department, you are more likely to need a `closeDepartment` function, which might involve a number of actions, such as transferring employees and closing an office, rather than simply deleting the specified department record as you would with the default delete behavior provided by ADF Business Components. You can implement this functionality in a custom method and expose it in a client interface.

Customizing Application Modules Programmatically

Generate a subclass of `ApplicationModuleImpl`:



Common use cases:

- Perform common application module operations.
- Override methods to add custom setup code.
- Declare custom service methods.

The `ApplicationModuleImpl` class is the base class for application module components. The application module is the ADF component that is used to implement a business service, so you can think of the application module class as the place where you write your service-level application logic. The application module coordinates with view object instances to support updatable collections of value objects that are automatically "wired" to business domain objects. The business domain objects are implemented as ADF entity objects.

You can use the `ApplicationModuleImpl` class to:

- Perform common application module operations, such as finding and creating view objects, accessing nested application module instances, and accessing the current transaction object
- Override methods to add custom setup code, such as setup code that is specific to a given view object instance in the application module
- Create custom methods or expose the methods in the client interface

Creating Custom Service Methods

Service methods:

- Are useful for:
  - Code that is not specific to a particular view object
  - Performing operations across view object instances
  - Managing transactions
  - Dynamically changing the data model
- Can be called from the client, requiring very little data manipulation in the client itself
- Are implemented in the application module's class
- Are added by:
  - Adding code to the Java class
  - Exposing the custom service methods in the client interface

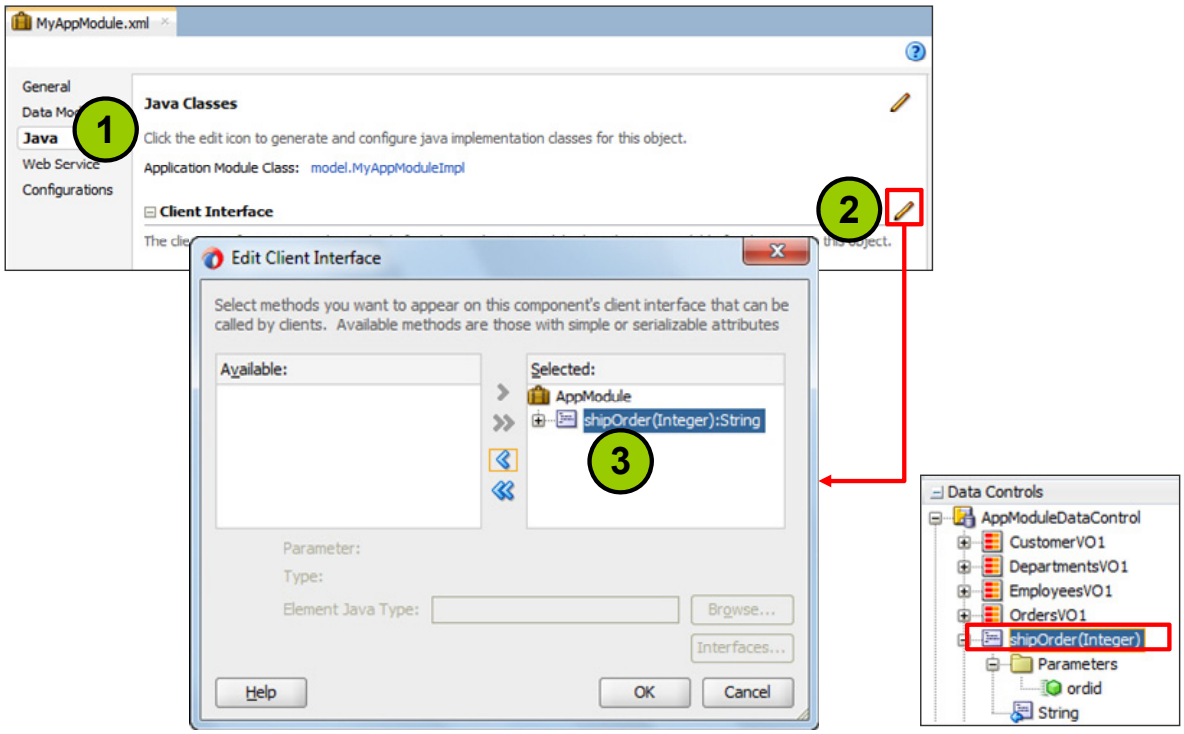
Service methods are methods that are specific to an application module. These methods may be independent of any view in the application.

You can also add code that affects the behavior of a particular view object in the application module; for example, you can change the `WHERE` or `ORDER BY` clause of a view object. You would do this if you were reusing a view object in multiple application modules and wanted the view object to behave differently in different application modules.

Example: Deciding When to Add Custom Code to an Application Module

Suppose that two application modules, OrderManagementApp and CustomersApp, both have CustomersVO in their data model. CustomersApp requires code to perform analysis of customers' purchasing history; OrderManagementApp does not need this information. Adding this code to CustomersVO would add size and complexity to the view object for the sake of functionality that will never be used by OrderManagementApp. Therefore, you should add your custom code to the CustomersApp application module.

Exposing Custom Service Methods in the Client Interface



If you want clients to be able to invoke your custom service method, you need to include the method in the application module's client interface. To do this, perform the following steps:

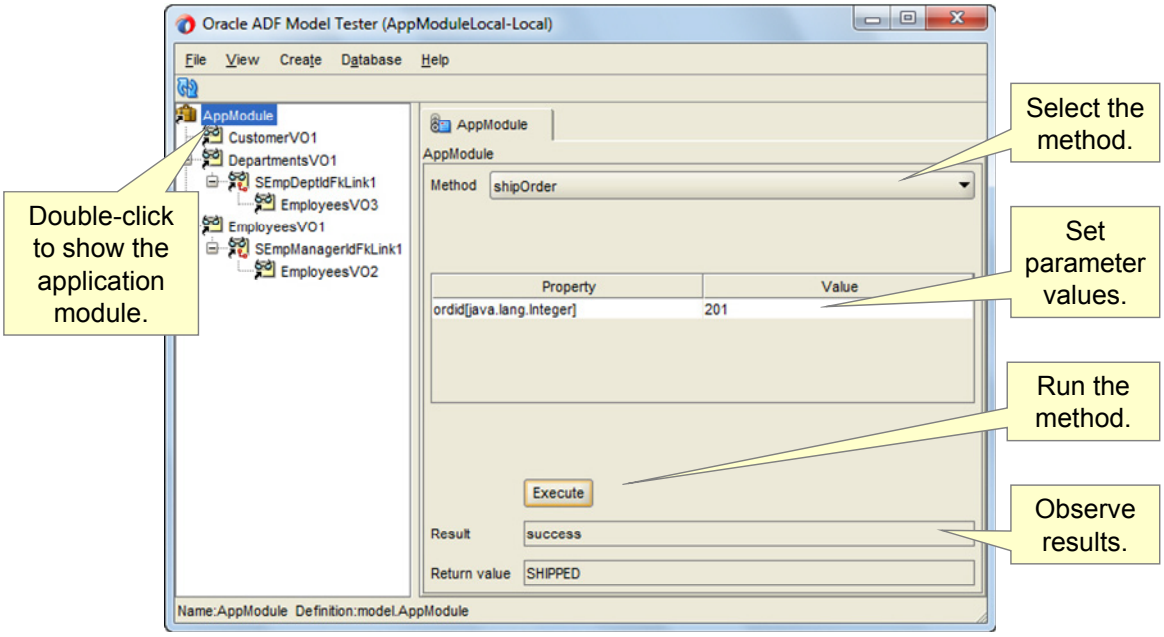
1. In the application module editor, click the Java tab.
2. Click the Edit (pencil) icon in the Client Interface section.
3. Shuttle the method to the Selected pane, and then click OK.

JDeveloper creates a Java interface with the same name as the application module in the `common` subpackage of the package where your application module resides. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all the functionality that your application module inherits from the `ApplicationModuleImpl` class. Each time a method is added to or removed from the Selected list in the Edit Client Interface dialog box, the corresponding service interface page is updated automatically.

JDeveloper also generates a client proxy class that is used when you deploy your application module for access by a remote client. All the generated files are displayed in the Applications window under the application module's node.

Note: If your method does not appear in the list of available methods in the Edit Client Interface dialog box, the method might not conform to one or more of the rules for inclusion. Only public methods with parameter and return types that are primitives or that implement the `Serializable` interface are included in the list.

Testing the Client Interface in the Oracle ADF Model Tester



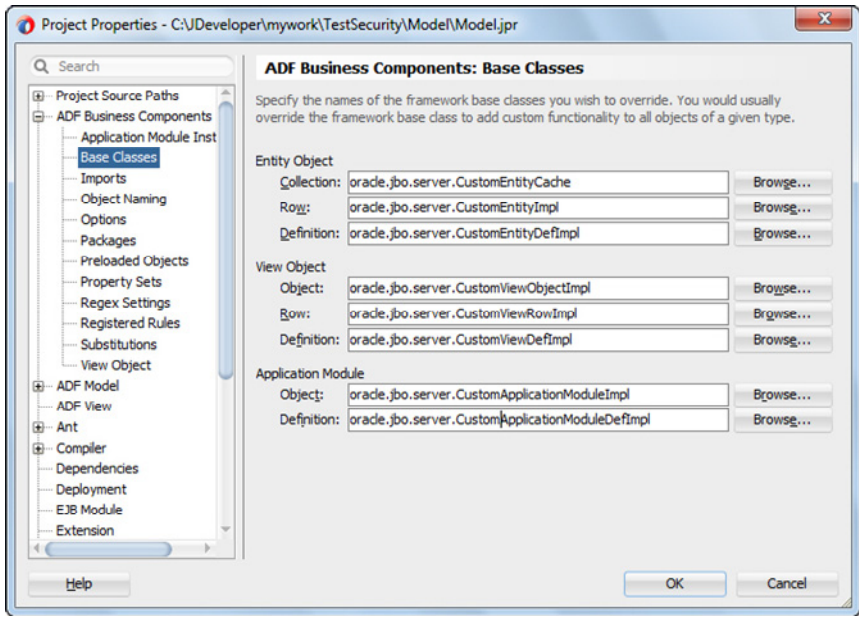
ADF Model Tester

You can use the Oracle ADF Model Tester to test service methods. Double-click the application module in the tree at the left. From the Method list, select the method that you want to test, and supply values for any required parameters. When you click Execute, the results (including a return value, if any) are displayed in the lower part of the window.

Keep in mind that you can use the ADF Model Tester to test any public method that you expose as a client interface, including custom methods that you define in the view object class, the view object row class, or the application module class.

Creating Extension Classes for ADF Business Components

Create framework extension classes and configure JDeveloper to use them:



Before you begin to develop application-specific business components, you can create a layer of classes that extend all of the ADF Business Components framework base classes described in this lesson. When you create your implementation classes, they will inherit from your classes instead of the classes provided by Oracle. With this approach, you have more control over the behavior that is inherited by your implementation classes. Even if you do not expect to change the default behavior of the Oracle classes, creating the framework extension classes at the beginning of your project gives you more flexibility over changing the default behavior at a later time. Any changes that you make to your framework extension classes will be inherited by all the classes that extend your base classes.

After creating the framework extension classes, you configure your model project to use them. Go to the project properties for your model project (double-click the project in the Applications window). Under ADF Business Components > Base Classes, specify your extension class names. These settings configure JDeveloper to use your extension classes whenever you generate an implementation class for an ADF business component.

For example, suppose that you want to add logging code that logs all occurrences of a particular framework action. You can add this logic to your extension class, `CustomEntityImpl`, and every entity object that you create inherits from this class and automatically implements the logging feature.

The following list of classes represents a common set of customized framework base classes that you might create in a package name of your own choosing, such as `com.yourcompany.adfextensions`. Each class imports the `oracle.jbo.server.*` package.

- public class `CustomEntityImpl` extends `EntityImpl`
- public class `CustomEntityDefImpl` extends `EntityDefImpl`
- public class `CustomViewObjectImpl` extends `ViewObjectImpl`
- public class `CustomViewRowImpl` extends `ViewRowImpl`
- public class `CustomApplicationModuleImpl` extends `ApplicationModuleImpl`
- public class `CustomDBTransactionImpl` extends `DBTransactionImpl2`
- public class `CustomDatabaseTransactionFactory` extends `DatabaseTransactionFactory`

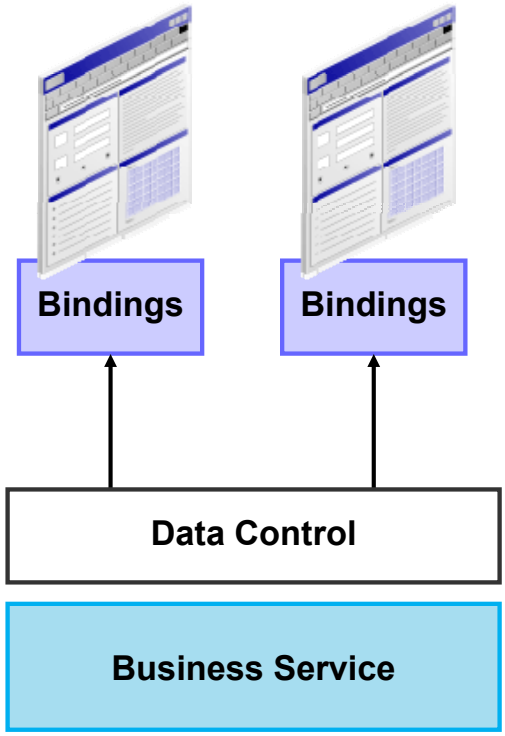
Note: Explaining how to create framework extension classes is beyond the scope of this course. For more information about creating these classes, see *Developing Fusion Web Applications with Oracle Application Development Framework:*

http://docs.oracle.com/middleware/1212/adf/ADFFD/bcadvgen.htm

Accessing Data and Operations Through the Model Layer

Access business services through the model layer, which consists of:

- Data control
  - Abstracts the business service
  - Collections, attributes, methods, and so on
- Bindings
  - Bind to a UI component
  - List, attribute, tree, and so on



ORACLE

You can also write code that uses ADF binding classes to access data and operations that are defined in the business services layer.

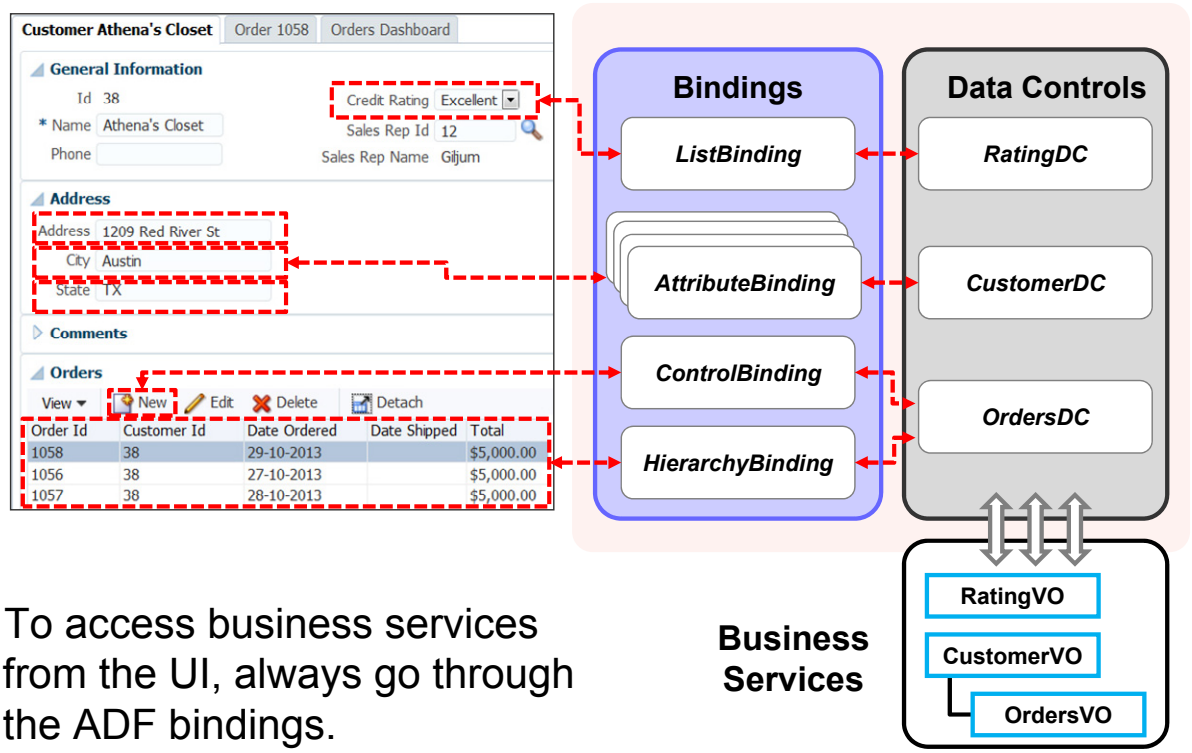
In previous lessons, you learned how to bind business methods to UI controls (such as buttons) by using drag and drop. However, there may be situations when you need to have more programmatic control over when and how methods are called. For example, rather than calling a single method when a user clicks a button, you might need to add programmatic logic to determine which methods to call and then call the methods in the correct sequence. In addition, all of this logic must be called from a single user action, such as clicking a button.

To implement this, you code against the model layer. In fact, whenever you want end users to access data or operations defined in the business services layer, you code against the model layer. Remember that the model layer consists of the following:

- **Data control:** Provides a public interface for the business service. The data control exposes things like data collections, attributes, methods, and operations that are available for binding.
- **Bindings:** Enable you to connect UI components to data controls. There are different types of bindings for different kinds of UI components (such as list bindings, attribute bindings, tree bindings, and so on).

The binding calls the data control, which calls the action in the business services layer. When you create pages in JDeveloper by dragging a data control element to a page, the bindings are created for you. For example, when you drag a collection to the page and create a list, JDeveloper generates a list binding. Similarly, text fields are backed by attribute bindings, tree controls are backed by hierarchy bindings, and so on (you learn about the different types of bindings next).

ADF Binding Types



To access business services from the UI, always go through the ADF bindings.

ORACLE

The easiest way to learn how the different types of bindings work is to see how UI components on a page map to ADF bindings in the model layer. In the slide, you see the following binding types and the UI controls that they back:

- **List binding:** Binds the items in a list (such as a selection list) to all values of an attribute in a data collection
- **Attribute binding:** Binds a single value (such as an input text field) to a specific attribute in an object
- **Control binding:** Binds a command control (such as a button or link) to an operation that is exposed in the data control
- **Hierarchy binding:** Binds an collection of data (such as rows and columns in a table) to a data collection

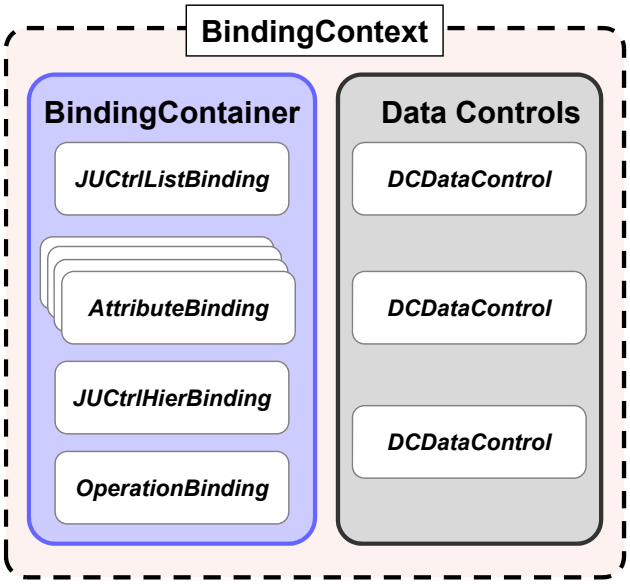
For example, the control that displays the credit rating selection list is backed by a list binding that accesses the RatingVO view object through the RatingDC data control.

The slide shows only a subset of the bindings that exist for the page in the example. Many other bindings exist, including attribute bindings for the values that are displayed in the table.

Note: For a full description ADF binding types, see *Developing Fusion Web Applications with Oracle Application Development:*

http://docs.oracle.com/middleware/1212/adf/ADFFD/bcdcpal.htm

Java Classes Behind the ADF Bindings



As you can see, there is a Java class that backs every declarative feature that you saw in the previous slide. Some of the common classes and interfaces that you work with include:

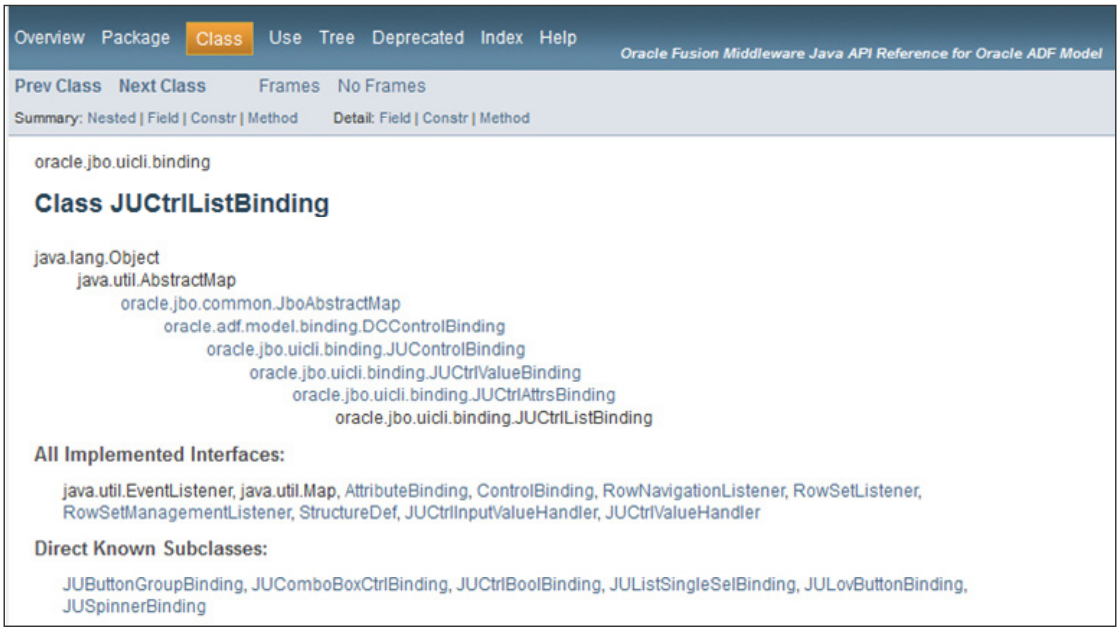
- **`JUCtrlListBinding`:** The class responsible for displaying a list of values
- **`AttributeBinding`:** The interface for a group of classes that bind a single attribute value exposed in a data control to a view component
- **`JUCtrlHierBinding`:** The class that is used to bind tree, tree table, and table components to the model
- **`OperationBinding`:** The interface for a group of classes that bind command components to operations exposed in the model.

All the bindings for a particular page exist in a binding container. As you learned in an earlier lesson, the binding container is a run-time instance of a page definition file. There is a binding container for each data-bound page in the application.

The binding container together with the data controls give you the binding context.

You learn how to work with the binding context later in this lesson.

Exploring the Class Hierarchy

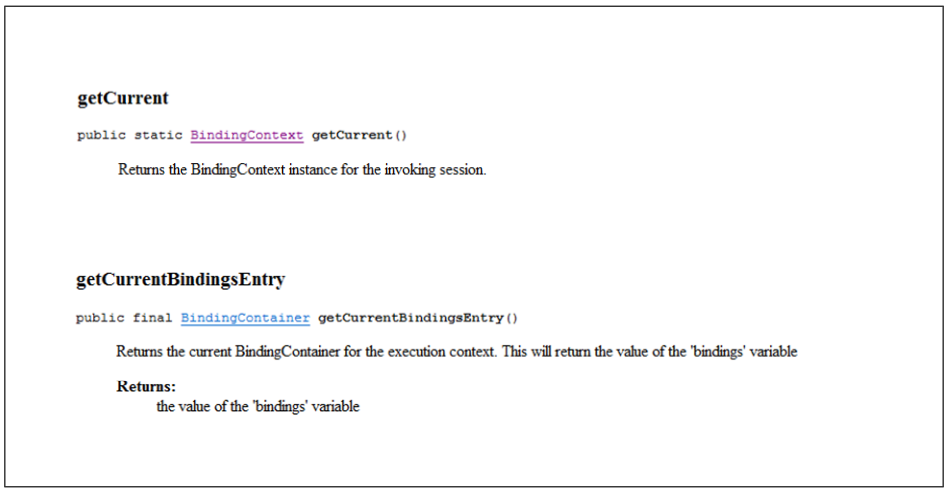


With ADF binding classes in particular, the structure of the inheritance can be quite deep. You should explore the class hierarchy to find the correct methods that you want to use.

Note: For a full description of the methods that you can call on bindings, see the *Java API Reference for Oracle ADF Model*:

http://docs.oracle.com/middleware/1212/adf/ADFMR/index.html

Commonly Used Methods in the `BindingContext` Class



getCurrent

```
public static BindingContext getCurrent()
```

Returns the BindingContext instance for the invoking session.

getCurrentBindingsEntry

```
public final BindingContainer getCurrentBindingsEntry()
```

Returns the current BindingContainer for the execution context. This will return the value of the 'bindings' variable

Returns:
the value of the 'bindings' variable

See the Java API documentation for the full list.

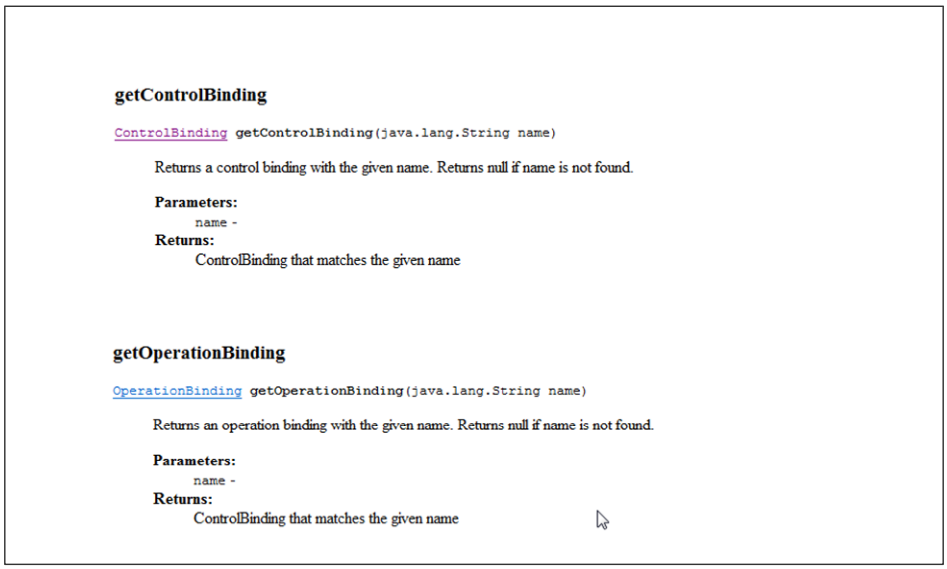
So far, you have learned that ADF provides a binding context, which contains a binding container for every data-bound page in your application. To work with ADF bindings programmatically, you first need to:

1. Get a reference to the binding context for the current session. To get a reference to the binding context, call the `getCurrent()` method.
2. Get a reference to the current binding container so that you can work with the bindings for the current page. To get a reference to the current binding container, call the `getCurrentBindingsEntry()` method on the binding context.

After you have a reference to the current binding container, you can get access to specific bindings for the current page. You learn how to do that next.

For a full description of the methods in the `BindingContext` class, see the *Java API Reference for Oracle ADF Model* at: http://docs.oracle.com/middleware/1212/adf/ADFMR/oracle/adf/model/BindingContext.html.

Commonly Used Methods in the `BindingContainer` Class



getControlBinding

`ControlBinding getControlBinding(java.lang.String name)`

Returns a control binding with the given name. Returns null if name is not found.

Parameters:
name -
Returns:
ControlBinding that matches the given name

getOperationBinding

`OperationBinding getOperationBinding(java.lang.String name)`

Returns an operation binding with the given name. Returns null if name is not found.

Parameters:
name -
Returns:
ControlBinding that matches the given name

See the Java API documentation for the full list.

After returning a reference to the current binding container, you can access specific bindings for the current page. For example, the slide shows two methods that you can use to access bindings:

- **`getControlBinding()`:** Enables you get a generic control binding (for data). For example, you can call this method to return an attribute binding for a text field or a hierarchy binding for a table.
- **`getOperationBinding()`:** Enables you to get an operation binding for methods and operations. For example, you can use this method to return the binding for a button.

In the next slide, you see a code example that demonstrates how to access several different types of bindings.

Note: For a full description of the methods in the `BindingContainer` interface, see the *Java API Reference for Oracle ADF Model*:

http://docs.oracle.com/middleware/1212/adf/ADFMR/oracle/binding/BindingContainer.html

Example: Accessing ADF Bindings from a Backing Bean

```
public String doAction() {
  //Get the binding context and the binding container
  BindingContext bcontext = BindingContext.getCurrent();                    1
  BindingContainer bcontainer = bcontext.getCurrentBindingsEntry();         2

  //Get an attribute value of department name
  AttributeBinding atb = (AttributeBinding)
      bcontainer.getControlBinding("DepartmentName");                       3
  System.out.println(atb.getInputValue());

  //Get an attribute value from the current row of a table
  JUCtrlHierBinding hib = (JUCtrlHierBinding)
      bcontainer.getControlBinding("EmployeesView3");                       4
  System.out.println(hib.getCurrentRow().getAttribute("Email"));

  //Call a built-in operation
  OperationBinding ob = bcontainer.getOperationBinding("Next");             5
  ob.execute();
  return null;  }
```

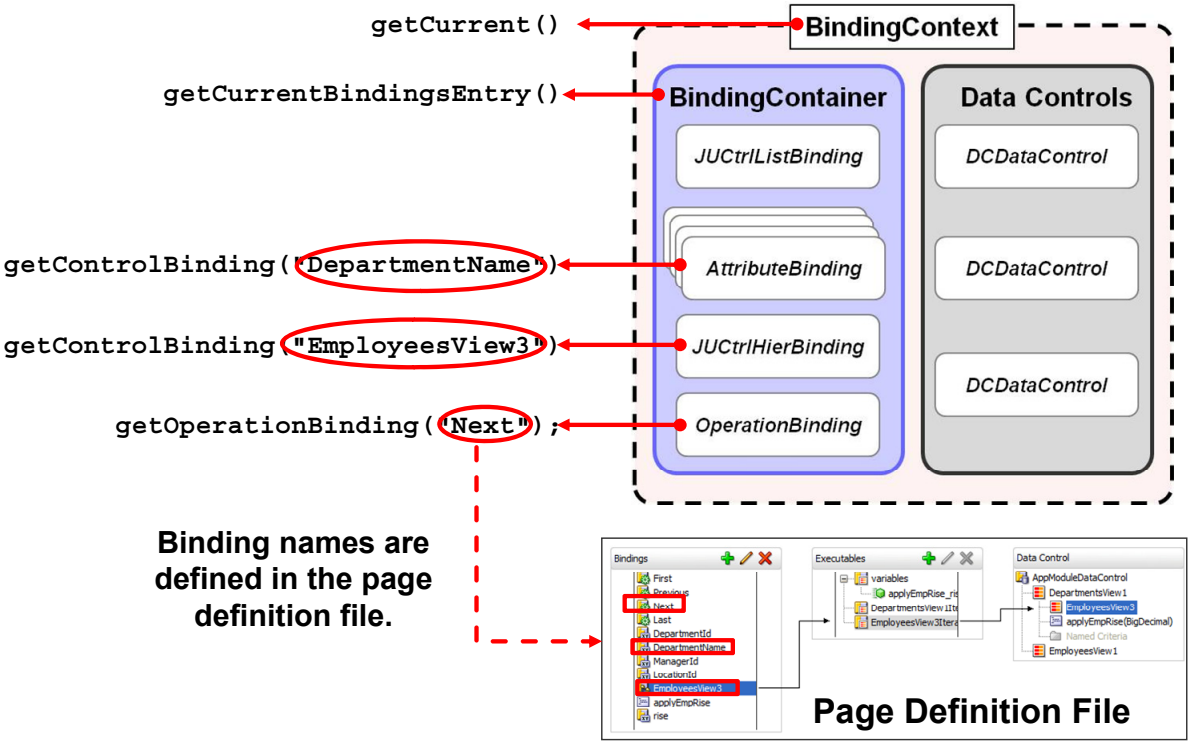
ORACLE

The example in the slide shows how to access the bindings for several different types of UI components. The code in the example:

1. Gets a reference to the binding context
2. Gets a reference to the binding container. The binding container gives you access to all the bindings on the page. From here, you can perform such actions as getting and setting values, as well as invoking a particular method.
3. From the binding container, gets an attribute binding named `DepartmentName`. The example shows how to get the value from the binding and print it to system output.
4. From the binding container, gets a reference to the hierarchy binding named `EmployeesView3`. With access to the hierarchy binding, you can perform such actions as getting the current row (remember that the hierarchy binding is a collection and not a single value). You can also get an attribute value from the row. The example shows how to get the attribute value for `Email` in the current row and print it to system output.
5. From the binding container, gets a reference to an operation binding. The operation binding enables you to call one of the built-in operations (for example, `Next`). The example calls the `execute()` method to invoke the operation.

This example shows the code for the backing bean. To call this code, you need to register the bean as a managed bean. The steps for creating and registering managed beans are described in the lesson titled “Adding Functionality to Pages.”

A Closer Look at the Example



This slide shows how the code in the example accesses specific types of binding objects.

- **`getCurrent()`:** Returns a reference to `BindingContext`, which contains all the bindings for the application plus the data controls
- **`getControlBinding("DepartmentName")`:** Returns a reference to the `AttributeBinding` object for the `DepartmentName` binding. This allows the code to access the value of the `DepartmentName` attribute.
- **`getControlBinding("EmployeesView3")`:** Returns a reference to the `JCtrlHierBinding` object for the `EmployeesView3` binding. This allows the code to access a collection of rows that contain employee records.
- **`getOperationBinding("Next")`:** Returns a reference to the `OperationBinding` object for the `Next` operation. This allows the code to invoke the built-in `Next` operation.

The slide also shows the bindings that are defined in the page definition file for the example. Notice that the binding names (`DepartmentName`, `EmployeesView3`, `Email`, and `Next`) are defined on the bindings page. For example, in the slide you see that the `Next` binding refers to an operation binding that is defined in the Bindings area.

If you are not sure which name to use in your code, you can look in the page definition file.

Summary

In this lesson, you should have learned how to:

- Generate Java classes for business components
- Override class methods
- Implement programmatic modifications
- Add service methods to an application module
- Use business component client APIs
- Access ADF bindings programmatically

Practice 13 Overview: Programmatically Customizing the Data Model

This practice covers the following topics:

- Generating extension classes for entity objects and view objects
- Overriding an inherited method in the subclass that you generated for `EntityImpl`
- Creating a custom method and adding it to the client interface
- Accessing ADF bindings programmatically from backing beans

14

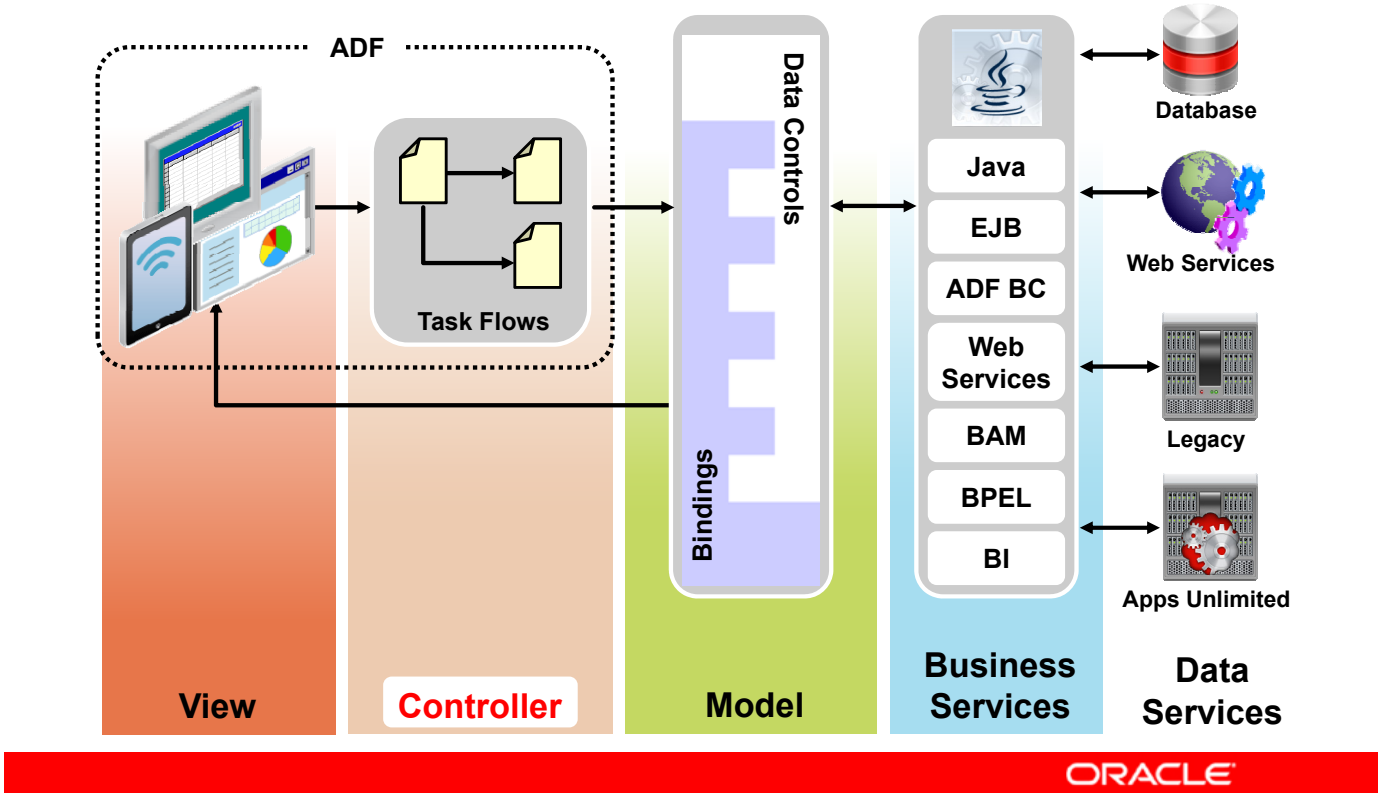
Implementing Transactional Capabilities

Objectives

After completing this lesson, you should be able to:

- Explain ADF BC transaction handling
- Identify the transactional scopes provided by ADF
- Implement task flow transaction control
- Specify data control scoping
- Handle transaction exceptions

Implementing Transactional Capabilities in the Controller Layer



Because ADF applications can use multiple data controls, the ADF Controller (ADFc) allows one or more data controls to be grouped together and committed or rolled back as a group. The controller does this through the facilities provided by the task flow transaction and data control scope options.

The underlying business services still own the transaction and ultimately execute the commit and rollback operations. But the ADF Controller defines the boundaries of the task flow transaction, namely, where the transaction starts and stops and which data controls are involved.

Handling Transactions with ADF Business Components

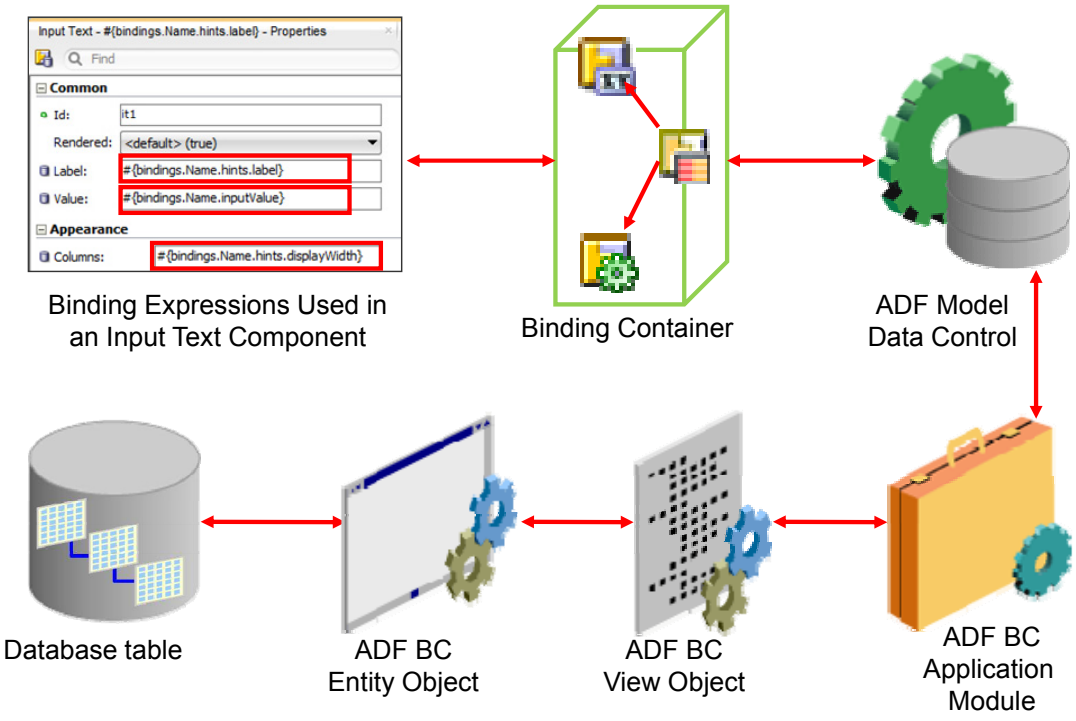
- Root application modules:
  - Handle transaction and concurrency support
  - Use a single database connection
  - Provide transaction capabilities for inserts, updates, and deletes for all view objects in the application module
  - Are all committed or rolled back at once
- No coding is required unless you want to customize the default behavior.
- For nested application modules, the root application module provides the transaction context for the others.
- To commit or roll back the root application module, you typically make use of the associated bindings of the same name.

ORACLE

A *transaction* is a persisted collection of work units that can be committed or rolled back together as a group. The Business Components framework handles all transactions at the application module level. An *application module* is a transactional container for a logical unit of work, so any updates, inserts, or deletes for all view objects in the application module are committed or rolled back at once. The outermost, or root, application module provides the transaction context for all nested application modules. This context also provides a single database connection.

At run time, ADF provides transaction management for the application. The transaction management is completely transparent to the application developer. There is no need for additional code, unless there is a need to customize the default behavior.

Default ADF Model Transactions



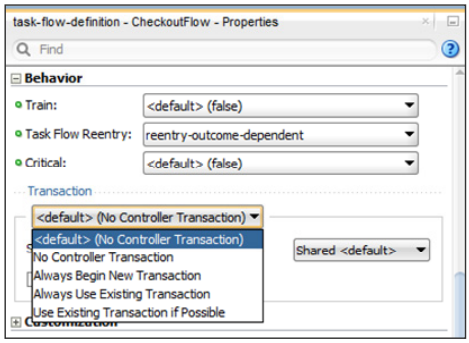
In this example, the user makes a change to one or more of the fields on the page and clicks Commit. The binding container associates the changes with the ADF model. The ADF model change is committed directly to the Transaction object of the current associated ADF BC application module instance to handle the transaction.

The Transaction object, which is transparent, then validates any entity rows that are marked as invalid in its pending changes list. When all of the rows pass validation, the Transaction object completes the database commit.

If there are any errors raised at the database level, those errors are pushed all the way up through the framework.

Task Flows and Application Modules

- Bounded task flows include options to manage the transaction behavior from the ADF Controller layer.
- Data control scopes:
  - Isolated
  - Shared
- Transaction options:
  - No Controller Transaction
  - Always Begin New Transaction
  - Always Use Existing Transaction
  - Use Existing Transaction if Possible



ORACLE

ADF bounded task flows provide multiple options for controlling transaction management. They also provide the ability to share the data controls of any calling task flow through data control scopes.

There are two basic groups of transactions options: using the controller and managing the transaction manually (No Controller Transaction). All of these options are discussed in the following slides.

Data Control Scopes

- Data control scopes determine whether:
  - Data controls are shared across task flows calls for each user session
  - A new instance of the data control is created for each
- Sharing behavior is implemented through data control frames.
- The following options are set on the task flow.
  - Shared
  - Isolated

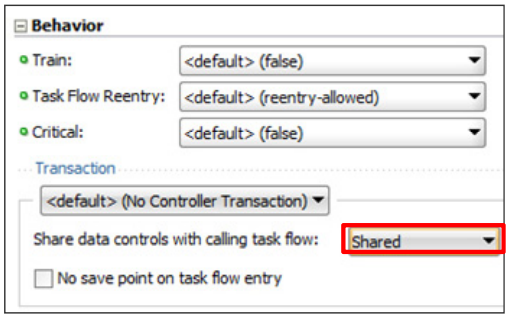
Data control scopes refer to the visibility or life of a data control across different parts of an application. When a task flow defines a shared data control scope, this implies the task flow will attempt to share any instance of a data control (and, by implication, its state) with the task flow's caller. This sharing will happen if the data controls have the same definition. This is in contrast to creating a new instance of a data control for a task flow, where no sharing occurs.

The sharing behavior is implemented through the use of *data control frames*.

Shared Data Control Scope

Shares data controls with calling task flow:

- Between calling and called task flows
- If data controls are of the same name and type
- Only one instance of the data control will be created and shared.
- View objects sharing:
  - Current row indicators
  - Edit the same records
- Single database connection



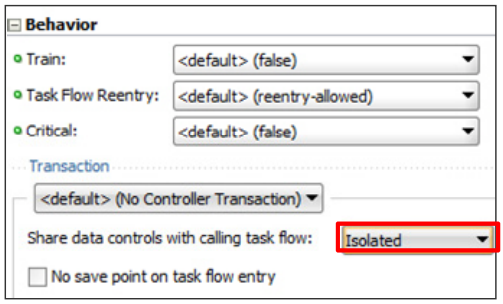
ORACLE

The *shared* option is used when you want to share the data control with the calling task flow. If the called task flow consumes data controls of the same name and type as the calling task flow, the called task flow will use those. It will create new instances of any data controls that are not used by the calling task flow. This means that, by default, the view objects being used will share current row indicators as well as share editing on the same rows. It also means that there will be only one database connection.

Isolated Data Control Scope

Does not share data controls with calling task flow:

- Uses multiple instances of the data controls
- Allows disparate data control transactions
- Separate view objects
  - No record coordination
  - Isolated record editing
- Multiple application module instances
- Multiple database connections



The *isolated* option makes the called task flow self-contained, which means that it does not share data controls from the calling task flow. It will create its own instance of any data controls it needs. As a result, it will have completely separate data control transactions. It will therefore also have separate view object instances, which means there is no row coordination with the calling task flow. The called task flow will not be aware of currently selected rows or edits made by the calling task flow. It will also create a separate instance of the application module and will instantiate its own database connection.

Data Control Frame

- Is an internal run-time ADF object that groups task flows and their data controls at run time
- Defines the boundaries of the ADF controller/task flow transactions
- An unbounded task flow has its own data control frame.
- A bounded task flow with a data control scope of *isolated* has its own data control frame.
- A bounded task flow with a data control scope of *shared* uses the data control frame of the calling task flow.

As you have just seen, when one task flow calls another, the task flow can either share an instance of a data control or create a separate isolated instance of the same data control. The internal object that task flows use to share their data controls or to store their own isolated data control is known as a *data control frame*.

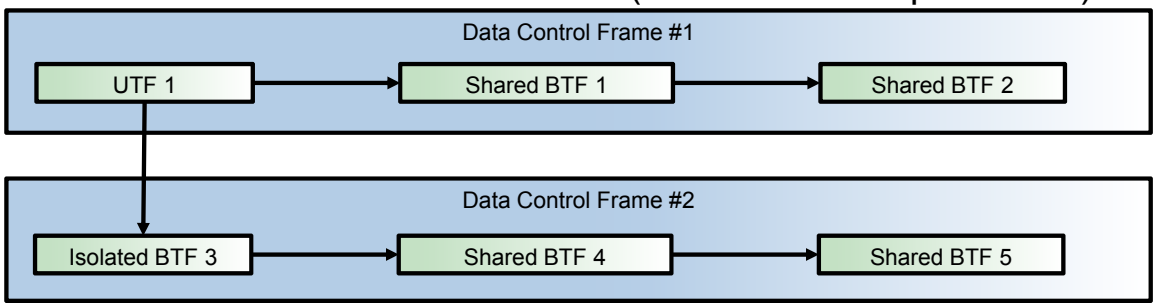
Task flows making use of the task flow transaction management options, when committing or rolling back, use the data control frame to know which data controls to perform the transaction operations on. A data control frame is created at run time for your application's unbounded task flow and for any bounded task flow with a data-control-scope value of *isolated*. When a task flow specifies a data control scope value of *shared*, the called task flow uses the data control frame of the calling task flow rather than creating its own.

This allows the called task flow to share data control instances attached to the data control frame. Alternatively, if a called task flow specifies a data control scope value of *isolated*, a new data control frame is created and a new instance of any data controls used by the bounded task flow will be attached to the newly created data control frame.

To specify whether data controls are shared between the calling and called task flows, you must set a data control scope value of either *shared* or *isolated* on the called bounded task flow. The default value is *shared*.

Data Control Frame: Examples

- Example #1 includes:
  - Unbounded task flow 1 (UTF 1)
  - Bounded task flow 1 (data control scope *shared*)
  - Bounded task flow 2 (data control scope *shared*)
- Example #2 is created for:
  - Bounded task flow 3 (data control scope *isolated*)
  - Bounded task flows 4 and 5 (data control scope *shared*)



In this example, when Unbounded Task Flow 1 (UTF1) calls Bounded Task Flow 1 (BTF1), it shares its data control frame because BTF1 specifies the data control scope as *shared*. Likewise, Bounded Task Flow 2, which also specifies a *shared* data control scope, becomes part of data control frame 1.

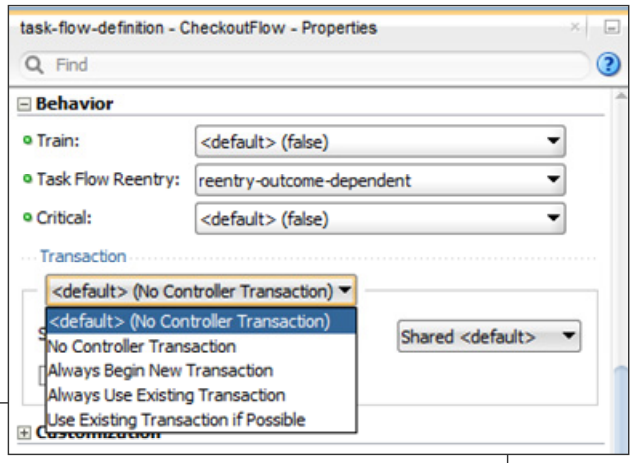
In contrast, when UTF1 calls Bounded Task Flow 3, which specifies an *isolated* data control scope, a new data control frame is created for that task flow. However, because Bounded Task Flows 4 and 6 specify a data control scope of *shared*, they participate in data control frame 2.

So, in this application case, there are two data control frames that maintain independent states.

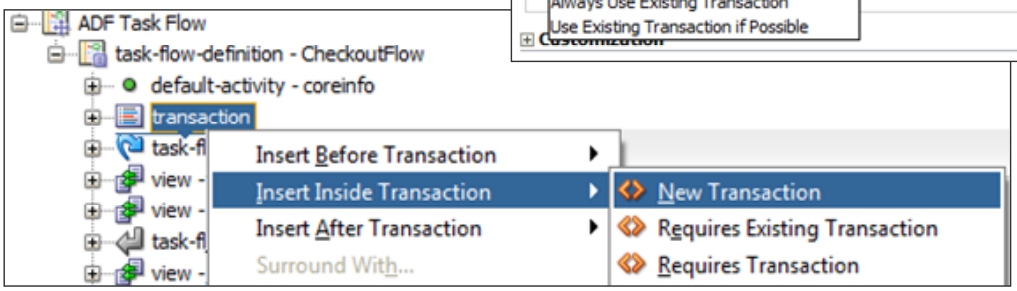
Specifying Transaction Options

Specify transaction start options on a called ADF bounded task flow.

In Properties window:



In Structure window:

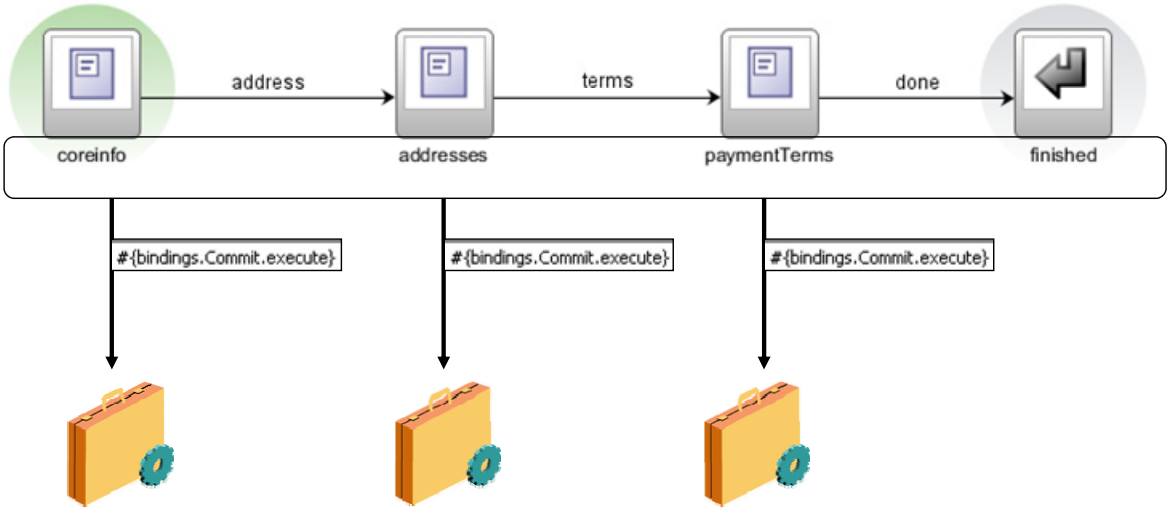


Transaction options are set on the called task flow and specify whether a called ADF bounded task flow should:

- Use no controller transaction
- Always begin a new transaction
- Always use an existing transaction
- Use an existing transaction if possible

If no transaction option is specified, a transaction is not started on the entry of the called ADF bounded task flow. A run-time exception is thrown if the ADF bounded task flow attempts to access transactional services.

"No Controller" Option



This example shows a bounded task flow that uses the No Controller managed transaction option. To help show how this affects transaction management, each of the pages consumes data controls from different application modules. Because each one is separate, the developer is responsible for managing each commit separately.

If you have pages that are based on different application modules, allowing the application modules to control the transactions means that the changes on each page are committed separately. A commit failure on one of the application modules does not prevent the other changes from being committed, because the transaction object of each application module is separate from the others.

If you choose the No Controller option for transaction management, you must provide code to manage the transactions.

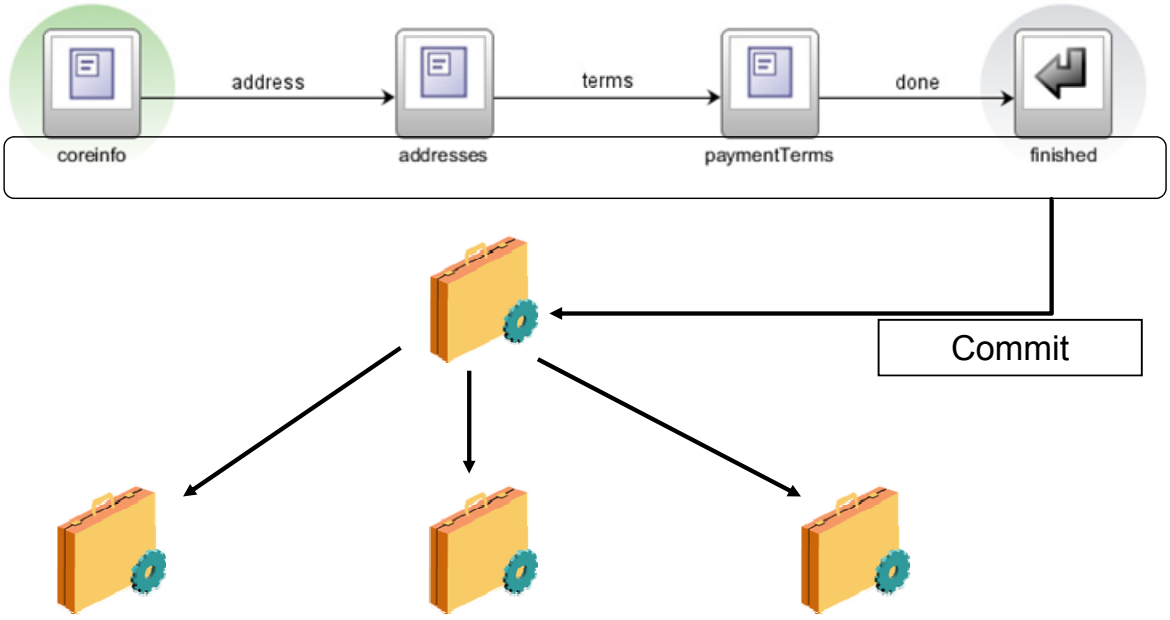
<No Controller Transaction> Option

- Would be better named <No Controller “Managed” Transaction>
- Using this option discards the managed transactions at the controller level:
  - Thus the option name: “The controller is not controlling the transaction.”
  - Similar to how applications were built in JDeveloper 10*g*
  - Simply turns off Controller transaction management
- This option can be used in combination with the three other task flow transaction options.
- Mixing the No Controller option with controller options in a chain of task flows is not recommended.

The No Controller Transaction option really means that the Controller does not manage the transaction. This option simply turns off the transaction management capability of the controller. With this option, as you saw in the previous slide, the developer chooses to allow the application module to manage the transaction. If all data controls are from the same application module, an application module level commit will commit all the changes in the task flow. If the data controls are from different applications, the developer must add a commit from each of the involved application modules.

You can use this option in combination with the other options. However, mixing the No Controller option with the Controller options in a chain of task flows is not recommended. This combination may cause conflicts in how data controls are included in transactions.

Controlling Transactions in Task Flows

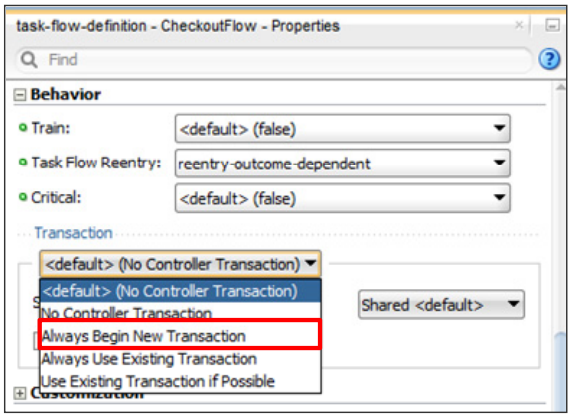


This is the same example you saw earlier. However, in this example we use a Controller option to manage the transaction. In the example, we use an ADF bounded task flow to represent a transactional unit of work and to declaratively manage transaction boundaries.

When you allow the bounded task flow to control the transaction, any application modules involved are treated as nested application modules. The task flow interacts with the transaction object of a “dummy” application module, which provides the transaction context for all the nested application modules. If the commit on one of the nested application modules fails, the entire transaction fails.

"Always Begin New Transaction" Option

- Starts a new transaction on the data control frame
- Is used to start the task flow transaction boundary in a chain of bounded task flows
- Supports both *isolated* and *shared* data control scope:
  - Because you typically use this option to start a new transaction, *isolated* makes the most logical sense.



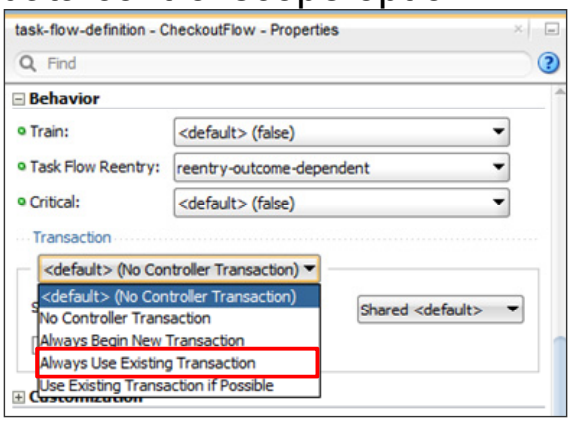
- Is responsible for finalizing the transaction

ORACLE

The Always Begin New Transaction option always begins a new data control frame. You usually select this option for the first task flow in a chain of task flows to set the beginning of the transaction boundary. This option supports both *shared* and *isolated* data control scopes, but because this option usually signals the start of a new transaction, *isolated* makes better sense. When you use this option, the task flow is responsible for finalizing the transaction with either a commit or rollback.

"Always Use Existing Transaction" Option

- Joins an existing data control frame transaction
- Used in a chain of bounded task flows in which you need the current task flow to join the transaction of the previous task flow
- Supports only the *shared* data control scope option (*isolated* is disabled)
- Cannot finalize the transaction



ORACLE

You use the Always Use Existing Transaction option when this task flow *must* be a part of an already-started transaction. For example, you would use this on a task flow that is used to add invoice line items to an invoice. An invoice line item task flow should not try to commit or roll back an invoice transaction, but it does need to be part of the transaction that creates the invoice. So the end activity of this type of task flow should be a return to the calling task flow.

Because the task flow joins an existing transaction, the data control scope can only be set to *shared,* with *isolated* disabled.

"Use Existing Transaction if Possible" Option

- Supports *isolated* and *shared* data control scope
- The transaction behavior is dynamic:
  - For *isolated* data control scope, the task flow has Always Begin New Transaction behavior at run time.
  - For *shared* data control scope:
    - If an open data control frame transaction is detected, the task flow behavior becomes Always Use Existing Transaction.
    - If there is no open data control frame transaction, the task flow behavior becomes Always Begin New Transaction.

Using the Task Flow Transaction Options

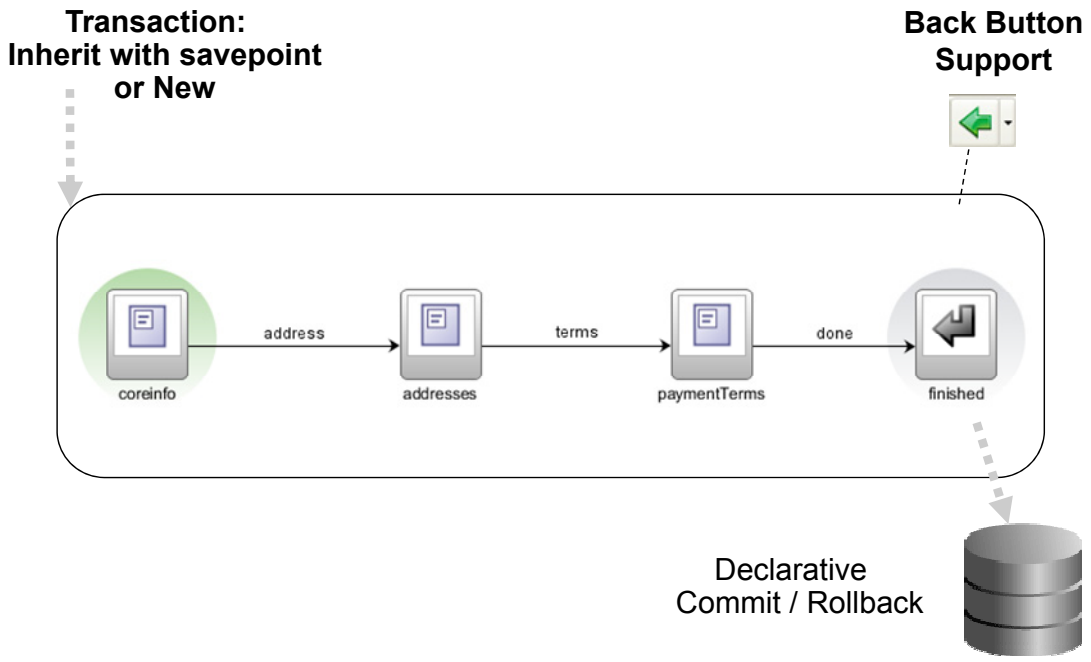
- <no controller transaction>
  - You must call the data control commit/rollback operations yourself.
  - You must add commit/rollback for each application module.
- If you use the three other task flow transaction options:
  - There is only one database connection
  - They are more scalable than <no controller transaction>
  - You should use the task flow return commit/rollback
    - The Controller will take care of the commit/rollback for all data controls attached to the data control frame as a group
- For separate transactions and connections:
  - Use isolated data control scope OR
  - Use <no controller transaction>

You should remember that if you choose the No Controller option, you must call the data control commit or rollback operations yourself. You must also have a commit or rollback for each application module that is consumed by the task flow.

If you use any of the three other options, there is only one database connection created, and those other options are generally more scalable than the No Controller option. With these options, you use the task flow commit or rollback, which will coordinate with the data control frame to commit the transaction as a unit, regardless of the number of application modules that are involved in the transaction.

If you need to have separate transactions and multiple database connections, you can use either the *isolated* data control scope or the No Controller option.

Transaction Support Features of Bounded Task Flows



Transaction options on the called task flow definition specify whether a called ADF bounded task flow should join an existing transaction, create a new one, or create a new one only if there is no existing transaction.

If the called ADF bounded task flow is able to start a new transaction (based on the transaction option that you select), you can specify whether the transaction is committed or rolled back when the task flow returns to its caller.

In a called task flow definition, you can specify two different return task flow activities that result in either committing or rolling back a transaction in the called ADF bounded task flow.

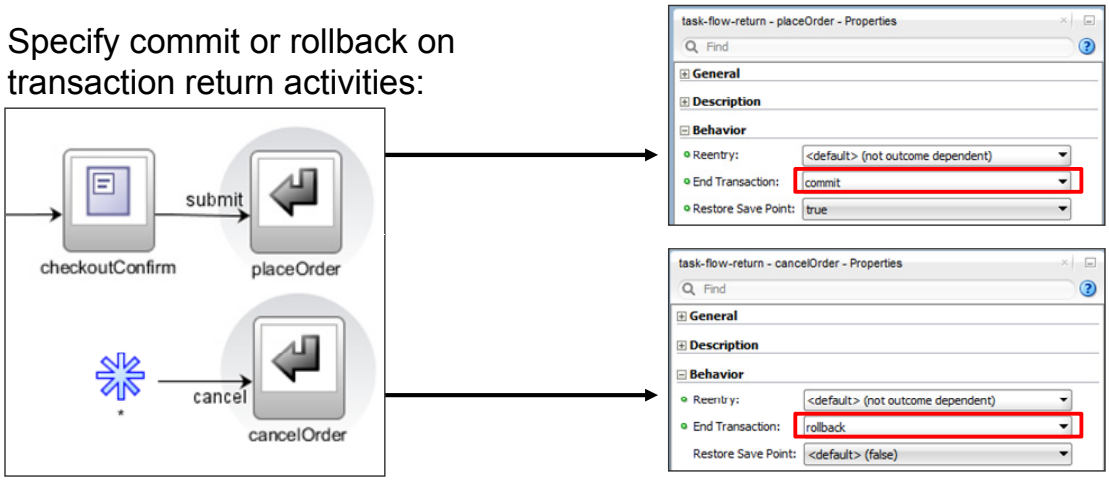
You can also specify how a page should behave when users navigate to it by using the Back button.

If you are using the No Controller Transaction option, the developer must use the commit and rollback options from the Data Control Palette. If you are using any of the other three options, use the task flow commit and rollback operations. You should not mix the two options.

Specifying Task Flow Return Options

Transaction Options	Commit or Rollback
No Controller	Not allowed
Begin New	Required
Always use existing	Not allowed
Use existing if possible	Required

Specify commit or rollback on transaction return activities:



If the called ADF bounded task flow is able to start a new transaction (based on the transaction option that you selected), you can specify whether the transaction should be committed or rolled back when the task flow returns to its caller. The table at the top of the slide shows the requirements to specify commit or rollback for each transaction option.

The commit and rollback options are set on the task flow return activity that returns control to the calling task flow. To set these options, use the End Transaction property in the Properties window under Behavior. The same task flow that starts a transaction must also resolve or finalize the transaction.

In a called task flow definition, you can specify two different return task flow activities that result in either committing or rolling back a transaction in the called ADF bounded task flow. Each of the task flow return activities passes control back to the same calling task flow. The difference is that one task flow return activity specifies the commit option, whereas the other specifies the rollback option.

In the example in the slide, if transaction processing completes successfully, the `submit` control flow rule passes to the `placeOrder` task flow return activity, which specifies the options to commit the transaction. If the transaction is cancelled before completion, the `cancel` control flow rule passes control to the `cancelOrder` task flow activity, which specifies the options to roll back the transaction.

Handling Transaction Exceptions

- With a bounded task flow, you can designate any activity as the exception handler.
- The exception activity is invoked if an exception is raised.
- Best practice is that you should designate an exception handling activity on transactional task flows.
- The exception handling activity can be any activity type, such as:
  - **View:** To display a message
  - **Router:** To call a method depending on the type of exception

ORACLE

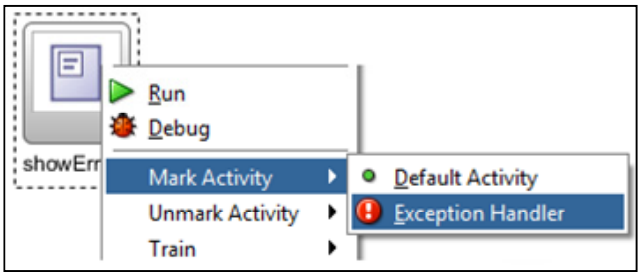
During the execution of an ADF task flow, exceptions can occur that may require some kind of exception handling, such as when a method call activity throws an exception or when a user is not authorized to execute the activity.

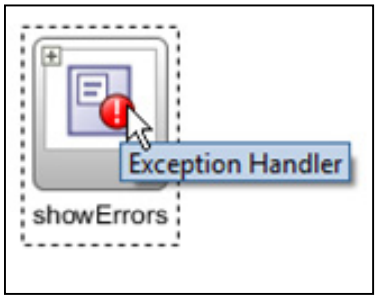
To handle exceptions thrown from an activity or caused by some other type of ADF controller application error, you can create an exception handler for an ADF bounded task flow. When an exception is raised in the task flow, the control flow passes to the designated exception handling activity. The exception handling activity can be any supported activity type, as in the following examples:

- A view that displays an error message
- A router activity that passes the control flow to a method based on an expression that evaluates the type of exception

As a best practice, any ADF bounded task flow representing a managed transaction should designate an exception handling activity. When running an ADF bounded task flow managed transaction, the ADF controller attempts to commit the transaction when it reaches a task flow return activity identified in metadata for a commit. If the commit throws an exception, the control is passed to the ADF bounded task flow's exception handling activity. This gives the user a chance to correct any data and then reattempt to commit it. You can use the exception handling activity to display a warning message on a page that tells the user to correct the data and attempt to commit it again.

Designating an Exception Handler






```
<exception-handler>ShowError</exception-handler>
...
...
<view id="ShowError"></view>
```


To designate an activity as an exception handler for a task flow:

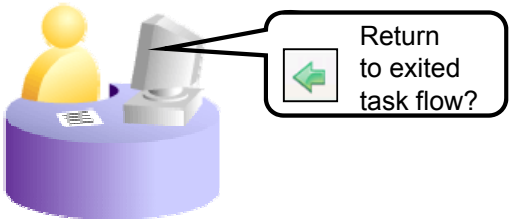
1. Right-click the activity in the task flow diagram, and then select Mark Activity > Exception Handler.
   A red exclamation mark is superimposed on the activity in the task flow to indicate that it is an exception handler.
2. To unmark the activity, right-click the activity in the task flow diagram, and then select Unmark Activity > Exception Handler.
   If you mark an activity as an exception handler in a task flow that already has a designated exception handler, the old handler is unmarked.

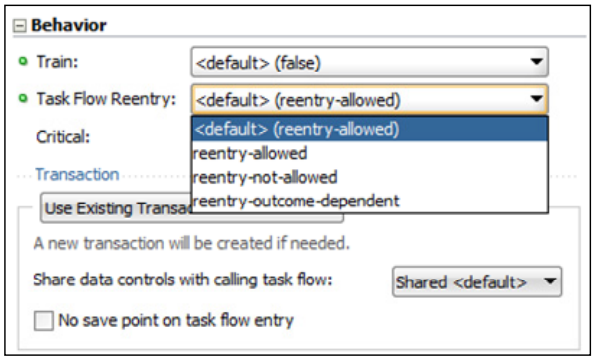
After you designate an activity as the exception handling activity for a task flow, the task flow metadata updates with an `<exception-handler>` element that specifies the ID of the activity (as shown in the slide).

Defining Responses to the Back Button

The Task Flow Reentry property determines whether the user can return to an exited task flow by clicking the browser's Back button:

- reentry-allowed: OK
- reentry-not-allowed: Throws exception
- reentry-outcome-dependent: Depends on the outcome of the exited task flow





ORACLE

For situations in which the end user clicks the Back button to navigate back to an ADF bounded task flow that was already exited, you can specify the following `task-flow-reentry` options. These options specify whether a page in the ADF bounded task flow can be reentered.

- **reentry-allowed:** Reentry is allowed on any view activity in the ADF bounded task flow.
- **reentry-not-allowed:** Reentry of the ADF bounded task flow is not allowed. An end user can still click the browser Back button and return to a page in the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, InvalidTaskFlowReentry) is thrown to indicate that the bounded task flow was reentered improperly. The actual reentry condition is identified on the submit of the reentered page.

- **reentry-outcome-dependent:** Reentry of an ADF bounded task flow using the browser Back button is dependent on the outcome that was received when the same ADF bounded task flow was previously exited via task flow return activities. For example, a task flow representing a shopping cart can be reentered if the user exited by canceling an order, but it cannot be reentered if the user exited by completing the order. On reentry, ADF bounded task flow input parameters are evaluated using the current state of the application rather than the application state existing at the time of the original ADF bounded task flow entry.

If the Task Flow Reentry property of the task flow is set to reentry-outcome-dependent, a property of the task flow return activity determines whether the user can use the Back button to navigate back to the called task flow. On the Behavior tab of the task flow return's Properties window, you can set the Task Flow Reentry property to reentry-allowed or reentry-not-allowed. This enables you to specify a different response to the Back button depending on the outcome that is returned to the calling task flow.

When a user reenters an ADF bounded task flow by using the browser Back button and reentry is allowed, the value of a managed bean on the reentered task flow is set back to its original value—the same value that it had before the user left the original task flow.

This results in the managed bean value resetting back to its original value before a view activity in the reentered task flow is rendered. Any changes that occurred before reentry are lost. To change this behavior, specify the <redirect> element on the view activity in the reentered bounded task flow. When the user reenters the bounded task flow using the Back button, the managed bean has the new value from the parent task flow, and not the original value from the child task flow that is reentered.

Summary

In this lesson, you should have learned how to:

- Explain ADF BC transaction handling
- Identify the transactional scopes provided by ADF
- Implement task flow transaction control
- Specify data control scoping
- Handle transaction exceptions

Practice 14 Overview: Implementing Transactional Capabilities

This practice covers the following topics:

- Specifying transaction options on a bounded task flow
- Defining an exception handling activity for the bounded task flow

15

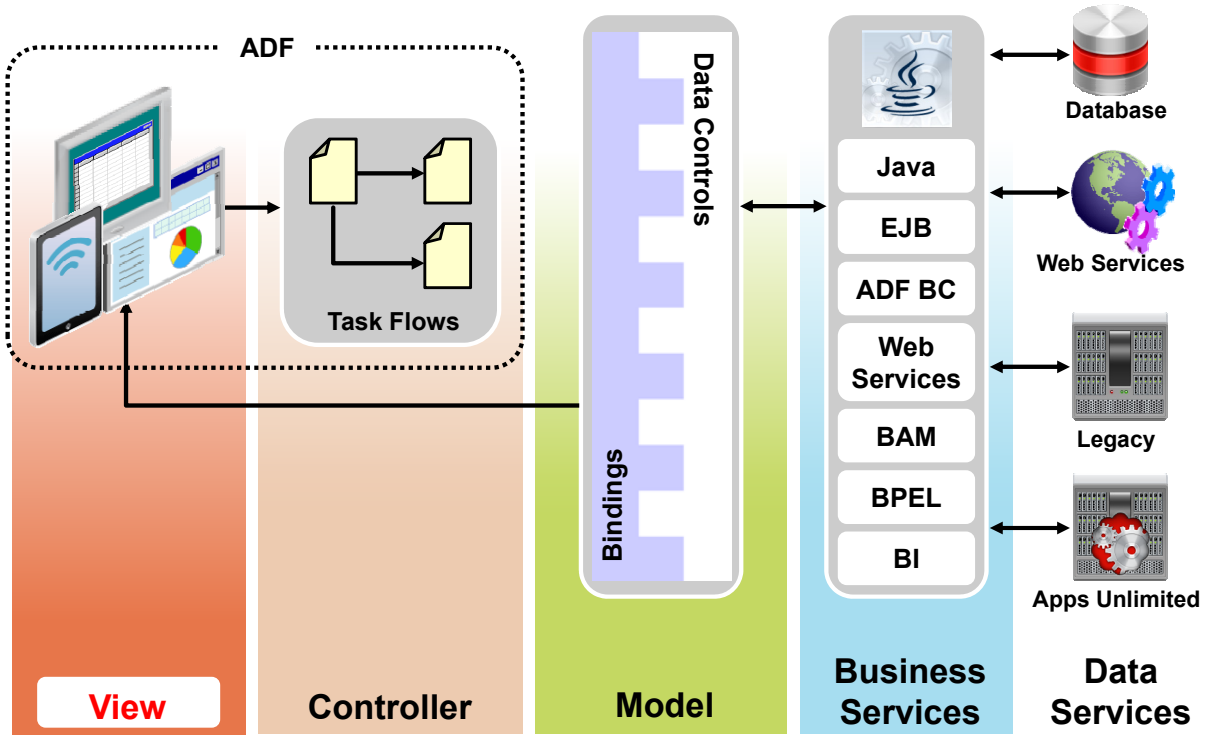
Building Reusability into Pages

Objectives

After completing this lesson, you should be able to:

- Create and use page templates
- Create and use page fragments
- Describe the functionality for packaging reusable components into libraries

Building Reusability into Pages



In a previous lesson, you learned about the fundamentals of building the view layer. You learned about ADF Faces and the underlying JavaServer Faces technology on which ADF Faces is built. You also learned about ADF Faces UI components and about using data controls to build data-bound UI components.

In this lesson, you learn how to create and use components that are reusable, including page templates, page fragments, and bounded task flows in regions. You also learn how to package reusable components into libraries.

ADF Reusability Features

Several reusability features exist for pages in Oracle ADF:

- Page templates for enabling a consistent look across pages
- Page fragments that can be displayed within a page that contains other content
- Regions that enable you to insert a bounded task flow composed of page fragments as part of a larger page
- ADF libraries for packaging these and other reusable components into libraries

ORACLE

Duplicating code or content is one of the worst practices in application development. Code reuse increases developer productivity and application maintainability. It can sometimes be an easy solution to copy code, but changing the application becomes much more difficult, making the application less able to adapt to new requirements or fix problems quickly. Ideally, an application should not contain multiple versions of the same code in the model, view, or controller layers.

Application-specific code can also be reused across modules and even across related projects. This latter type of reusability enables developers to make rapid changes, exploit new features globally, and spend less time testing and debugging.

A core principle of ADF is reusability, and this also extends to page design. Oracle ADF provides several reusability features. This lesson focuses primarily on reusability features at the page level, which include page templates, page fragments, regions, and ADF Libraries. There are other reusability features in ADF, including reusable ADF Faces components, that are outside the scope of this course.

Page Templates

- Are reusable JSF pages that enable a consistent look across an application
- Are built from standard ADF components and can contain model components
- May be nested
- Use partial page refresh when navigating between pages that use the same template
- Use three types of files:
  - Page-specific definition file (`.jsf`)
  - Optional page data binding definition file (`.xml`)
  - Definition file for all templates: `pagetemplate-metadata.xml`

ORACLE

A page template provides a reusable page definition that enforces a consistent page layout across an application.

Page or page fragment developers define pages that use the template and implicitly inherit the look and feel and prescribed layout of the template. Template developers can change the page layout for all the pages or page fragments that use the page template by changing only the page template definition.

Page templates use standard ADF Faces components, and they can contain attributes and ADF bindings for greater flexibility. A page template can be nested within another page template, and it can also be based on an existing page template. Pages that use the same page template take advantage of partial page refresh so that only the content that is specific to a page changes without the need to refresh the template.

There are two sets of page template definition files.

- For each page template, there is:
  - A JSF file (`.jsf`), which defines the rendered UI in terms of other JSF components, facet references, and EL expressions that are resolved against the page template's UI attributes and model
  - An optional model-specific page definition (`.xml`) file
- Across all page templates, there is:
  - A `pagetemplate-metadata.xml` file, which contains the paths of all the page template definitions in the project (located in the project's `META-INF` directory)

Components of a Page Template

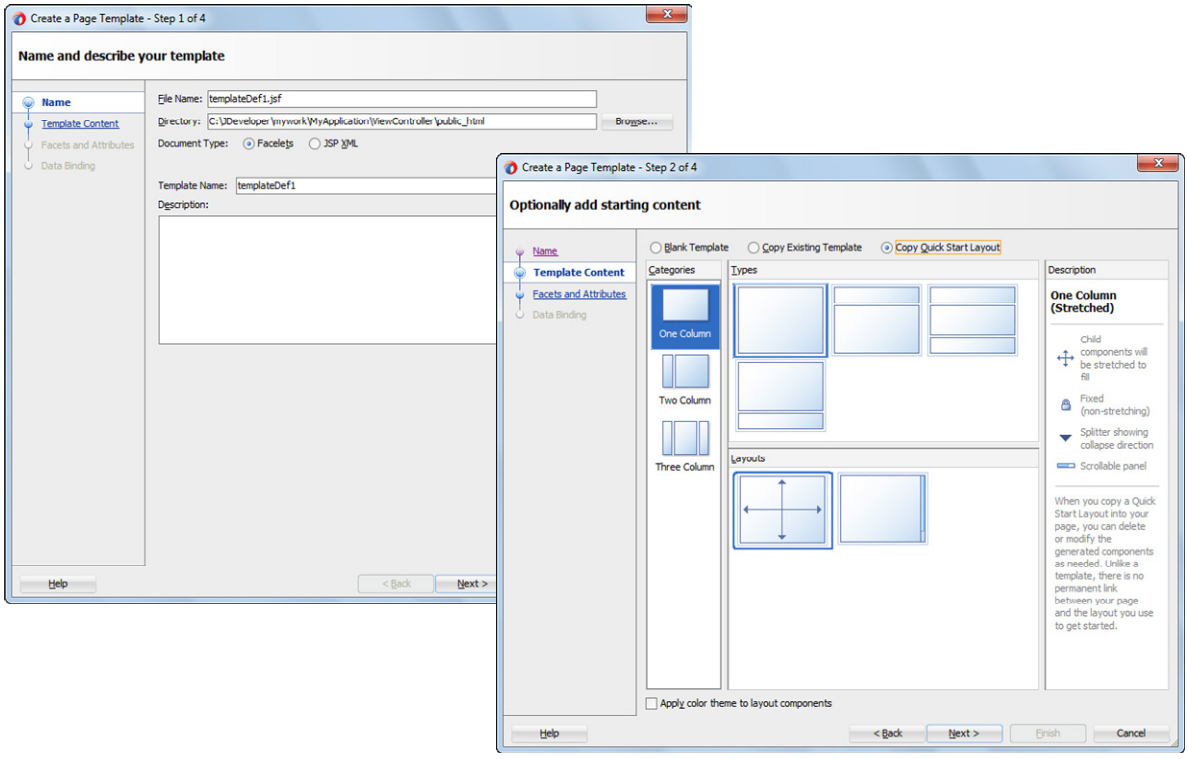
- **Facet:** Is a placeholder where page developers can add content
- **Attribute:** Enables page developers to provide information to use the template in a page-specific way
- **Model parameter:** Enables bindings in a template to use a parameter values passed in by a page that uses the template

The main component of a page template is called a *facet*. Facets that are defined on the page template provide areas where page developers include content that is specific to the page they are building.

Attributes are optional components on a template. They enable page-specific information to be provided to various components in the template (for example, to enable certain areas of the page to be hidden or shown depending on the attribute value provided by the page developer).

Model parameters provide a way to parameterize the bindings on a template. For example, to define the display of a list of customer orders in a template, a model parameter might be created to enable page developers to pass in the `customerId` binding from the referencing page.

Creating Page Templates

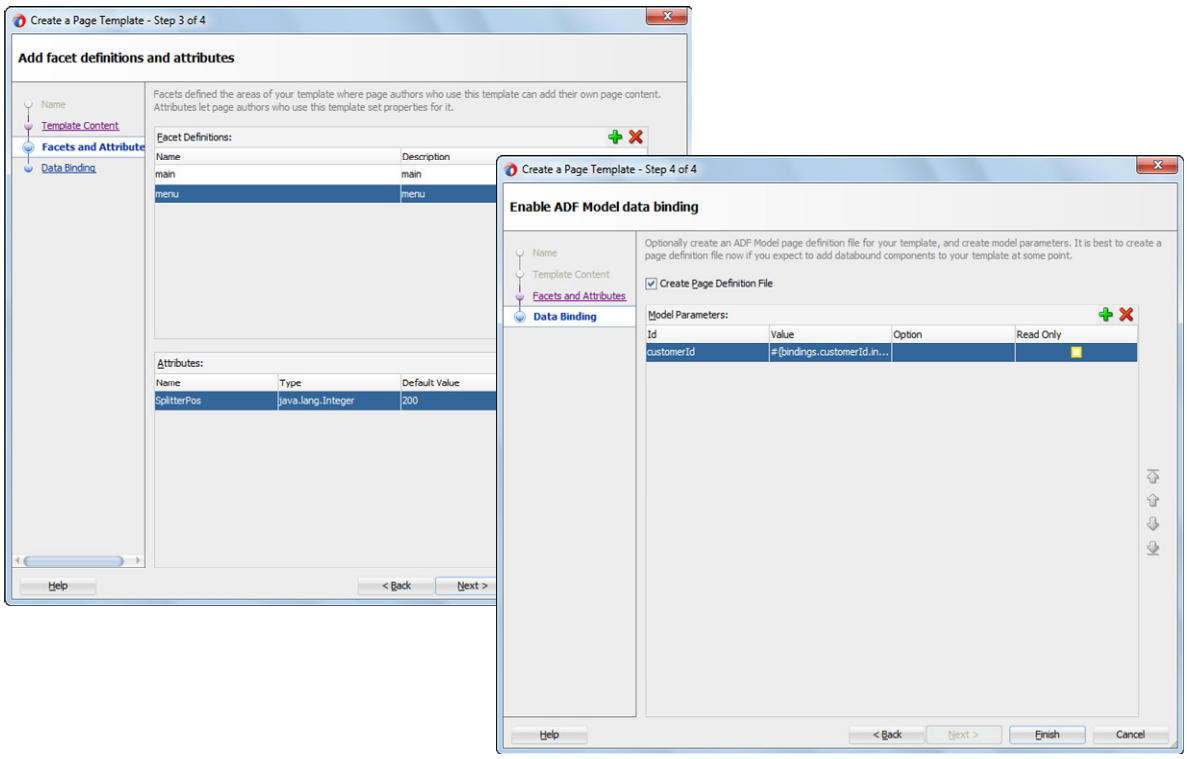


You can create a page template from the New Gallery (File > New > From Gallery) by selecting Web Tier > JSF/Facelets > ADF Page Template.

On the first two pages of the Create Page Template Wizard, you define:

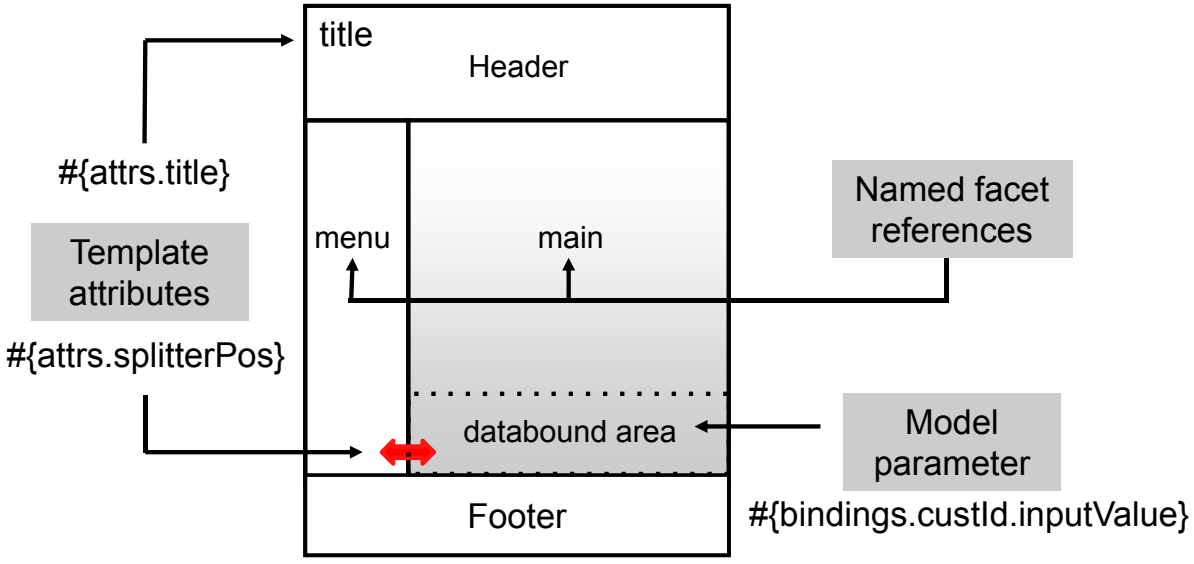
- The file name for the template, including the extension (`.jsf` or `.jspx`) plus the template name
- A quick start layout for the template. Either use a provided quick start layout, or copy from an existing template.

Defining Page Templates



In steps 3 and 4 of the page template, you define facet definitions, attributes, and model parameters. These components can also be added to the template after initial creation.

Creating a Page Template



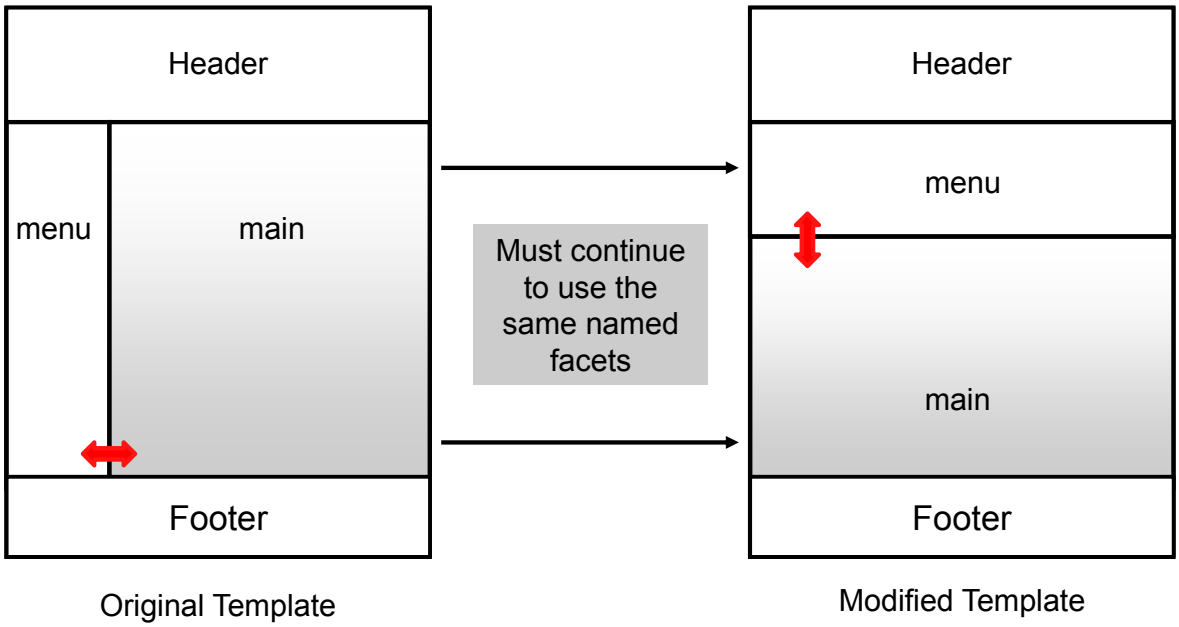
After you close the Create Page Template Wizard, the page template opens in the visual editor. You can add components by dragging them from the Components window, and you use the Properties window to set properties on components to affect appearance and behavior.

The example in the slide shows the header, footer, menu area, and main content area that appear on all pages that use this template. In this case, the template developer has created static header and footer content. Additionally, `facets` have been defined for `menu` and `main`, which are placeholders where the page developer who uses this template can drop content. This content could be any JSF component, region, or portlet.

In addition, template developers can add attributes to the page definition that are accessible via Expression Language (EL). For example, the template definition may include a title attribute and place an output text component on a specific area of the page with its value property specified as `#{attrs.title}`. The user of the template needs to define the text for only the title attribute, which then appears in the correct place on the page. Attributes might also be used to customize other component properties (such as a `splitterPosition`, `rendered`, or any other property) to enable dynamic use of the template.

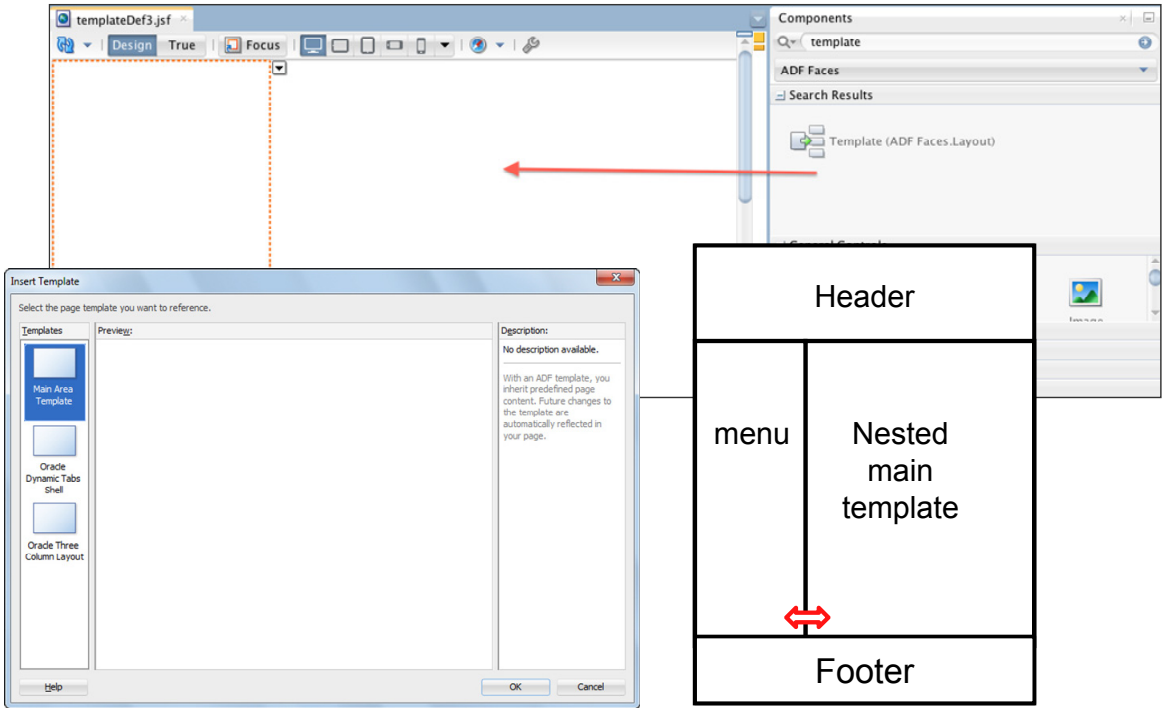
Finally, model parameters can be defined to parameterize the way that bindings are used on the page template. The EL is evaluated at run time; in this case, the `#{bindings.custId.inputValue}` expression is evaluated in the context of the page that uses this template, rather than the template itself. As a result, the page template can specify that a list of customer orders should be displayed in a certain place on the page, and the page that references the template will provide the correct `custId` binding value to display the current customer's orders.

Editing Page Templates



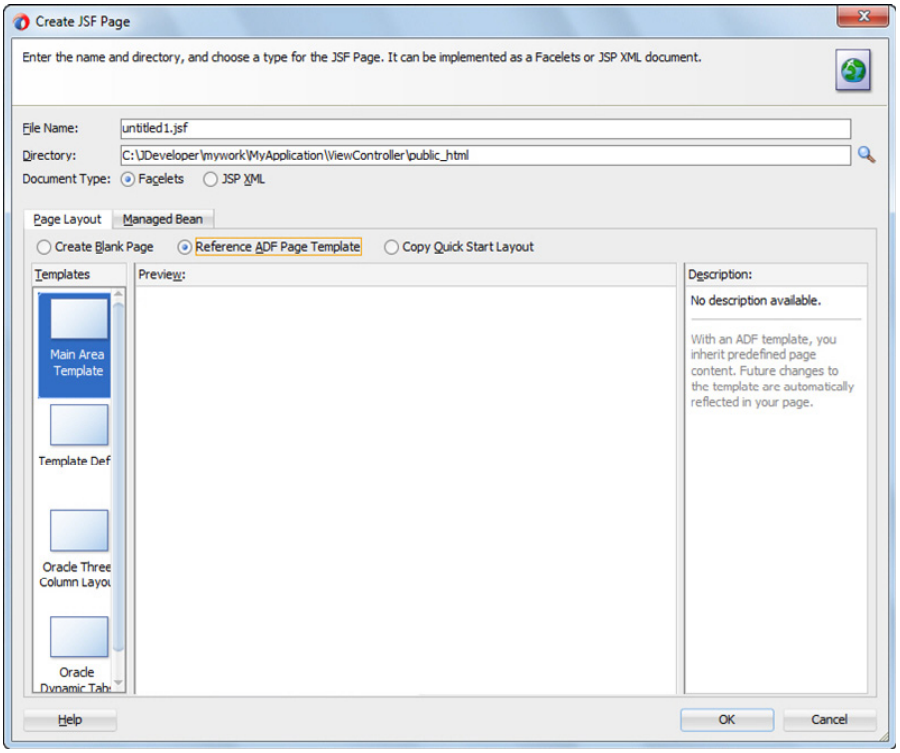
An important advantage of page templates is that, when you edit the template, any page that uses the template picks up the changes automatically. However, you are required to maintain the named `FacetRef`s in the template to avoid breaking pages that use the template. The modified template shown in the slide continues to use facets `menu` and `main` after modification, as well as the same `splitterPos` attribute that was defined, so pages built using that template will still work.

Nesting Page Templates



To nest templates, create the innermost template first. Then, for each page that references one or more nested templates, drag the template component to the visual editor. In the resulting dialog box, select the template to include at the specified location. Then add facets to the outermost template that refer to the corresponding facet names of the nested template.

Applying a Page Template to a Page



When you create a JSF page in JDeveloper, the wizard gives you the choice to select an available page template on which to base the creation of the skeleton JSF page. The template defines the entire page layout as it is defined in the template definition file.

Pages that are created by using page templates are accessed, packaged, versioned, bookmarked, and consumed just like any other stand-alone pages.

You can also apply a page template to a page fragment. Page fragments are discussed later in this lesson.

Note: A page template does not necessarily need to be used to construct an entire page or page fragment. It can also be used for reusable sections of the page. However, there is no design-time support for this, and doing so is outside the scope of this course.

Page Fragments

- Are built like regular pages
- Have page definition files like regular pages do
- Are not defined as a full webpage
- Can be displayed on a page that contains other content
- Cannot contain `af:document` or `f:view` tags
- Cannot be run on their own
- Are used primarily in bounded task flows

It is easier to maintain large, complex pages if they are divided into several page fragments. Depending on how you design a page, the page fragments created for an entire page can also be reused on other pages. For example, suppose that different parts of several pages use the same components; you might find it beneficial to create page fragments containing those components and reuse those page fragments on several pages or in several places on the same page.

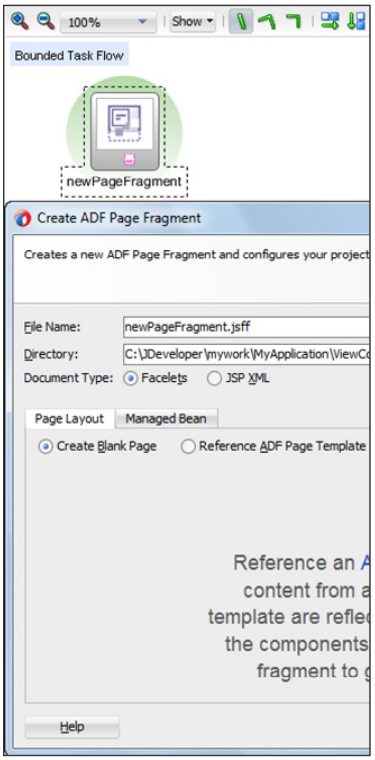
Deciding how many page fragments to create for one or more complex pages depends on your application, the degree to which you want to reuse portions of a page between multiple pages, and the need to simplify complex pages.

Page fragments are created and used much like pages, except that a page fragment is not defined as a full webpage. A page fragment does not contain `f:view` or `af:document`. Its contents are simply enclosed within `jsp:root`, and page fragments are therefore not runnable on their own. As a result, page fragments are used primarily in bounded task flows.

Creating a Page Fragment

You can create a page fragment in any of the following ways:

- Using the New Gallery
- Right-clicking the ViewController project
- Double-clicking an unimplemented view in a bounded task flow that uses page fragments



To create a page fragment:

1. In the Applications window, right-click the project where you want to create and store page fragments (typically ViewController). Select New > ADF Page Fragment to open the Create ADF Page Fragment Wizard.
2. Enter a name for the page fragment file. By default, JDeveloper uses `.jsff` for the source file extension.
3. Accept the default directory for the page fragment or choose a new location. By default, JDeveloper saves page fragments in the project's `/public_html` directory in the file system. You might change the directory so that it groups fragments according to the bounded task flows in which they are used or the functions that they perform (for example, the default directory `/public_html/orders`).
4. (Optional) Select a template to base the fragment on a template.

Another way to create a page fragment is to drag a view to a bounded task flow that uses page fragments, and then double-click that view to create the page fragment.

When you are finished, JDeveloper displays the page fragment file in the visual editor. You can drag components from the Components window to the page fragment. You can use any ADF Faces or standard JSF component.

Using a Page Fragment on a Page

Use a page fragment on a page by:

- Inserting a bounded task flow with page fragments as a region on your page
  - The page fragment has its own binding context.
- Using the af:declarativeComponent tag to import the fragment
- Do *not* use the jsp:include component.
  - Cannot use partial-page refresh
  - Cannot pass parameters using EL
  - Is not customizable with MDS

Modifying the page fragment affects all pages that use it (but check the overall layout of consuming pages).

When you build a JSF page, the page can use one or more page fragments that define different portions of the page. The same page fragment might be used more than once on a page, or (more typically) the same page fragment might be used on multiple pages.

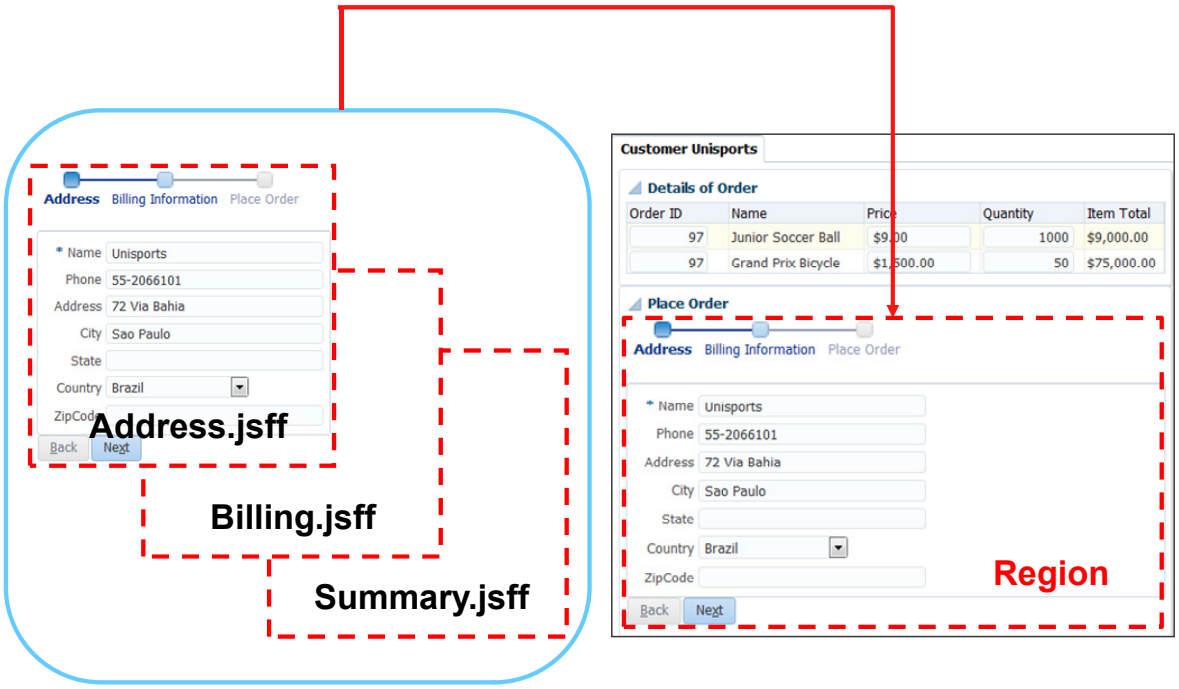
In most cases, a page fragment is inserted on a page as part of a region, meaning that the developer drags a bounded task flow to an area of a page and defines the task flow to be represented by an ADF region on the page; the page fragments defined in the task flow are then displayed on the page as necessary.

Occasionally page fragments are used to implement a small piece of UI that is not part of a bounded task flow. In this case, avoid using the `jsp:include` component to import the fragment. Doing so is an anti-pattern, causes inconsistent results, and has limited functionality. The reason is that JSF parameters are not supported by the `jsp:param` element that was historically used with the `jsp:include` tag, and anything within a `jsp:include` is not customizable using metadata services (MDS). Instead, use the `af:declarativeComponent` tag to import the fragment into the page.

Note: This component is part of the ADF Faces declarative component library, and its use is outside the scope of this course. For more information, see the ADF Faces tag documentation or the *Web User Interface Developer's Guide for ADF*.

When you modify a page fragment, the pages that consume the page fragment automatically display the modifications. When you make changes to a fragment (especially layout changes), you should check that the overall layout of pages that consume the page fragments still meets the requirements.

Regions: Review



As you learned in a previous lesson, an ADF region is a JSF component that represents a task flow as a smaller part of a larger page. A region can be used on a page or page fragment. You use a region to wrap a task flow for display on a page that has other content. Regions are therefore a primary implementation of reuse in ADF applications. They encourage the reuse of fragments in various task flows.

ADF Libraries

An ADF Library enables you to reuse components of an ADF application. The following types of components can be packaged in an ADF Library:

- Data controls
- Business components
- Task flows
- Task flow templates
- Page templates
- Declarative components
- Skins

Reusing Components

In the course of application development, certain components are often used more than once. Whether the reuse happens within the same application, or across different applications, it is often advantageous to package these reusable components into a library that can be shared by different developers, across different teams, and even across organizations.

ADF Libraries provide a convenient and practical way to create, deploy, and reuse high-level components. You should design your application with component reusability in mind. If you create components that can be reused, you package them into ADF Library JAR files and add them to a reusable component repository. If other developers need a component, they can look in the repository for that component and then add the library to their project or application.

For example, you can create entity objects, view objects, and an application module for a particular domain and package it as an ADF Library to be used as the model project in several different applications. Or, if your application consumes ADF page templates, you may be able to load a page template component from an ADF Library to create applications with a common look and feel. You can also construct a page flow by stringing several bounded task flows pulled from an ADF Library.

Note that ADF Library JAR files are not ordinary JAR files. They include additional metadata that is required by JDeveloper.

Reusable Components Supported by ADF

- **Data Controls:** Any data control can be packaged into an ADF Library JAR.
- **Application Modules:** When you package an application module data control, you also package the business components that are associated with that application module.
- **Business Components:** Business components are the entity objects, view objects, and associations used in the business services layer; you can package them by themselves or together with an application module.
- **Task Flows and Task Flow templates:** Task flows can be packaged into an ADF Library JAR for reuse. ADF bounded task flows that are built using pages can be dropped on pages, creating a link to call the bounded task flow. If an ADF task flow template was created in the same project as the task flow, the ADF task flow template is included in the ADF Library JAR and is reusable.
- **Page Templates:** You can package a page template and its artifacts into an ADF Library JAR. If the template uses image files and they are included in a directory in your project, these files are also available for the template during reuse.
- **Declarative Components:** You can create declarative components and package them for reuse. The tag libraries that are associated with the component are included and loaded into the consuming project.

Packaging Reusable Components into Libraries

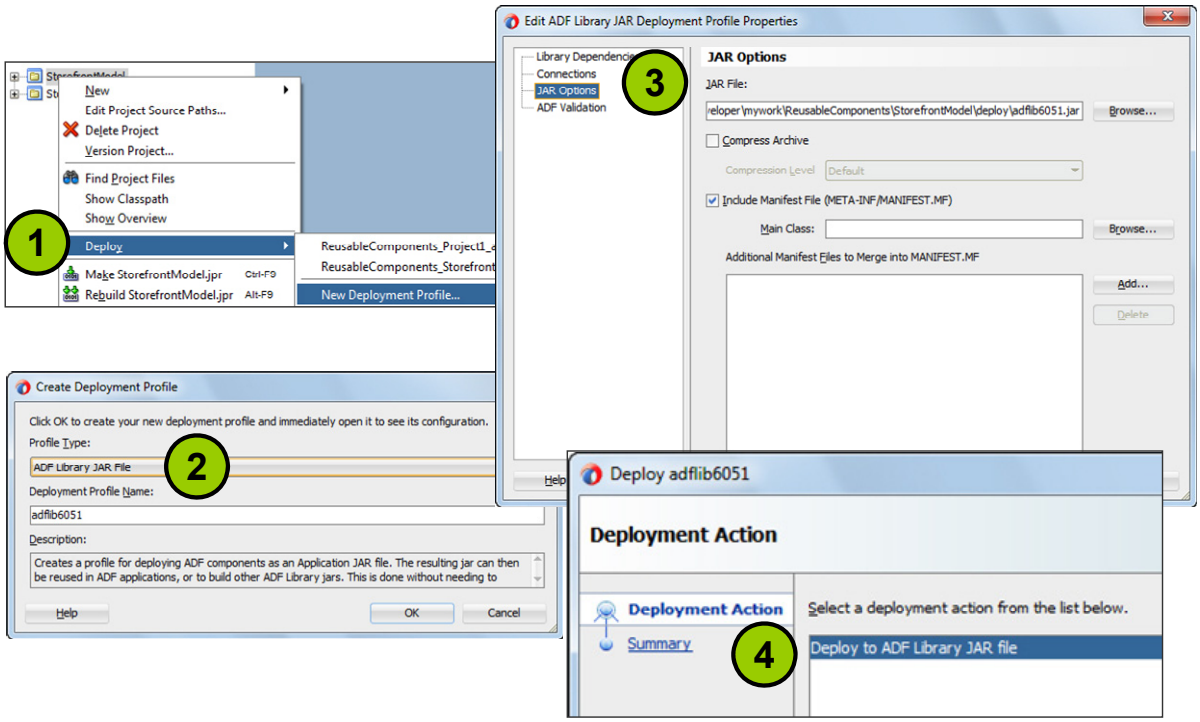
General steps to use ADF libraries:

1. Create a project that contains the reusable code.
2. Define the ADF Library JAR deployment profile.
3. Package components into ADF Library JAR files.
   - Basically a standard JAR file with some additional metadata
4. Publish the JAR files to a shared location.
5. Add one or more libraries to consuming projects to use its components.
   - A project may consume more than one ADF Library.
6. Republish the ADF Library as necessary.

An ADF Library is created from a project containing one or more reusable components. It is essentially a standard Java archive (JAR) file with some additional metadata, and it is created for you by the ADF Library deployment profile. After the library is created, development teams should publish it to a file system so that other developers can access it. When consuming developers are ready to access the components in a library, they can add it to their project so that the components can be used.

A project can consume more than one ADF Library. For example, a page template from one library and a bounded task flow from another library might both be added to a single ViewController project.

Creating an ADF Library

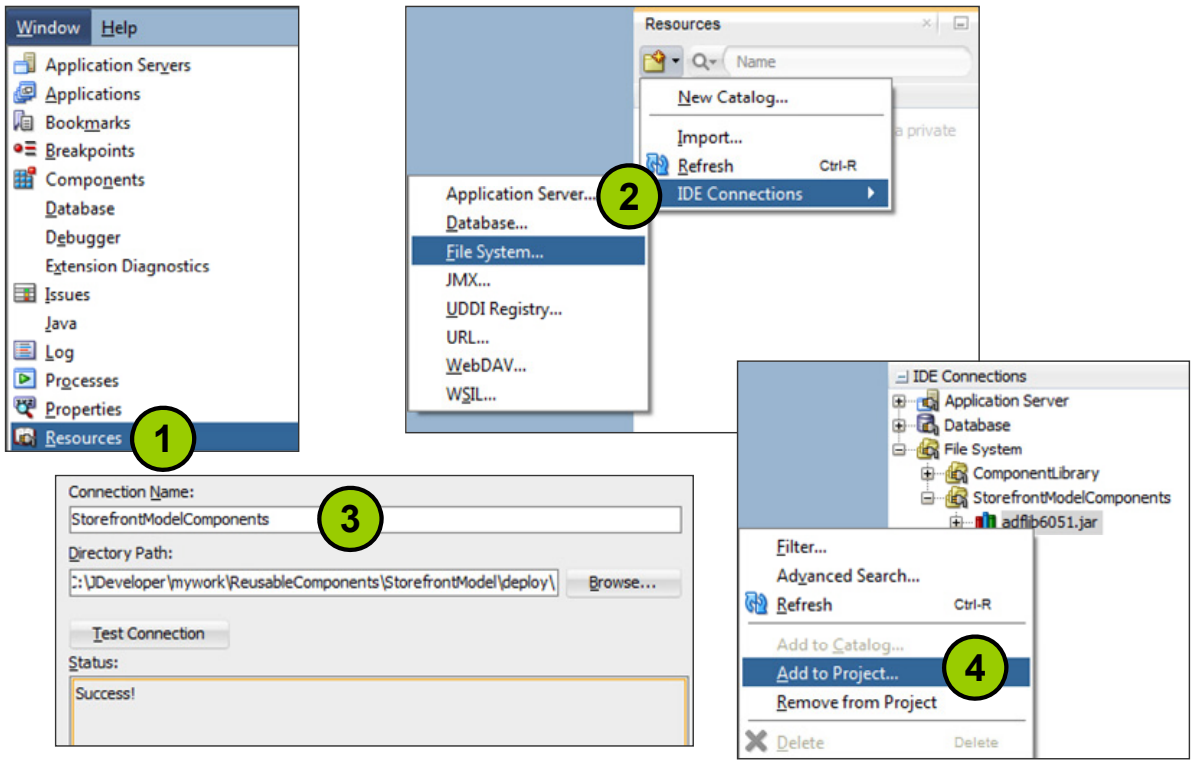


Each project can have one ADF Library JAR. If you create multiple projects and want to reuse components from each of the projects, you must create an ADF Library JAR for each project. Do not be tempted to combine multiple components such as business components, task flows, and page templates all in one project to create one ADF Library JAR. Doing so is an anti-pattern; multiple libraries (and therefore projects) should be used instead.

To package and deploy a project into the ADF Library JAR:

1. In the Applications window, right-click the project that contains reusable components and choose Deploy > New Deployment Profile.
2. In the Create Deployment Profile dialog box, select ADF Library JAR file from the Profile Type list and enter a name for the deployment profile. Click OK.
3. Edit the profile that you just created. In the ADF Library JAR Deployment Profile Properties dialog box, verify the default directory path or enter a new path to store your ADF Library JAR file. Click OK, and then click OK again.
4. To deploy the project as an ADF Library JAR file, right-click the project in the Applications window and select Deploy > *profile* > to ADF Library (where *profile* is the name of the deployment profile). This creates the JAR file in the deployment directory of the project (by default named `deploy`). Developers can then copy the JAR file to a source control system or other shared resource for shared access.

Adding an ADF Library to a Project



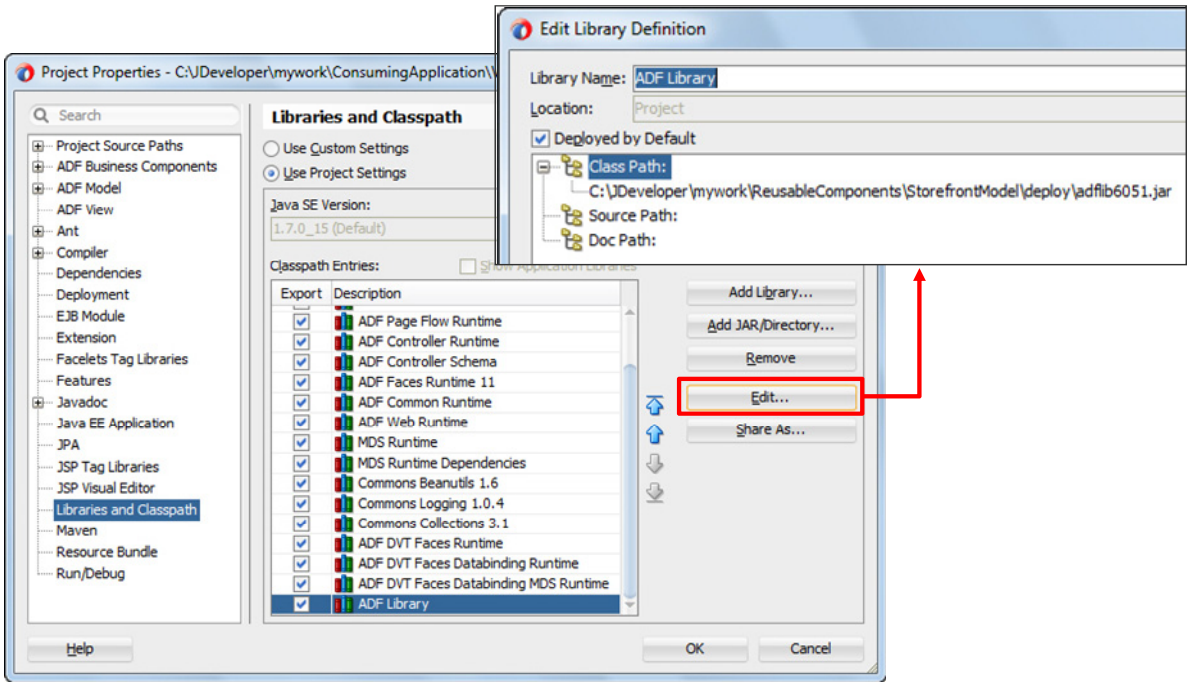
Using the JDeveloper Resources window is the easiest and most efficient way to add library resources to a project.

To use the Resources window to add the component to the project:

1. Select Resources from the Window menu.
2. In the Resources window, click the new folder icon and choose IDE Connections > File System from the menu.
3. In the Create File System Connection dialog box, enter a name for the Connection ID and then enter the path of the JAR. Click Test Connection and then click OK if the connection is successful.
4. The new ADF Library JAR appears under the connection name in the Resources window. With the consuming project open and selected in the Applications window, right-click the JAR (or any component in the JAR subdirectory) and select Add to Project. The JAR is added to the class path, and all its supported components are added to the current project.

You cannot choose to add only one component in a multicomponent ADF Library JAR. However, for application modules and data controls, you have the option to drag a particular application module or data control from the Resources window to the Data Controls window.

Viewing the ADF Libraries That Are Applied to a Project



Project Properties

To view the ADF libraries that are already applied to a project, right-click the project in the Applications window and select Project Properties (or double-click the project to open the project properties). Under “Libraries and Classpath,” select ADF Library and click Edit to see the ADF Library JAR files that are available.

Guidelines for Using ADF Libraries

- Adhere to strong naming conventions.
  - Avoid naming conflicts.
  - Goal: To facilitate identifying component functionality
- Standardize storage by agreeing on:
  - Type of repository needed
  - Storage and organization
  - Access methods
  - Schedule for uptake

Designing for Reuse

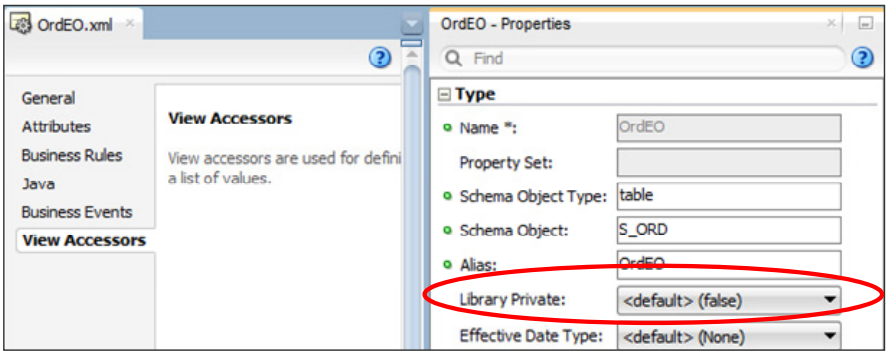
Creating and consuming reusable components should be included in the early design and architectural phases of software projects. You and your development team should consider which components are candidates for reuse, not only for current applications but also for future applications and for applications being developed in other departments.

When you create reusable components, you should try to create unique and relevant names for the application, project, application module, task flow, or any other relevant file or component. Do not accept the JDeveloper wizard default names, but try to create unique names to avoid name conflicts with other projects or components in the application. You should also consider creating standardized naming conventions so that both creators and consumers of ADF Library JARs can readily identify the component functionality.

You and your team should decide on the type of repository needed to store the library JARs, where to store them, and how to access them. You should consider how to organize and group the library JARs in a structure that fits your organizational needs.

Restricting Business Component Visibility in Libraries

Set the `Library Private` property for a business component to `true` if you do not want consumers to see the object in the library JAR.



When deploying a library for reuse, there may be some business components that are not intended for reuse but must be included in the library because other components depend on them.

Examples:

- The library developer wants consuming projects to access components in the library only through services exposed on the application module. All components in the library are private and may not be reused.
- The library developer wants consuming projects to access an entity object only by using or extending view objects in the library; the entity object contains attributes that should not be exposed in any view object. In this case, the entity object is private and the view objects are public.

You can mark an ADF BC object as public (the default) or private. Setting Library Private to `true` keeps the object from appearing to the consumer who is browsing the library JAR.

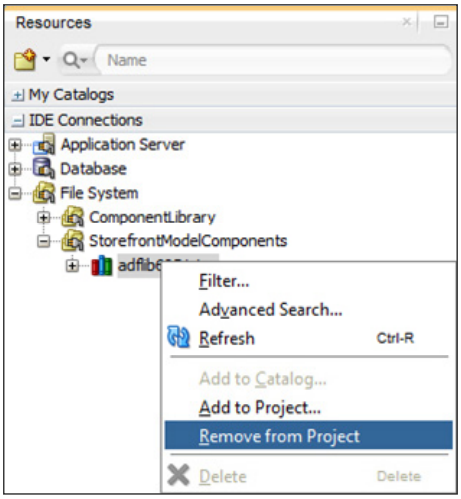
To mark a BC object as private:

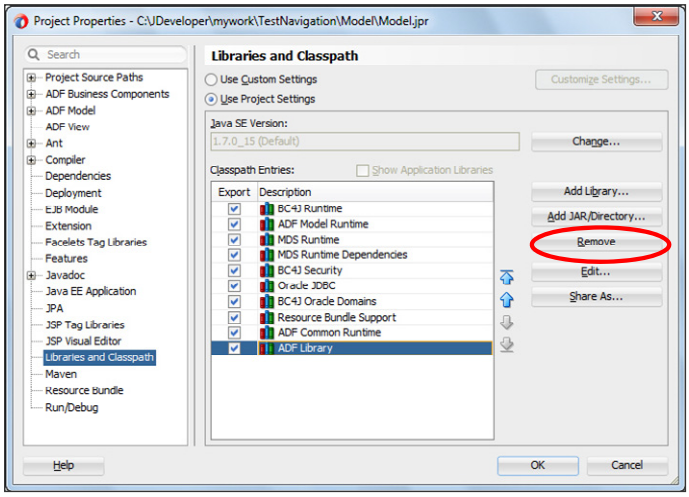
1. Open the object in the editor.
2. In the Property Inspector, click the Type tab and set Library Private to `true`.

When an object is marked as private, it is not visible in the Resource Catalog or in the Application Navigator of the consuming project.

Removing an ADF Library from a Project

Resources window **OR** **Project Properties**





You can remove an ADF Library JAR only if there are no dependencies from any components to the ADF Library components. When you remove a JAR, it is no longer in the project class path and all its components are no longer available for use.

There are two ways to remove an ADF Library from a project:

- By using the Resources window: Right-click the JAR in the Resources window and select “Remove from Project.”
- By manually removing the JAR using Project Properties:
  1. In the Applications window, double-click the project.
  2. In the Project Properties dialog box, select “Libraries and Classpath” in the left pane.
  3. In the “Libraries and Classpath” list, select ADF Library. Then click Edit.
  4. In the Edit Library Definition window, select the ADF Library JAR that you want to remove under the Class Path node. Then click Remove.
  5. Click OK to accept the deletion, and click OK again to exit the Project Properties window.

After you have deleted all ADF Library JAR files from the project, an ADF Library placeholder icon may still be present in the Project Properties Libraries window, as shown in the slide. You do not need to remove this icon.

Summary

In this lesson, you should have learned how to:

- Create and use page templates
- Create and use page fragments
- Describe the functionality for packaging reusable components into libraries

Practice 15 Overview: Building Reusability into Pages

This practice covers creating a page template for the main page and for another page.

16

Achieving the Required Layout

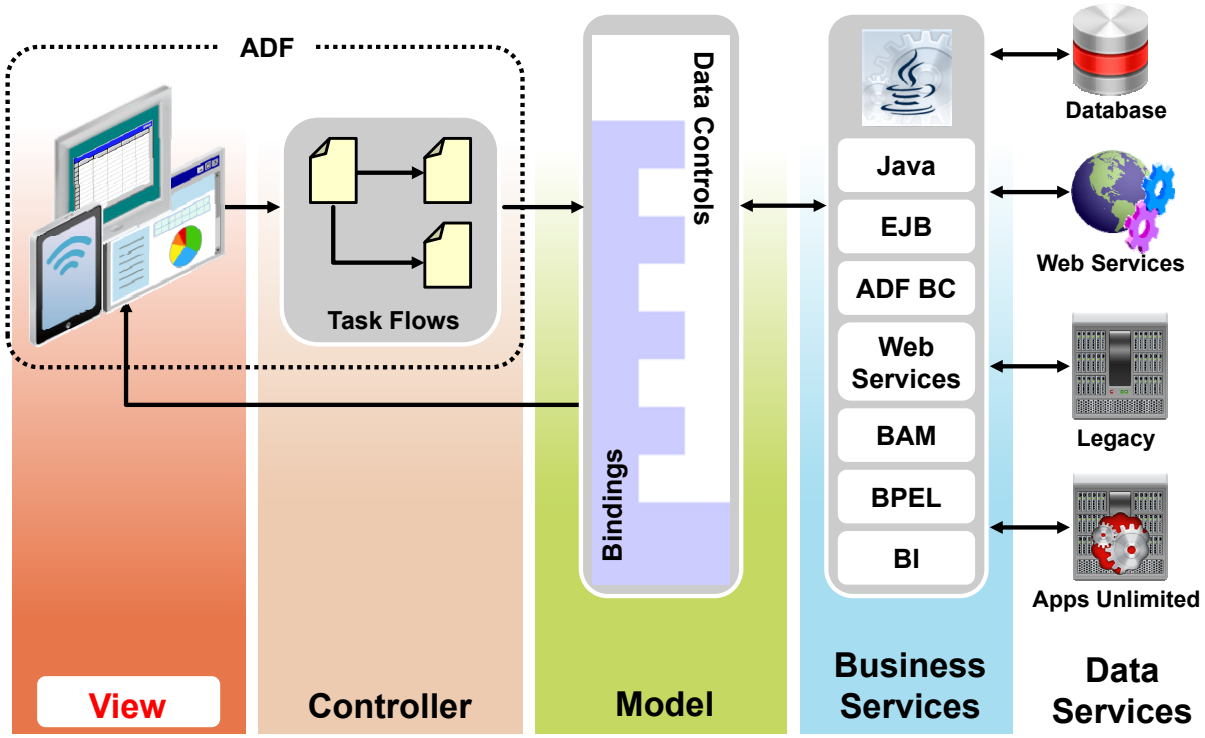
Objectives

After completing this lesson, you should be able to:

- Define and use component facets
- Explain what stretch and flow components are, and describe how to use them effectively
- Define and use complex layout components
- Define and use dynamic page layout
- Describe how to use skinning to change the appearance of an ADF application

ORACLE

Achieving the Required Layout



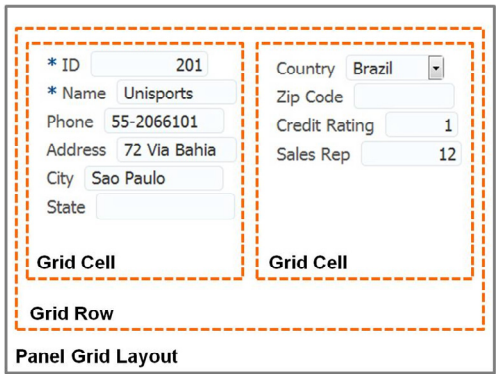
In an earlier lesson, you learned how to use ADF Faces components to add functionality to the application.

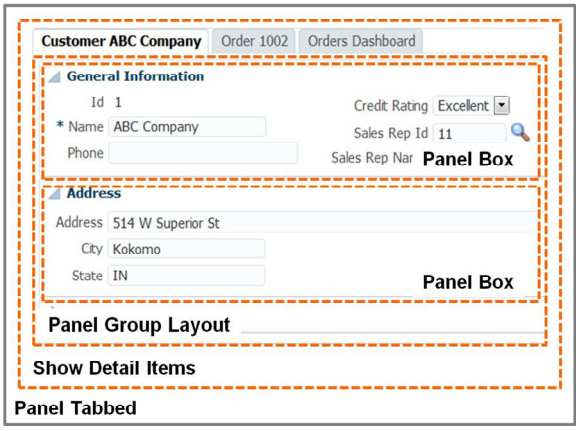
One of the goals in web development is to design attractive and usable pages. In this lesson, you learn how to use complex layout components and styles to achieve the required page appearance. You also learn how to use dynamic page layout techniques.

Overview of ADF Faces Layout Components

Use layout components to arrange other components on the page:

- Containers
- Interactive containers





- Components that add blank lines and spaces

ADF Faces provides layout components that can be used to arrange other components on a page. Usually, you begin building your page with these components. You then add components that provide other functionality (for example, data-bound components, buttons, menus, and so forth) either inside facets or as child components to these layout components.

In addition to layout components that simply act as containers, ADF Faces provides interactive layout components that can display or hide their content, and layout components that provide sections, lists, or empty space. Some layout components also provide geometry management functionality, such as stretching their content to fit the browser window when the window is resized, or the capability to be stretched when placed inside a component that stretches. Understanding the geometry management functionality of layout components is key to building attractive and usable pages.

The slide shows some commonly used containers and interactive containers. You learn more about these components in the following slides.

Layout Containers

Commonly used ADF Faces layout containers:


Decorative Box: Stretches child in center facet


Panel Border Layout: Lays out children consecutively


Panel Collection: Aggregates collection components


Panel Grid Layout: Lays out children in a grid


Panel Form Layout: Organizes children in a form


Panel Stretch Layout: Stretches child in center facet

ORACLE

- **Decorative Box:** Includes borders; stretches the child in the center facet to fill the available space
- **Panel Border Layout:** Lays out all children consecutively in the middle
- **Panel Collection:** Aggregates collection components like table, tree table, and tree to display menus, toolbars, and status bar items
- **Panel Grid Layout:** Lays out components within a grid made of rows and cells
- **Panel Form Layout:** Organizes components in a form
- **Panel Stretch Layout:** Stretches the child in the center facet to fill available space.

Interactive Layout Containers

Commonly used ADF Faces interactive layout containers:


Panel Accordion: Displays expandable panes vertically


Panel Box: Displays box with show and hide capability


Panel Dashboard: Arranges boxes in columns and rows


Panel Header: Provides header and an optional icon


Panel Tabbed: Displays a set of tabbed panels


Panel Splitter: Displays two panes divided by a splitter

ORACLE

- **Panel Accordion:** Displays collapsible and expandable panels within a vertical layout
- **Panel Box:** Displays a box with information that can be displayed or hidden below a header
- **Panel Dashboard:** Arranges components in rows and columns (similar to the Panel Form Layout component). However, instead of containing text components, the children of Panel Dashboard components are Panel Box components that contain content.
- **Panel Header:** Provides header type functionality, such as message display or associated help topics
- **Panel Tabbed:** Displays a set of tabbed panels
- **Panel Splitter:** Displays two panes (either horizontally or vertically) that are separated by a repositionable splitter

Geometry Management of Layout Containers

Layout Component	Stretchable by Parent	Stretch Its Children
Decorative Box	Yes	Yes
Panel Accordion	Yes	Yes
Panel Border Layout	No	No
Panel Box	Yes	Yes (if stretched by parent)
Panel Collection	Yes	Yes
Panel Form Layout	No	No
Panel Grid Layout	Yes	Yes
Panel Group Layout	Yes (if layout is scroll and vertical)	No

ORACLE

In modern web layouts, you might have some content on the page that needs to stretch to claim maximum space, some content that needs to scroll, and some content that must remain static when the user resizes the browser. ADF Faces layout components include geometry management capabilities to ensure the following:

- Components respond as expected when the browser is resized.
- Components fill the space available in a specific area of a view.
- Stretching behavior is handled based on parent or child component settings.

To work with layout components effectively, you need to understand which layout components are stretchable by a parent component and which layout components stretch their children components.

Geometry Management of Layout Containers

Layout Component	Stretchable by Parent	Stretch Its Children
Panel Dashboard	Yes	Yes
Panel Drawer	Yes	Yes
Panel Header	Yes	Yes (if stretched by parent)
Panel Springboard	Yes	Yes
Panel Splitter	Yes	Yes
Panel Stretch Layout	Yes	Yes
Panel Tabbed	Yes	Yes

This slide is continued from the previous slide.

Creating Layouts

1. Create a stretchable outer frame:
   - Use components that support being stretched and that also stretch their children (such as Panel Splitter, Panel Stretch layout, or Panel Grid Layout).
   - Use page templates and regions for reuse.
2. Create islands of flowing (non-stretched) components:
   - To transition from stretching to flowing, use Panel Group Layout with `layout="scroll"`.
   - Do *not* specify heights in percent units, use the position style, or stretch anything vertically inside flowing islands.
3. Customize the appearance of components:
   - Use themed components, custom skins, and style attributes.
   - Add scrolling, margins, borders, and padding.

Creating a Stretchable Outer Frame

To build attractive and usable pages, begin by creating a stretchable outer frame composed of components that support being stretched and that also stretch their children, such as Panel Splitter, Panel Stretch Layout, and Panel Grid Layout. You can determine whether a component supports these features by looking at the geometry management section of the tag documentation for the component that you want to use. To ensure maximum reuse and to centralize maintenance, you should consider using page templates and regions.

Creating Islands of Flowing Components

After creating the stretchable frame, create islands of flowing (nonstretched) components. To transition from stretching to flowing, you can use the Panel Group Layout component with the layout attribute set to `scroll`. For a consistent and reliable page layout, do not:

- Specify heights in percent units
- Use the position style
- Attempt to stretch anything vertically inside the flowing islands

The ADF Faces components in the following list cannot be reliably stretched (this list is not complete):

- Most input components
- Panel Border Layout
- Panel Form Layout
- Panel Group Layout with `layout="default"` or `layout="horizontal"`
- Panel Header with `type="flow"`
- Panel Label and Message
- Panel List

Customizing the Appearance of Components

After creating islands of flowing components, you customize the appearance of the components. Use themed Decorative Box components to organize your page layers with visual distinction and decorative borders. If an existing skin does not provide the visual distinction that you require, use a custom skin to modify the appearance of a component consistently across your user interface.

You can also use the `styleClass` attribute to customize appearance. Keep the corresponding style definitions in an easy-to-maintain location such as a custom skin, the `metaContainer` facet of the document component, or a style provided by the resource tag.

If all else fails, use component attributes such as `inlineStyle`, `contentStyle`, and `labelStyle`. These are less declarative and harder to maintain; they contribute more to the page's raw HTML size and may not even be needed if one or more of the above mechanisms are used. Styles are directly processed by the web browser, which gives you a great deal of power but at the cost of being less declarative and more error prone. The browsers do not support all styles on all elements, and certain combinations of styles produce unexpected results. Specifically, avoid specifying `inlineStyle` with values specified for height, width, top, bottom, left, right, or position. To specify width, use `styleClass="AFStretchWidth"` or `styleClass="AFAuxiliaryStretchWidth"` instead.

You can also customize the appearance by including scroll bars. You should have scrollbars only around flowing island content. The recommended transition component for switching from a stretching outer frame to a flowing island is the Panel Group Layout component with `layout="scroll"`. The browser determines whether scroll bars are needed and adds them automatically. You should avoid using nested Panel Group Layout components; otherwise, the user sees multiple scroll bars. In general, try to minimize the number of areas that require scrolling.

You can also add margins, borders, and padding, but it is more difficult than you might expect because of the browser's "CSS Box Model rules." You might need to use multiple components together to achieve the layout you require. For example, to add padding to a scrolling area, you can use a Panel Group Layout component with `layout="scroll"` to define the scrolling area. You can then nest another Panel Group Layout component inside with the `layout="vertical"` to add extra padding. In a stretching area, you may need to wrap a component inside a Panel Stretch Layout component with spacers in its top, start, end, and bottom facets for the padding.

Best Practices for Page Layout

To avoid development problems and reduce maintenance costs, use the following guidelines:

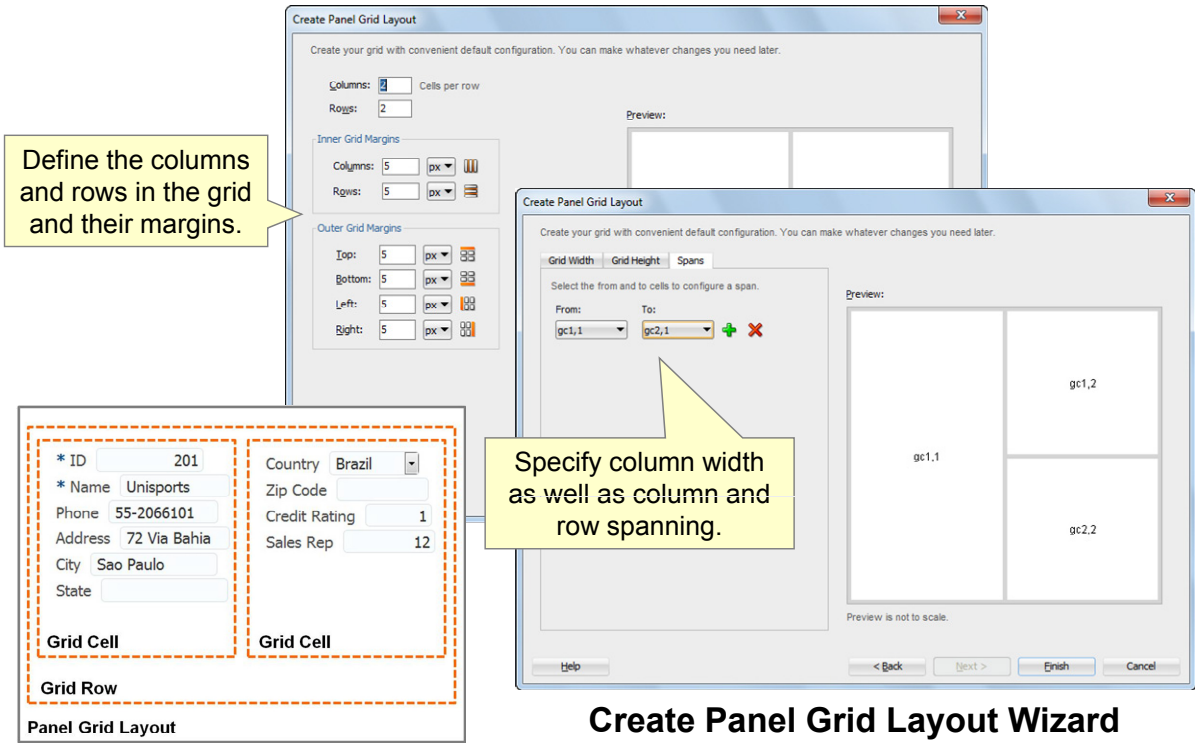
- Never stretch a component vertically inside a flowing container.
- Never specify a height value with percent units.
- Never use the "position" style.
- Take advantage of the layout pattern examples in the ADF Faces Rich Client Demo for guidance on creating layouts:
  - **Branding bar:** Logo, title, and global navigation links
  - **Form layout:** Organized labels and fields
  - **Stretched header:** Header with vertical stretchable body
  - **Tiled flowing:** Spaced titles that flow vertically
  - **Tiles stretching:** Spaced titles that stretch vertically

ORACLE

- Never try to stretch something vertically when inside a flowing (nonstretched) container. Attempting to do this will result in inconsistent behavior across web browsers.
- Never specify a height value with percent units. Instead, define stretching declaratively by following the process described in the previous slide.
- Never use the "position" style.
- Take advantage of the layout pattern examples in the ADF Rich Client Demo for guidance on how best to create a desired layout. Be sure to click the View Page Source and View Template Source links to see the tags and attributes that are used to produce the layout for each page.

In the slides that follow, you learn how to use some of the more commonly used layout components that are available in ADF Faces.

Laying Out Components in a Grid: Panel Grid Layout Component



Create Panel Grid Layout Wizard

To arrange components in a grid of rows and cells, use the Panel Grid Layout component, which provides the most flexibility of all the layout components while producing a fairly small amount of HTML elements. With it, you have full control over how each cell is aligned within its boundaries. The Panel Grid Layout component uses child Grid Row components to create rows and then, within those rows, uses Grid Cell components to create columns. You place other components in the Grid Cell components to display data, images, other content, or nested grid layouts. Each row defines the height and margins. Each cell defines width, margins, column span, row span, horizontal alignment, and vertical alignment, so there is no need to add spacers and separators to manage padding.

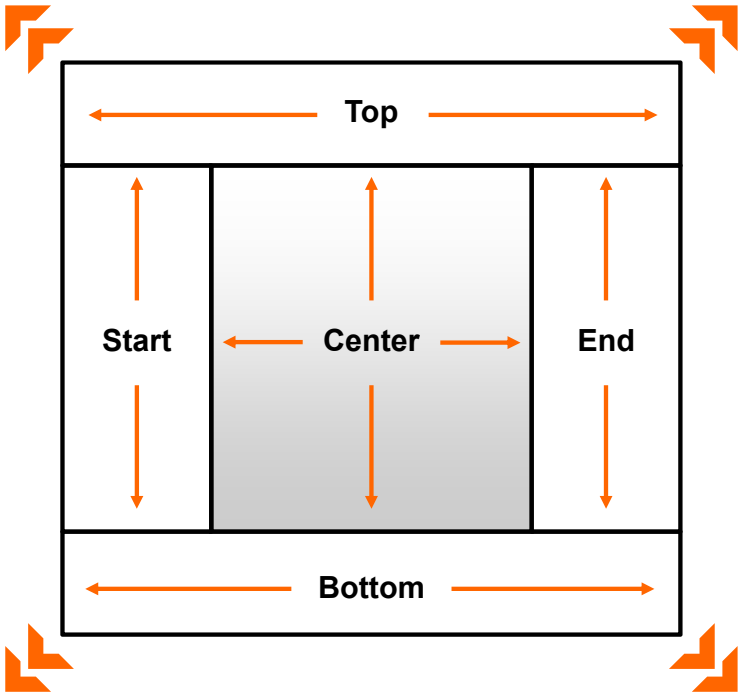
You can use the Create Panel Grid Layout Wizard in JDeveloper to configure grid properties such as:

- The number of columns and rows in the grid
- Inner and outer grid margins
- Column width (based on the widest cell in the column)
- Column and row spanning

When placed in a component that stretches it children, the Panel Grid Layout component stretches to fill its parent container by default. However, whether or not the content within the grid is stretched to fill the space is determined by settings for the Grid Row and Grid Cell components. By default, the child contents are not stretched.

You should avoid excessive nesting of Panel Grid Layout components. Because of how the component determines its dimensions, excessive nesting might result in rendering issues.

Laying Out Components to Stretch Across a Page: Panel Stretch Layout Component

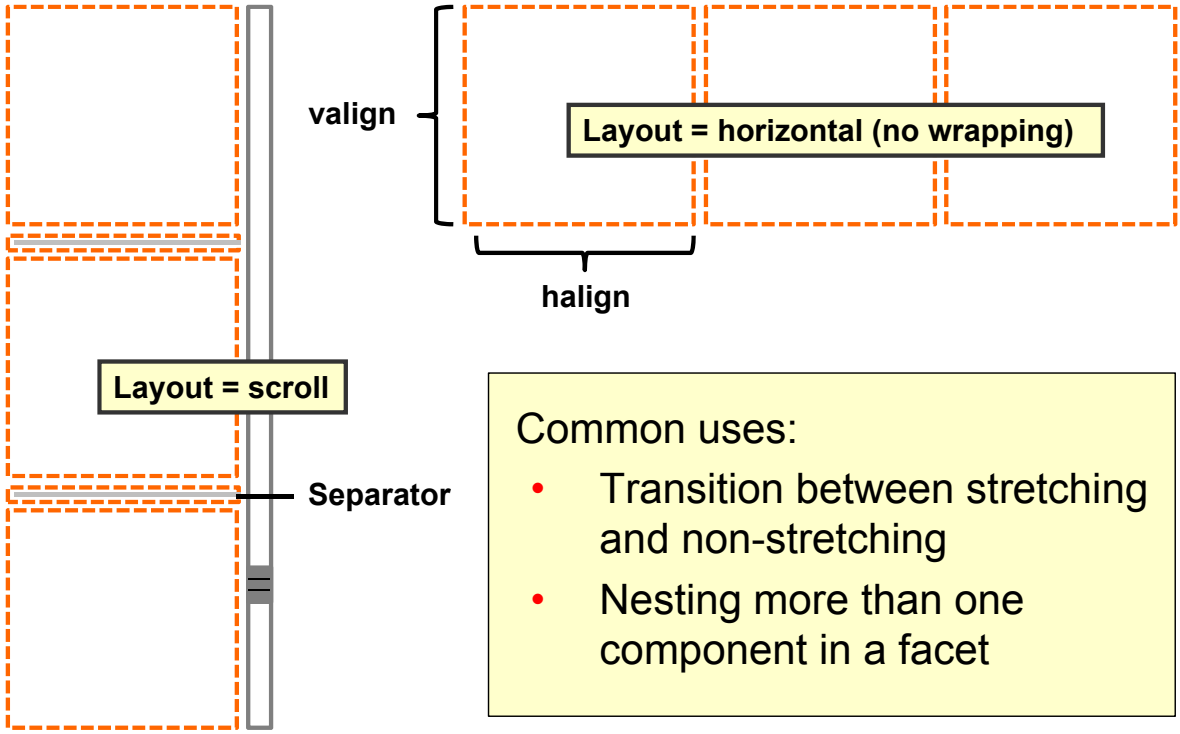


Use the Panel Stretch Layout component to arrange content in defined areas on a page and to enable content to stretch when the browser is resized. The Panel Stretch Layout component cannot have any direct child components. Instead, you place components within its facets: top, start, center, end, and bottom.

Panel Stretch Layout is one of the components that can stretch components placed in its facets. When you set the height of the top and bottom facets, any contained components are stretched to fit the height. Similarly, when you set the width of the start and end facets, any components contained in those facets are stretched to that width. If no components are placed in a facet, the facet does not render. That is, the facet will not take up any space. If you want the facet to take up the set space but remain blank, insert a spacer component. Components in the center facet are then stretched to fill up any remaining space.

Instead of specifying a value for the height of the top or bottom facet, or the width of the start or end facet, you can set the height or width to `auto`. This enables the facet to size itself to use exactly the space required by the child components of the facet. Space is allocated based on what the web browser determines is the required amount of space to display the facet content. Use the `auto` setting infrequently because it affects page performance.

Grouping Related Components: Panel Group Layout Component



Use the Panel Group Layout component to arrange a series of child components consecutively, vertically, or horizontally. The Panel Group Layout component does not stretch its children.

The default is to lay out child components consecutively with wrapping. At run time, when the contents exceed the available browser space (that is, when the child components are larger than the width of the parent container), the browser flows the contents onto the next line so that all child components are displayed.

You can set the `layout` attribute on the component to arrange child components horizontally or vertically:

- **`horizontal`:** Arranges child components in a horizontal line. No wrapping is provided when contents exceed the amount of available browser space. In a horizontal layout, you can align the nested components horizontally (by setting the `halign` property to `center`, `start`, `end`, `left`, or `right`), and you can align them vertically (by setting the `valign` property to `middle`, `top`, `bottom`, or `baseline`). By default, horizontally arranged child components are aligned in the center with reference to an imaginary horizontal line, and are aligned in the middle with reference to an imaginary vertical line.

- **`scroll`:** Arranges child components in a vertical layout in which child components are stacked vertically and a vertical scrollbar is provided when necessary. You do not have to write code to enable the scroll bars or set inline styles to control the overflow. For example, when you use the Panel Group Layout component with a Panel Splitter component that lets users display and hide child components, you do not have to write code to show the scroll bars when the contents are displayed or to hide the scroll bars when the contents are hidden. Simply wrap the contents to be displayed inside a Panel Group Layout component and set the layout attribute to `scroll`.
- **`vertical`:** Arranges child components in a vertical layout in which child components are stacked vertically

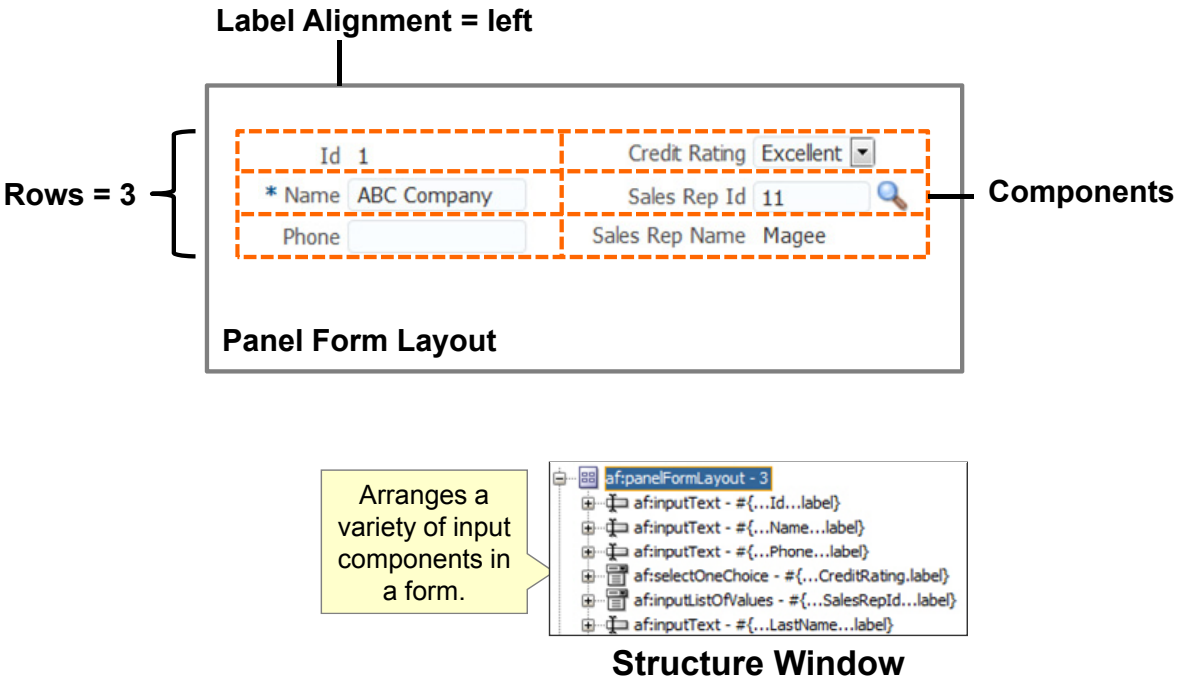
In all arrangements, each pair of adjacent child components can be separated by using a Separator component.

Unlike the Panel Splitter and Panel Stretch Layout components, the Panel Group Layout component does not stretch its child components. Therefore, the Panel Group Layout component (set to `scroll`) provides a good transition between stretched and flowing components. It supports being stretched but does not stretch its child components.

The Panel Group Layout component is also useful when you need to include more than one component in a facet. You can add the Panel Group Layout component to the facet and then add multiple children to the Panel Group Layout component.

Note: The Panel Group Layout component provides very little control over how individual children of the structure are presented. If you find yourself nesting multiple Panel Group Layout components, consider using a Panel Grid Layout component instead.

Laying Out Components in Forms: Panel Form Layout Component



Use the Panel Form Layout component to arrange multiple input components (such as input fields and selection list fields) in one or more columns. Each field in the form is a child component of the Panel Form Layout component. Notice that the form in the example is composed of a variety of input components.

To specify the number of rows in the form, set the value for the Rows property. If there are more child components than rows, the remaining child components are placed in a new column (assuming that a new column is allowed by the MaxColumns property). The default value for the MaxColumns property is `3`. If you have more rows than you can display in three columns, set the MaxColumns property accordingly. Otherwise, the MaxColumns setting is used, and the form may have more rows than you specified.

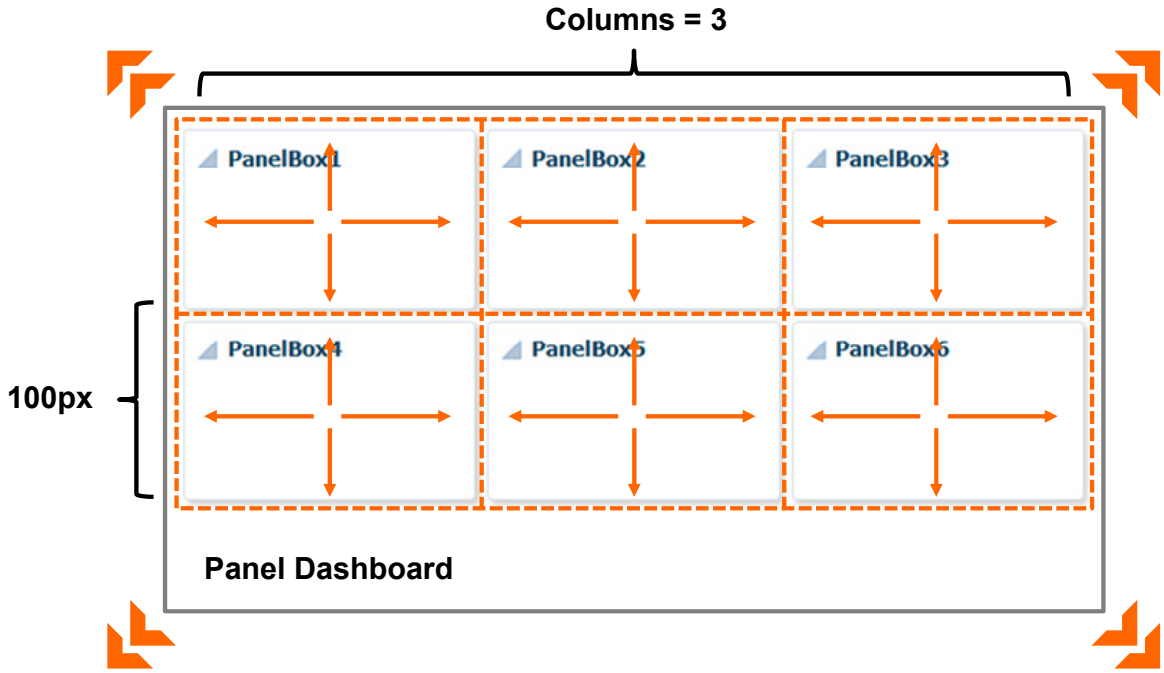
In the example, the Rows property is set to `3`. No value is specified for MaxColumns, so the form is arranged using three rows and two columns. You can also set the LabelAlignment property of the Panel Form Layout component to control where the label appears (either at the start of the field or on top of the field).

When you use a Panel Form Layout component to arrange components that have different heights, the components might not line up horizontally as you would like. If you want to line up the components more precisely, consider using a Panel Grid Layout component instead.

ADF Faces uses default label and field widths as determined by the standard HTML flow in the browser. You can also specify explicit widths to use for the labels and fields. Regardless of the number of columns in the form layout, the widths that you specify apply to all labels and fields. You specify the widths using either absolute numbers in pixels or percentage values. If the length of a label does not fit, the text is wrapped. If your page will be displayed in languages other than English, you should leave extra space in the labels to account for text expansion after translation.

Tip: If you use components other than input components (that is, components that do not have label attributes) or if you want to group several input components with one single label inside a Panel Form Layout component, you should first wrap the components inside a Panel Label Message component.

Laying Out Components in a Dashboard: Panel Dashboard Component



Valid child components: Panel Box and Region

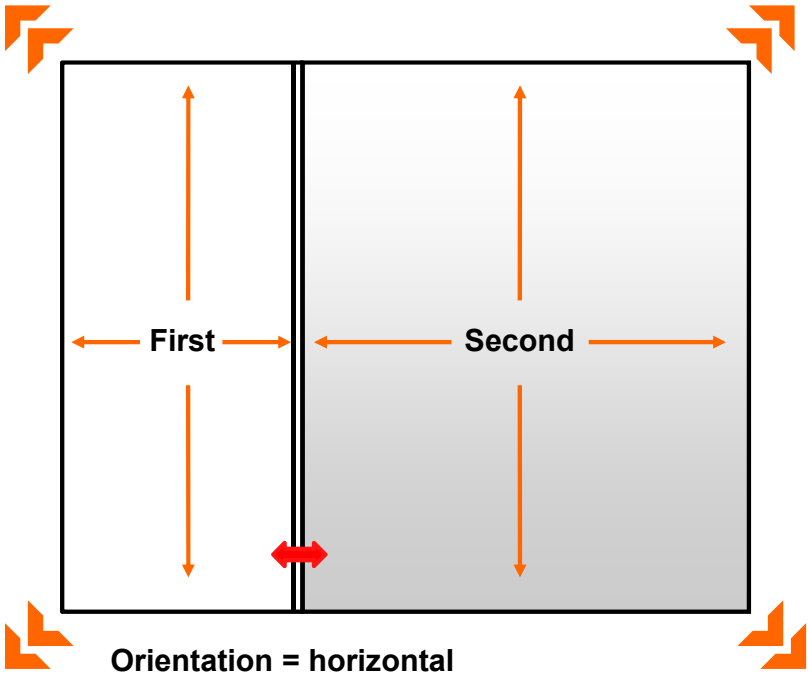
Use the Panel Dashboard component to arrange Panel Box components or ADF regions into a grid of rows and columns. The Panel Dashboard component stretches its children to fill the width of a column and the specified row height (specified in pixels in the RowHeight property). The child components will stretch to this height. To specify the number of columns, set the value for the Columns property.

If all of the child Panel Box components cannot fit within the dimensions of the Panel Dashboard, the Panel Dashboard provides a scroll bar.

When placed in a component that stretches its children, the Panel Dashboard by default stretches to fill its parent container, regardless of the number of children. This can result in blank space in the dashboard when the browser is resized to be much larger than the dashboard requires.

The Panel Dashboard component also supports declarative drag-and-drop behavior so that users can rearrange the child components.

Creating Resizable Panes: Panel Splitter Component

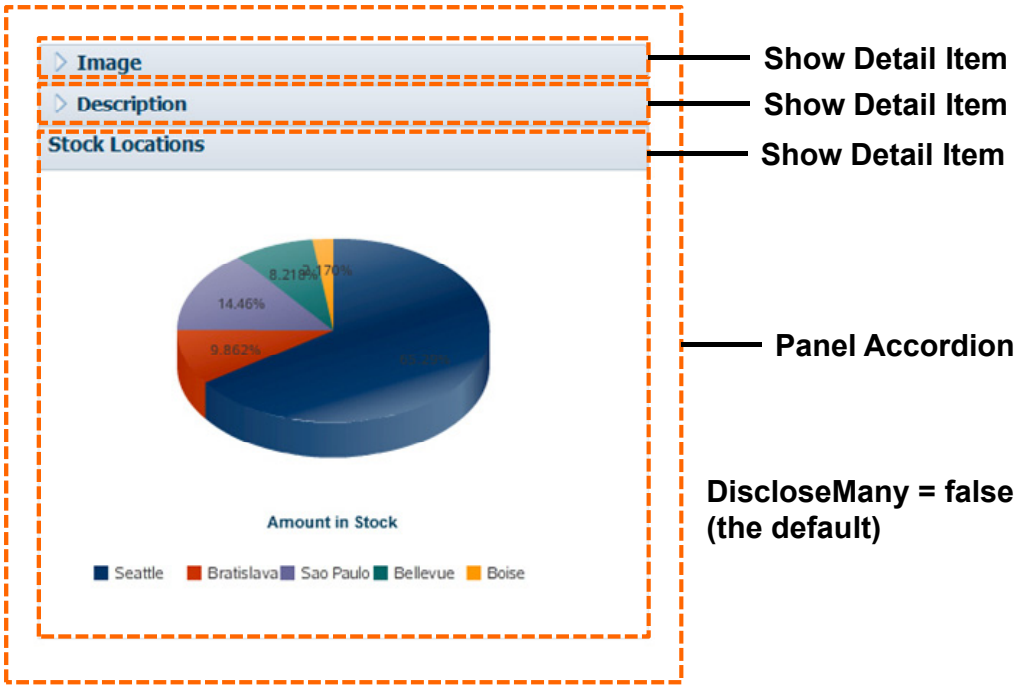


Use the Panel Splitter component to create resizable panes that stretch their content. The Panel Splitter component has two facets—first and second—which are separated by a repositionable splitter. You place components inside the facets of the Panel Splitter component. The Panel Splitter component uses geometry management to stretch its child components at run time. This means that when the user collapses one panel, the content in the other panel is explicitly resized to fill the available space.

Properties on the Panel Splitter component enable you to set the orientation of the splitter to either horizontal or vertical, to set the initial position of the splitter, and to make the splitter fixed (set the Disabled property to `true`).

You can nest Panel Splitter components to create a page with multiple resizable panes.

Displaying or Hiding Content in Panels: Panel Accordion Component



ORACLE

Use the Panel Accordion component with nested Show Detail Item components when you need to display multiple areas of content that can be expanded and collapsed. You can enable users to expand more than one panel at any time, or to expand only one panel at a time. When more than one panel is expanded, the user can adjust the height of the panel by dragging the header of the Show Detail Item component.

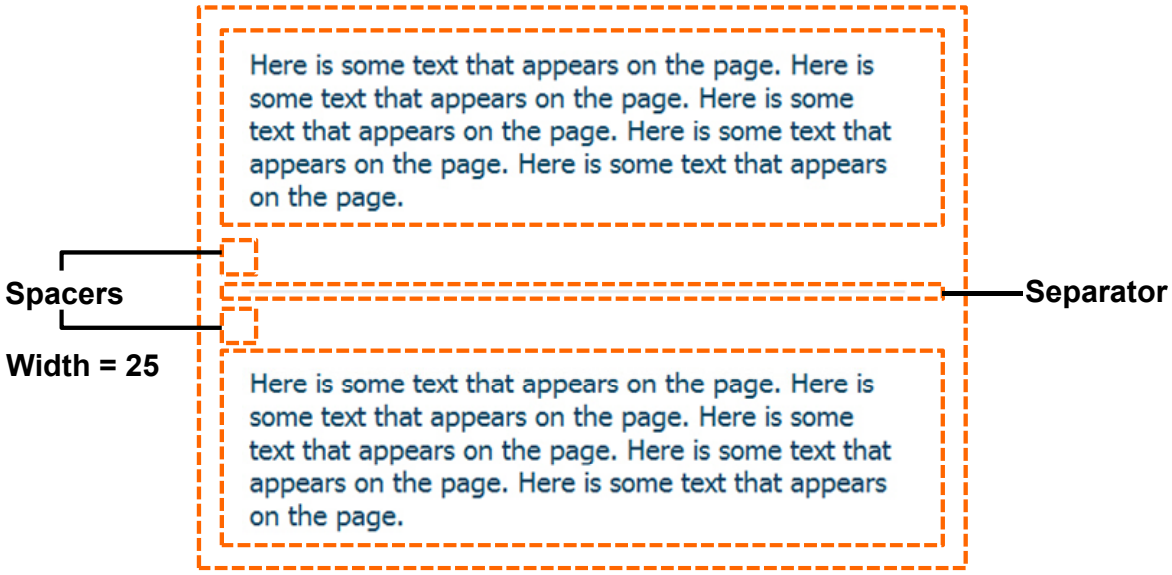
At run time, when the expanded panel content takes up too much space to display in the browser, ADF Faces automatically displays overflow icons that enable users to select and navigate to any panels that are out of view.

You can use more than one Panel Accordion component on a page (typically in different areas of the page) or nested. After adding the Panel Accordion component, insert a series of Show Detail Item components, with one for each panel. Then insert components into each Show Detail Item component to provide the panel content.

You can set additional properties on the Panel Accordion component to control whether users are able to expand the contents of more than one panel at a time (set the `DiscloseMany` property), and whether users are able to collapse all panels (set the `DiscloseNone` property). You can also configure the Panel Accordion so that the panels can be rearranged by dragging and dropping (set the `Reorder` property).

ADF Faces provides additional components that provide similar expandable and collapsible panels, including the Panel Tabbed, Panel Drawer, and Panel Springboard components.

Adding Blank Space and Lines to Layouts: Spacer and Separator Components



You have already learned that the Grid Layout Component enables you to specify padding to add blank space to a page. However, when you are using other types of layout components and want to add blank space and lines to the layout, you can use the Spacer and Separator components.

Spacer Component

Use the Spacer component when you want to space components so that the page appears less cluttered. You can include either (or both) vertical and horizontal space in a page by specifying values for the Height and Width properties.

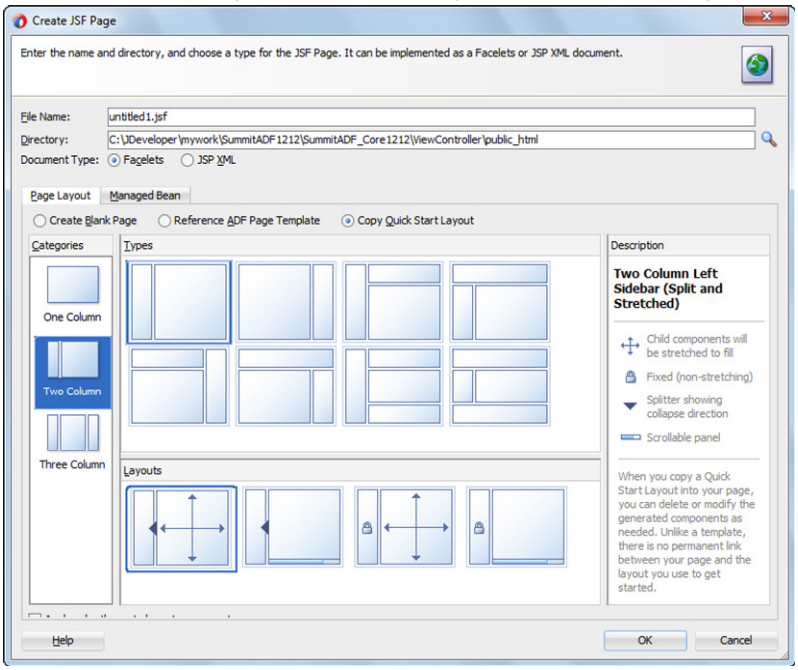
Separator Component

Use the Separator component to create a horizontal separator. This component can be stretched horizontally by a parent layout component that stretches its children (for example, a Panel Stretch Layout component).

Using Quick Start Layouts

Use quick start layouts to build layouts quickly and correctly.

Best Practice:
Use quick start layouts to create layouts that work correctly in all browsers.



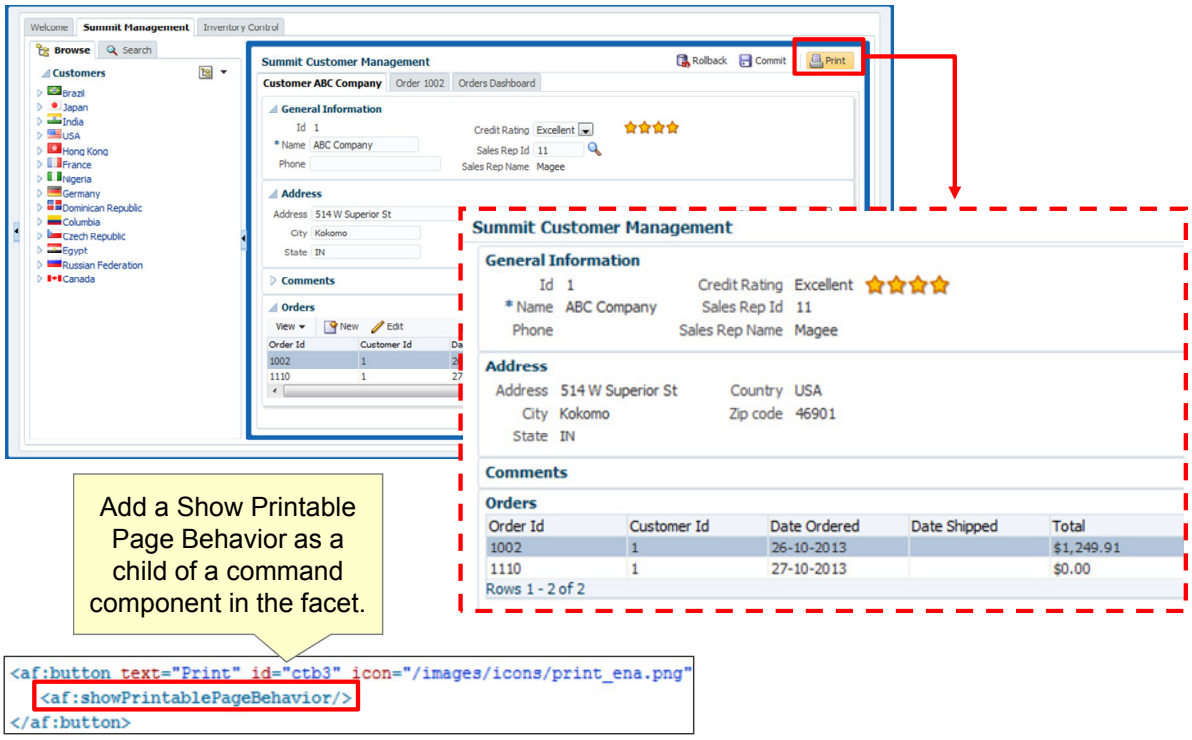
When you use the New Gallery Wizard to create a JSF page (or a page fragment), you can choose from a variety of predefined quick start layouts. When you choose one of these layouts, JDeveloper adds the necessary layout components and sets their attributes to achieve the look and behavior you want. In addition to saving time, using the quick start layouts ensures that layout components are used together correctly to achieve the desired geometry management.

You can choose from one, two, or three column layouts and then determine how you want the columns to behave. For example, you might want one column's width to be locked while another column stretches to fill available browser space. The slide shows the quick start layouts that are available for a two-column layout with the second column split between two panes. You can also choose to apply a theme to the selected quick start layout. When you select this option, additional color styling is added to some of the components used in the quick start layout.

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

When you want to use the same layout on many pages in your application, remember to define and use a page template (page templates are described in the lesson titled “Building Reusability into Pages”).

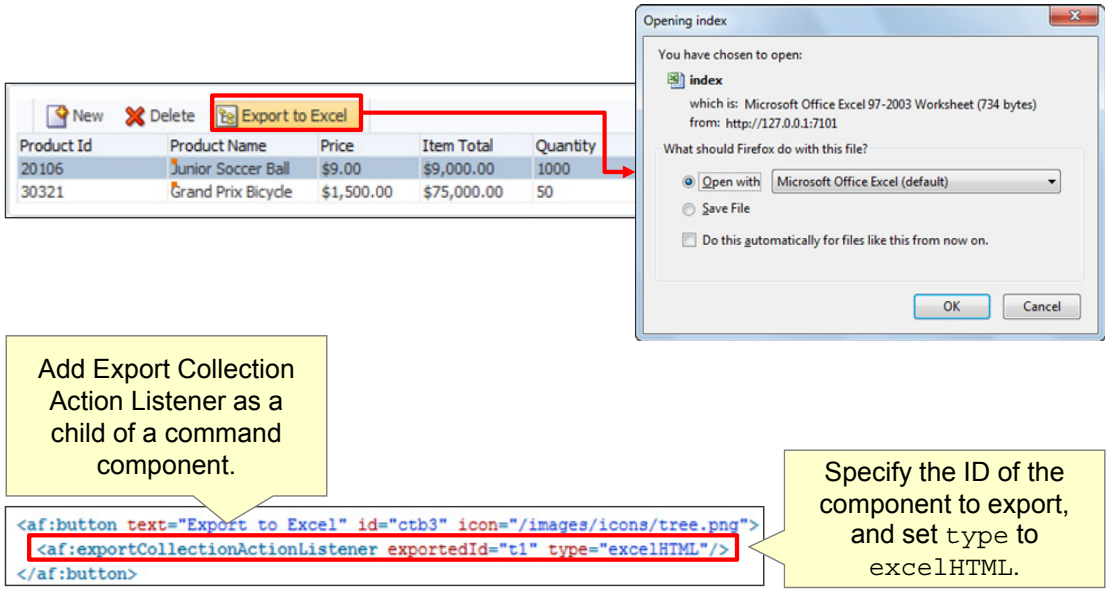
Adding a "Show Printable Page Behavior"



You can enable users to access a printable view of a facet in your layout by using the Show Printable Page Behavior. This behavior creates a printable page that turns off all stretching so that printed content is not truncated. To add the behavior to a facet, add a command component (for example, a button) to the facet that contains the content. Then insert the Show Printable Page Behavior (available under Operations in the Components window) as a child of the command component.

In the example in the slide, the Print button uses the Show Printable Page Behavior to display the facet that contains Summit Customer Management information.

Adding the Ability to Export to Excel



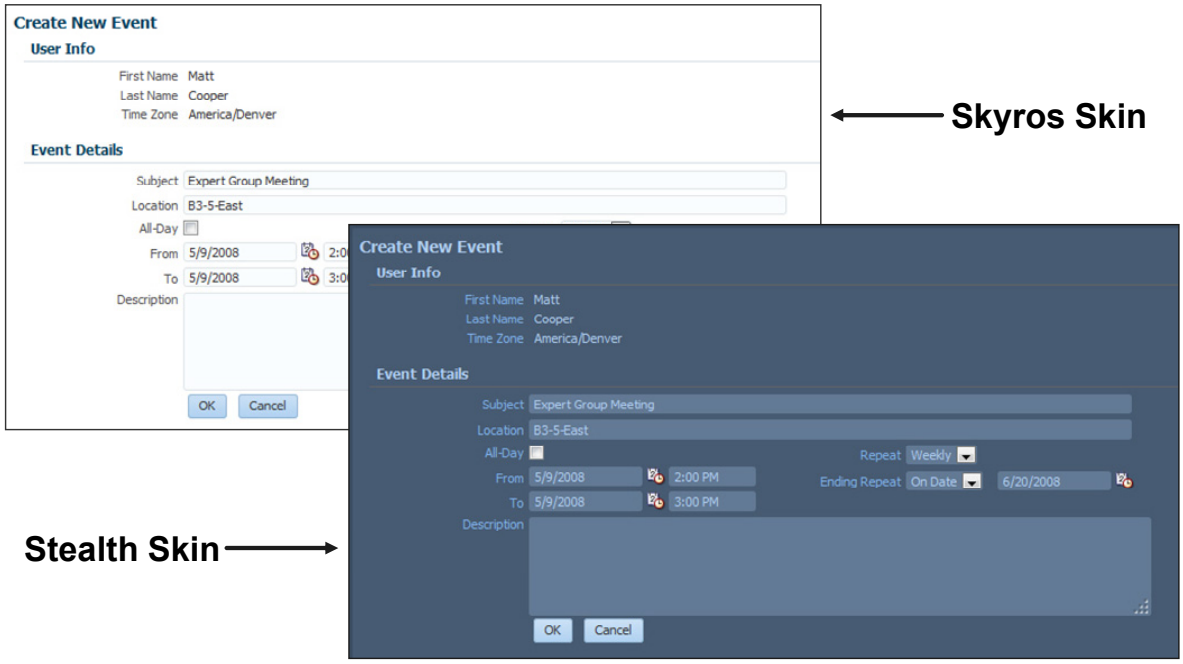
The Export Collection Action Listener tag is a declarative way to enable an action source, such as a command button or link, to export data from a collection model (such as a table, tree, or tree table).

To provide the ability to export to Excel, add a command component (for example, a button). Then insert the Export Collection Action Listener as a child of the command component. In the properties of the action listener component, you must specify the ID of the component that contains the content to export, and you must set the `type` property to `excelHTML`.

The “Export to Excel” button in the example in the slide exports the order details in the table to an Excel spreadsheet.

ADF Faces and Skinning

Use skinning to add a custom look and feel to your application.



Skinning is a facility in ADF Faces that enables developers to put a custom look and feel on top of the ADF Faces component set.

An ADF skin uses the format, properties, and selectors of cascading style sheets (CSS) to enable you to customize the appearance of ADF Faces components. Instead of providing a CSS file for each component or inserting a style sheet on each page of the application, you create one ADF skin for the web application. Every component that renders in the user interface automatically uses the styles defined by the ADF skin. This means you do not have to make design-time changes to individual pages to change their appearance when you use an ADF skin. Using an ADF skin also makes it easier for you to maintain a consistent appearance for all the pages that the application renders.

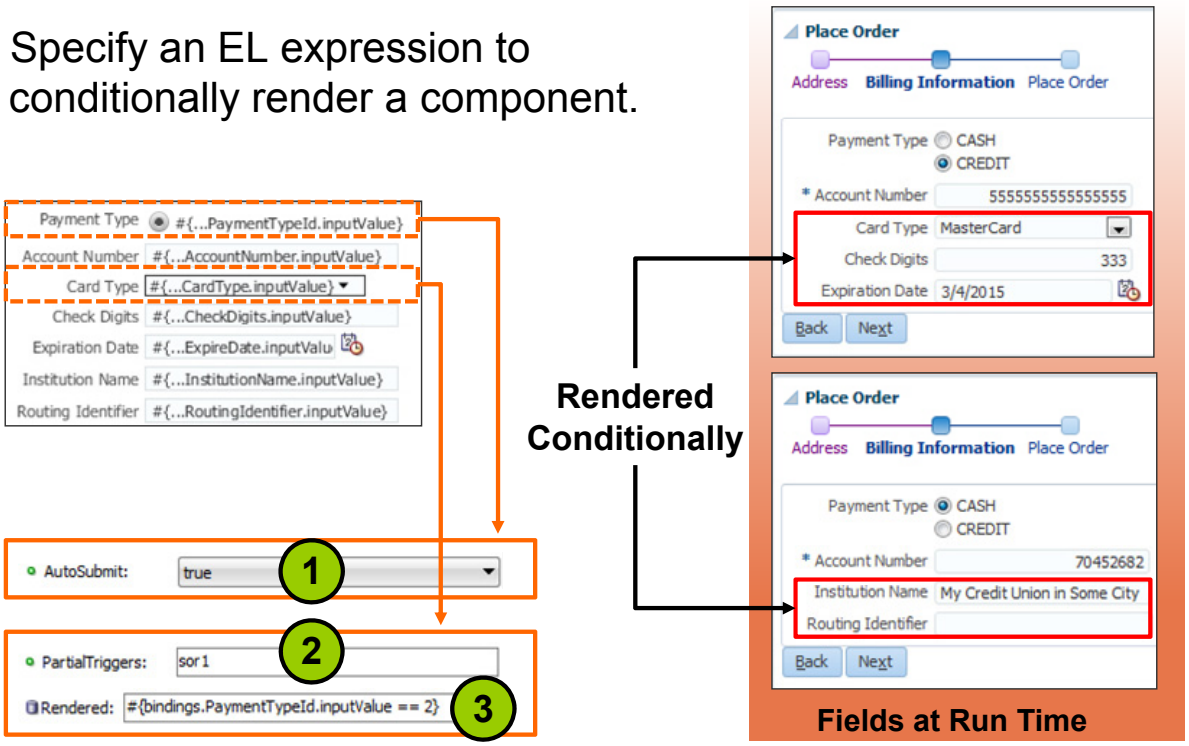
The slide shows two different skins applied to the same application: the Skyros skin and the Stealth skin.

Note: For more information about adding a custom skin to your application, see *Creating ADF Skins with Oracle ADF Skin Editor:*

http://docs.oracle.com/middleware/1212/skineditor/ADFSG/index.html

Using Expression Language to Display Components Conditionally

Specify an EL expression to conditionally render a component.



You can display a component conditionally by specifying an EL expression in the Rendered property of a component. If the expression evaluates to `true`, the component is rendered. If it evaluates to `false`, no output is delivered for the component.

The slide shows how to display a component based on the value selected in another component. The example shows a Billing Information page that contains radio buttons for choosing the payment type (CASH or CREDIT). If the payment type is CREDIT, you want users to see the Card Type, Check Digits, and Expiration Date fields. If the payment type is CASH, you want users to see the Institution Name and Routing Identifier. The slide shows how to set properties to conditionally render the Card Type field based on the selected payment type:

1. Select the Payment Type radio group, and set the AutoSubmit property to `true` so that the change is submitted when the user changes the selection.
2. Select the Card Type field, and set the PartialTriggers property to the ID of the component that should trigger a partial update. In the example, you specify the ID of the Payment Type radio group.
3. Also for the Card Type field, specify an EL expression that will determine whether the component is rendered. The expression in the example evaluates to `true` if the value of `PaymentTypeId` is `2` (the ID for the CREDIT option).

You use a similar process to conditionally render the other fields that are dependent on the payment type selection. At run time, the application displays the correct fields for each payment type.

Using the Switcher Component to Display Facets Conditionally

This lesson introduces one approach to displaying components conditionally to define dynamic layout. ADF Faces also provides the Switcher component when you want to render entire facets dynamically based on specific conditions.

For more information about the Switcher component, see *Developing Web User Interfaces with Oracle ADF Faces:*

http://docs.oracle.com/middleware/1212/adf/ADFUI/index.html

Refer also to the tag reference for the Switcher component at http://docs.oracle.com/middleware/1212/adf/TROAF/tagdoc/af_switcher.html.

Summary

In this lesson, you should have learned how to:

- Define and use component facets
- Explain what stretch and flow components are, and describe how to use them effectively
- Define and use complex layout components
- Define and use dynamic page layout
- Describe how to use skinning to change the appearance of an ADF application

Practice 16 Overview: Achieving the Required Layout

This practice covers the following topics:

- Examining layout issues in the course application
- Adjusting layout components that JDeveloper creates by default
- Creating new pages with required layouts

17

Debugging ADF Applications

Objectives

After completing this lesson, you should be able to:

- Define and use tools for logging and diagnostics
- Use the information in the logging window to better understand where a problem exists in the application
- Use the Oracle ADF Model Tester to debug business services
- Use JDeveloper to debug an application
- Use JUnit to help prevent regressions

Overview of Troubleshooting Tools

- JDeveloper message log
- ADF Model Tester
- Oracle ADF Logger
- JDeveloper Debugger
- JUnit and other testing frameworks
- Click history
- JDeveloper profiler
- JDeveloper code auditor
- WLS testing and logging tools
- JVM performance testing and logging tools

This lesson covers the first five tools that are listed in the slide. You learn how these tools can be used to troubleshoot an application.

Note: For more information about profiling and auditing applications, see *Developing Applications with Oracle JDeveloper:*

http://docs.oracle.com/middleware/1212/jdev/OJDUG/index.html

Troubleshooting Techniques

1. Identify the root error/exception in the message log.
2. Use the ADF Model Tester to narrow the issue.
3. Enable ADF logging (or add your own).
4. Use the debugger:
   - Step through application code.
   - Step through task flow phases.
   - Step through ADF source code.
5. Hypothesize a resolution:
   - Test code parameters/values in the debugger.
   - Test EL in the EL Evaluator.
6. Prevent future issues:
   - Create a regression test case.

ORACLE

The slide provides some general techniques to use when diagnosing issues. This is not an exhaustive list and may not be linear.

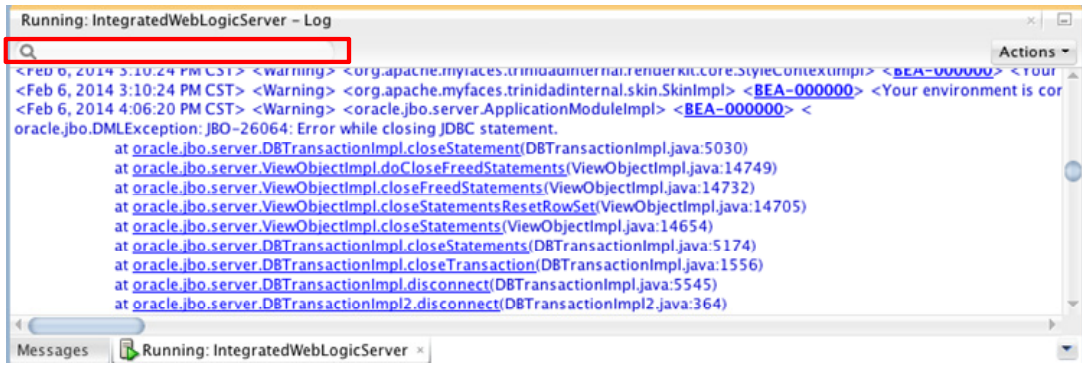
In some cases, identifying the root error in the message log (step 1) enables developers to completely understand the issue so that they can move immediately to testing a fix for the issue (step 5).

On the other hand, in some cases developers will reach step 5 and realize through testing that the error or issue is not what was originally hypothesized. In this case, more work must be done to understand the root cause of the issue by iteratively navigating through the earlier steps.

Reading the Message Log

The JDeveloper message log prints System.out messages and exceptions from the running JVM, including errors and messages from the ADF stack.

- Use the message log to identify the cause of an issue.
- All exceptions will be output. Scroll or search through downstream exceptions to identify the root exception.



When an error or exception is raised in an ADF application, the exception trace is printed in the message log. This is the first place developers should look to understand the root cause of the issue. You can usually click the error message to navigate directly to the problem line in the code. To ensure that the root cause of an issue is included in the message log window, Increase the size of the buffer in Tools > Preferences > Log > Maximum Log Lines.

Using the Oracle ADF Model Tester

Use the ADF Model Tester to isolate a suspected ADF Business Components issue from the view and controller layers.

- Set breakpoints in interface methods (application module impl, view object impl, and so on).
- Run the application module and choose Run or Debug.
- Open the method testing panel:
  - Select AM, VO, or VO Client and then select View > Operations.
- Specify method parameters and click Execute.
- Step through application code by using the debugger.

ORACLE

In earlier lessons, you used the Oracle ADF Model Tester to validate business components without the need to build a user interface. Running or debugging in the ADF Model Tester is also a very good way to isolate an issue that is suspected to reside in the model layer; developers can test the queries, validations, methods, and structure of the business components.

ADF Logger

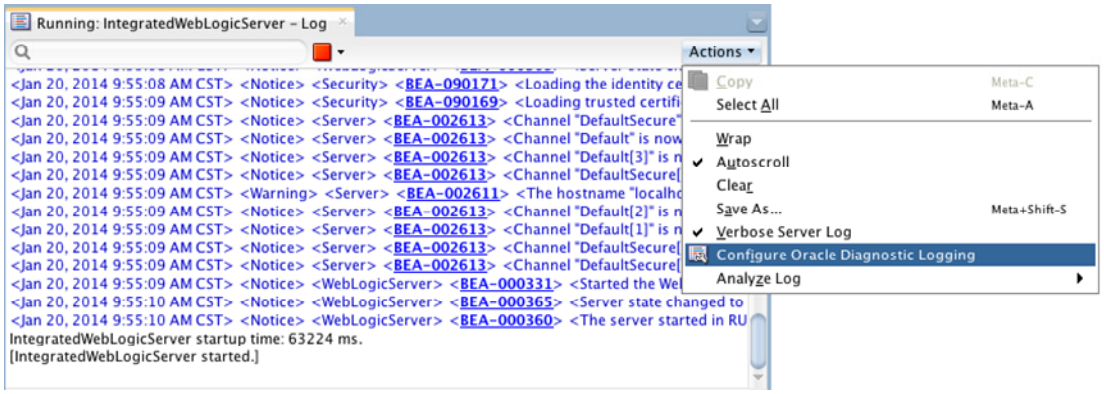
- Replaces more primitive `system.out.println()` or other logging to the message log
- Provides control over which messages are logged
  - Logging level
  - Module filter
  - Logging handlers
- Is configured by modifying the `logging.xml` file, which is accessed via a declarative editor
- Provides an analyzer for viewing log messages

The Oracle ADF Logger is a useful diagnostic tool that is built on the Fusion Middleware logging infrastructure that is called the Oracle Diagnostic Logger (ODL). The logger enables the automatic capture of context information for meaningful analysis. You have control over which messages are logged through the logging level and module filter.

With the ODL, ADF logging messages are handled in the same way as all logging messages in the application server—they are output in a common format and loaded into a common repository where they can be viewed as a single stream or analyzed in different ways. The logger is configured by modifying the `logging.xml` file, which is accessed exclusively via the ODL Configuration declarative editor.

Configuring ADF Logging

- The ODL configuration file controls what is logged and what level of severity is used.
- Access the editor for Oracle Diagnostic Logging Configuration from the Application Servers window or from the Log window.
  - Run the application (or server) first to configure both persistent and transient loggers.



The Integrated Oracle WebLogic Server (WLS) uses `logging.xml` to configure the ODL loggers globally. This file is accessed via the overview editor for Oracle Diagnostic Logging Configuration from the Application Server Navigator or from the Log window in JDeveloper.

The slide shows the Actions menu in the Log window for the running Integrated WebLogic Server. This is available whether or not a particular application is running on the integrated server. To open the ODL configuration for the Integrated WebLogic Server, choose Application Servers from the Windows menu, right-click the Integrated WebLogic Server node and choose Configure Oracle Diagnostic Logging.

Logging that is configured in this way (when the server is not running) will be persistent across running instances of the server. Regardless of the way the ODL configuration file is opened, if the server is running, you can configure transient as well as persistent logging. A transient logger logs only for the duration of the session. When the application server exits, the logging configuration reverts to the original configuration.

Of course, for applications deployed to stand-alone WebLogic Server domains, logging can be configured via Enterprise Manager (although excessive logging can have a negative impact on application performance).

A logging level is a threshold to control the logging of messages. For each logger, the logging can be individually set to any one of the following four severities:

- Failure reporting
  - **SEVERE:** Highest level of severity to catch unexpected errors during normal execution
  - **WARNING:** For internal errors or exceptions for which the user was not notified
- Progress reporting
  - **INFO:** Key flow steps
  - **CONFIG:** Configuration properties and environment settings

After a level is set, only messages that have a severity greater than or equal to the defined level are logged. For example, setting the level to `WARNING` means that logging occurs for `WARNING` and `SEVERE` messages.

Creating Logging Messages

```
import oracle.adf.share.logging.ADFLogger;
...
private static ADFLogger _logger =
    ADFLogger.createADFLogger(AppModuleImpl.class);
...
@Override
    protected void prepareSession(Session session) {
        _logger.info("Creating a new session");
        if (Session.class.equals(null)) {
          _logger.warning("Session is null");
        } else {  try {  doSomethingComplex();
          } catch  anotherAppException aex;
          { _logger.severe("Unexpected exception", aex); }
        }
        super.prepareSession(session);
    }
```


The code in the slide shows an example of adding logging messages to the `prepareSession()` method in an `AppModuleImpl` class.

You can also pass a package to the `createADFLogger()` method to broaden the scope of the logging. This is especially helpful when resource bundles are used for logging rather than strings. A resource bundle can thus be defined for use with logging across all classes in a particular package, thereby enabling the internationalization of the messages.

The `prepareSession()` method is overridden to include some logging messages when the session is created and also to log a warning if the session is null. Messages for other method calls with throwable exceptions are also logged. Of course, this is just one place where you can provide logging in your classes. The next slide shows some key logging points for an ADF application.

Examples of Key Log Points

- ADF Business Components
  - View Object Impl: `bindParametersForCollection()`
  - Application Module Impl: `doCheckout()`, `prepareSession()`
- Task flows
  - Initializer, Finalizer, Error Handler
- Managed beans
  - Common constructor superclass
- Override `DCErrorHandlerImpl` to catch ADFm exceptions.

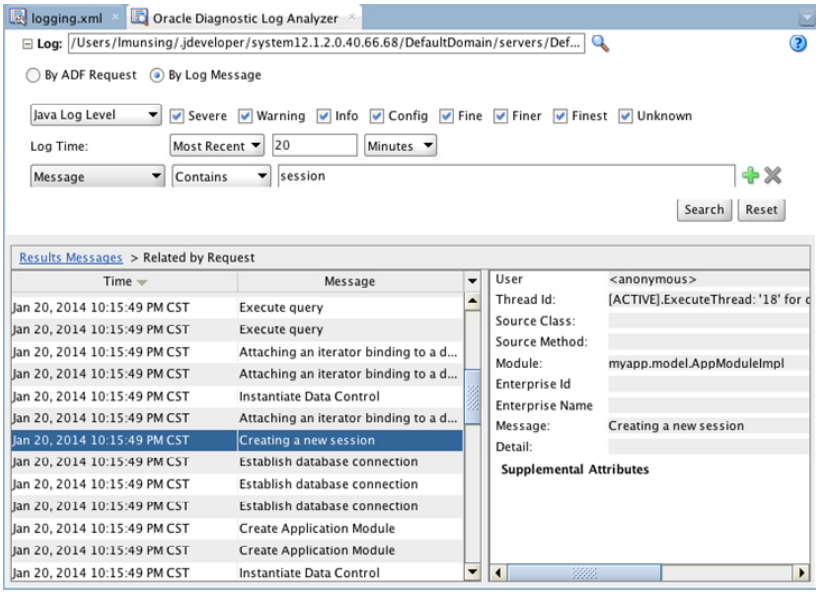
Key points in an ADF application that you may want to log include:

- **ADF Business Components:** The `bindParametersForCollection` method of view object impl files, and the `doCheckout` and `prepareSession` methods of application module Impl files. These points are where bind variable values are overridden and where PL/SQL session setup occurs, respectively. Developers should consider adding logging for these methods to the superclasses of the corresponding Implementation classes.
- **Task flows:** The initializer and finalizer methods of task flows, and any error handlers that are used. This logs any parameters passed to and from a task flow.
- **Managed beans:** Logging a common constructor superclass helps developers understand the life cycle of beans in the application.

Additionally, you can override `DCErrorHandlerImpl` to catch ADFm exceptions. Be aware of the values that are logged, ensuring that no sensitive or restricted access data can be included in the logs.

Viewing Log Messages in the Log Analyzer

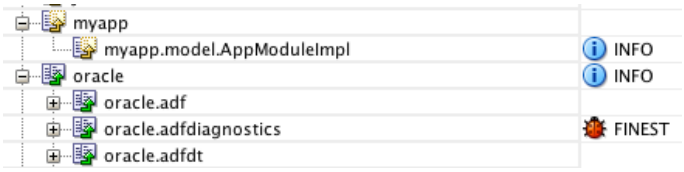
Use the Oracle Diagnostic Log Analyzer to view the entries of a log file. Filter by log level, entry type, log time, entry content, and so on.

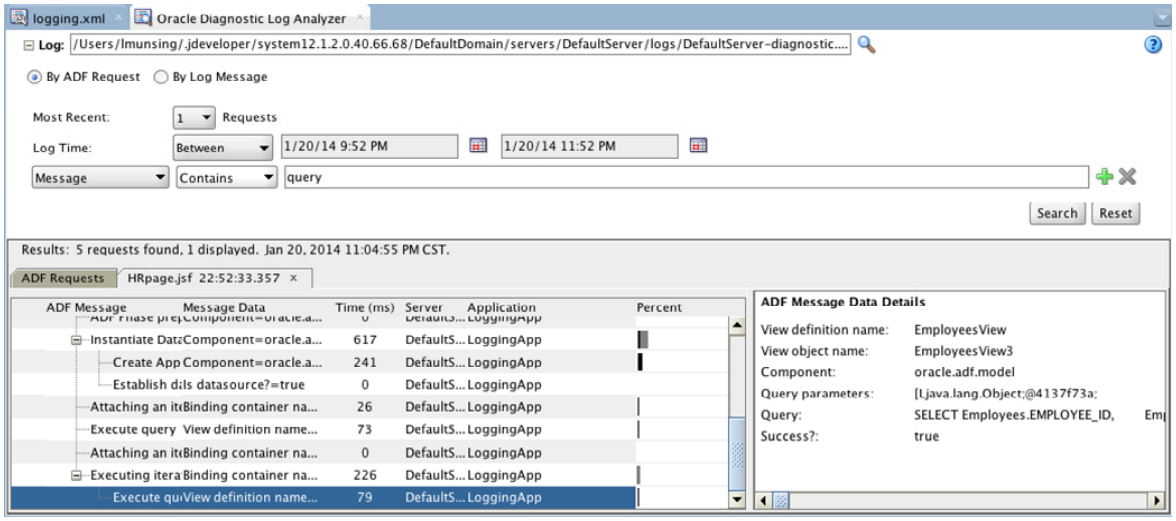


1. From the Tools menu, select Oracle Diagnostic Log Analyzer.
2. In the editor for Oracle Diagnostic Log Analyzer, navigate to the log file or enter the path and name of the log file.
3. Select the corresponding check box for each type of log entry you want to view. You must select at least one type.
   The available ODL log level types are Incident Error, Error, Warning, Notification, Trace, and Unknown. The available Java log level types are Severe, Warning, Info, Config, Fine, Finer, Finest, and Unknown.
4. Specify a time period for the entries you want to view. You can select the most recent period or a range.
5. To filter the results, use the query panel to search on a text pattern. For additional query panels, click Add.
6. To initiate the filters and display the log messages, click Search.
7. To order the results by the message ID, select the “Group by Id” check box.
8. To group the messages by time period or by request, in the Related column, select either “Related by Time” or “Related by Request.” (“Related by Request” is shown in the slide.)

Logging ADF Trace Information

Set oracle.adfdiagnostics module to FINEST to log detailed trace messages from ADF.





Set the oracle.adfdiagnostics module to FINEST in the ODL Configuration utility to log detailed trace messages from the ADF source, including the query issued to the database (as shown in the example). Use the “Related by ADF Request” option to view a complete breakdown of the request by phase, with elapsed times.

Vary Log Messages by Role

Customer Admin

```
ERROR – APP001: Raise process for King did not complete
```

Support

```
WARNING– APP101: Raise failed for Employee King, empty
salary value passed to raise routine
ERROR – NullPointerException
ERROR – APP001: Raise process for King did not complete
```

Developer

```
CONFIG – employee.class: updateSal():
Called with params: empId:101, newsal:<null>
WARNING – APP101: Raise failed for Employee King, empty
salary value passed to raise routine
INFO – employee.class: updateSal():
Setting commission to newSal * 0.1
ERROR – employee.class: updateSal(): NullPointerException
ERROR – APP001: Raise process for King did not complete
```


Logging can be used for both issue notification and diagnosis. In most production systems, logging occurs only for errors or exceptions. However, logging can be configured with additional details depending on the consumer of the logging information.

Consider the different levels of detail that the consumers of your logging may need. There may be a case for adding an extra dimension to your logging to vary the detail of messages at a particular severity level depending on the role of the user. However, clear logging messages or error numbering schemes can alleviate the need for this. For example, you might include logging code like the following to log INFO-level messages for a developer role:

```
if (context.isDevelopmentMode &&
    _logger.isLoggable(Level.INFO) ) {
  StringBuilder logMsg =
          new StringBuilder("Information:");
  logMsg.append(...);
  _logger.info(logMsg.toString());
}
```

JDeveloper Debugger

- The Debugger is very useful for pinpointing problems in your application.
- Set source breakpoints to pinpoint problems in the custom code.
  - Get the ADF Source from Oracle Support to set breakpoints directly in framework code.
- Set exception breakpoints to stop when a particular exception is thrown.
- When a breakpoint is encountered at run time, you can step through the code and view the values of the variables.
- To run an application in debug mode, right-click and select Debug.

JDeveloper's integrated debugger is very useful for finding and fixing problems in your code. The following are some key places to set breakpoints:

- Set a breakpoint in the `doIt()` method of the `JUCtrlActionBinding` class (`oracle.jbo.uicli.binding` package, available by downloading the ADF source). This is the method that executes when any ADF action binding is invoked, and you can step into the logic and look at parameters if necessary. However, this method of debugging is replaced by using the ADF declarative debugger (discussed later).
  You can also set method breakpoints in the following methods:
  - `oracle.jbo.server.ViewObjectImpl.executeQueryForCollection`: This is the method that is called when a view object executes its SQL query.
  - `oracle.jbo.server.ViewRowImpl.setAttributeInternal`: This is the method that is called when any view row attribute is set.
  - `oracle.jbo.server.EntityImpl.setAttributeInternal`: This is the method that is called when any entity object attribute is set, and the same can be achieved by setting a breakpoint at the value binding of the attribute in the page definition and by using the ADF Declarative Debugger.

Understanding Breakpoint Types

Type	Breaks when…
Source	A particular source line in a particular class in a particular package is run
Exception	An exception of this class (or a subclass) is thrown
Method	A method in a given class is invoked
Class	Any method in a given class is invoked
Watchpoint	A given field is accessed or modified

There are several different types of breakpoints, each with different uses:

- **Source:** A particular source line in a particular class in a particular package is run. You can create a source breakpoint in the New Breakpoint window, but it is usually easier to create it by clicking in the breakpoint margin in the editor at the left of the line on which you want to break.
- **Exception:** An exception of this class (or a subclass) is thrown. This is useful when you do not know where the exception occurs, but you know what kind of exception it is, such as a `java.lang.NullPointerException`.
  - The check box options enable you to control whether to break on caught or uncaught exceptions of this class.
  - The Browse button helps you find the fully qualified class name of the exception.
  - The Exception Class combo box remembers the most recently used exception breakpoint classes.

  Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.
- **Method:** A method in a given class is invoked. This is useful to set breakpoints on a particular method that you might have seen in the call stack while debugging a problem. If you have the source, you can set a source breakpoint wherever you want in that class, but this kind of breakpoint enables you to stop in the debugger even when you do not have a source for a class.

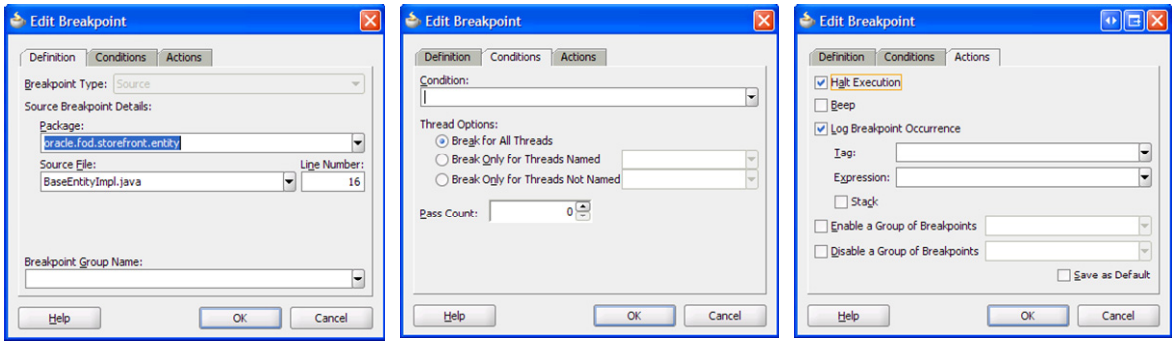
- **Class:** Any method in a given class is invoked. This can be useful when you might know only the class involved in the problem but not the exact method on which you want to stop. Again, this kind of breakpoint does not require a source. The Browse button helps you quickly find the fully qualified class name on which you want to break.
- **Watchpoint:** A given field is accessed or modified. This can be helpful to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can stop the debugger when any field is modified. You can create a breakpoint of this type by using the Toggle Watchpoint menu item in the shortcut menu when pointing at a member field in the source of the class.

To see the Debugger Breakpoints window, use the Window > Breakpoints menu choice from the main JDeveloper menu. Alternatively, you can use the optional key accelerator by pressing Ctrl + Shift + R.

You can create a new breakpoint by selecting Add Breakpoint from the context menu anywhere in the Breakpoints window or by clicking the plus sign (+) on the Breakpoints toolbar.

Setting Breakpoints

- You can create groups of breakpoints that can be enabled or disabled all at once
- You can use conditional breakpoints. Examples:
  - `value instanceof oracle.jbo.domain.Date`
  - `status.equalsIgnoreCase("shipped")`
  - `i > 50`
- You can use actions other than stop with breakpoints



You can use breakpoints for more than just stopping code execution. JDeveloper supports advanced breakpoint actions and conditions for fine-tuning when a breakpoint is triggered and what action happens. Virtually any Boolean Java expression can be a breakpoint condition, and it can be applied to all threads or a specific set of named threads.

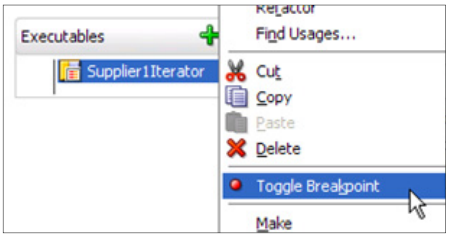
The first of the conditional breakpoint examples in the slide breaks only when the variable value is a Date type. The second example breaks only when the status is “shipped.” The third example breaks only when the variable `i` is greater than 50. The third example (`i` > 50) is particularly useful if you have a breakpoint inside a loop (with `i` as the loop variable) but you know your problem occurs only when the loop has executed a number of times.

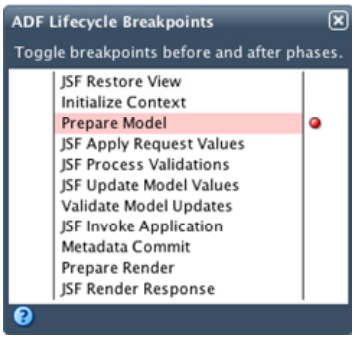
For actions, breakpoints typically just break into the code. However, you can have the debugger sound an audible beep, log a message or expression result to the log console, dump the stack, or enable or disable a group of breakpoints.

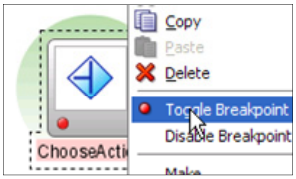
ADF Declarative Debugger

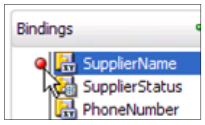
The Declarative Debugger enables you to set breakpoints on:

- Executables
- Task flow activities
- Bindings
- Lifecycle Events









ORACLE

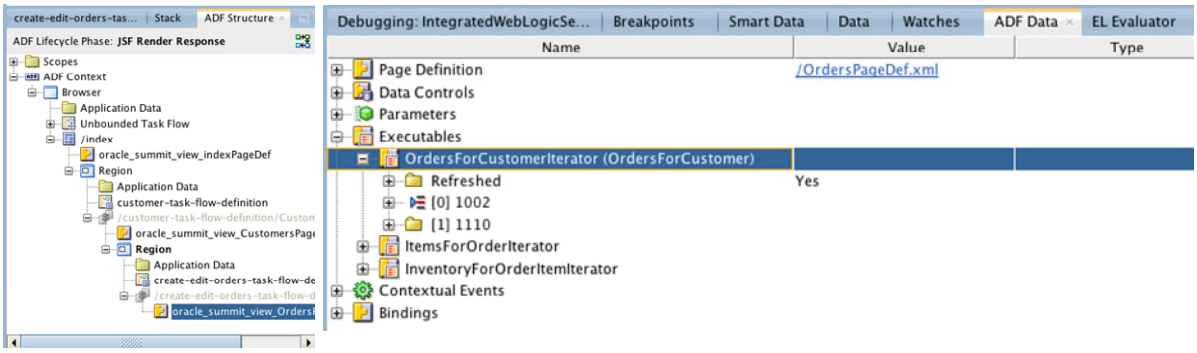
You use the ADF Declarative Debugger in JDeveloper to set breakpoints on ADF task flow activities, page definition executables, and method, action, and value bindings. Instead of needing to know all the internal constructs of the ADF code, such as method and class names, you can set breakpoints at the highest level of object abstraction. You can also set breakpoints at specific ADF lifecycle phases and contextual events.

You can set or remove such breakpoints by right-clicking task flow activities in the task flow diagram editor, or by right-clicking bindings or executables in the page definition file and selecting Toggle Breakpoint. You can also set or remove breakpoints on executables or bindings by clicking the left or right margins in the binding editor for “before” or “after” breakpoints, respectively.

For example, when you set a breakpoint on the Choose Action router activity in the CheckoutFlow, a red dot icon appears on the activity, as shown in the slide. When the breakpoint is reached at run time, the application pauses and the icon changes to a red arrow. This prevents developers from having to know the class or method on which to place the breakpoint to discover what values are being passed to the activity. But Java code breakpoints are also supported because the ADF declarative debugger is built on top of the Java debugger.

ADF Declarative Debugger

Breakpoint reached:

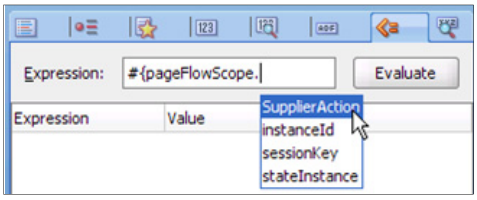


After the application pauses at the breakpoint, you can view the runtime structure of the objects as a tree in the ADF Structure window. All loaded view ports are displayed in the structure. The ADF Data window displays a list of data for a given object that is selected in the ADF Structure window, including arguments, variables, and fields.

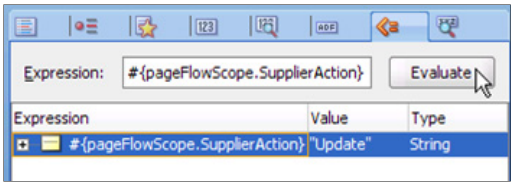
Using the EL Evaluator at Breakpoints

The Expression Language evaluator enables:

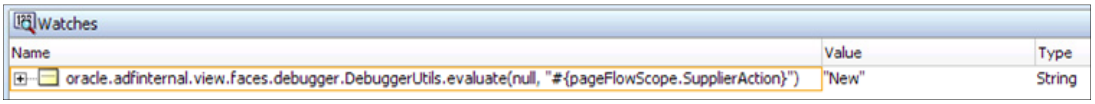
- Introspection of the values between the view and model
- Watches based on EL



Using the discovery function



An evaluated expression



An evaluated expression on the watch list displays the changed value.

When you debug and reach a breakpoint, you can use the EL Evaluator to examine the value of an EL expression at that point of execution. The EL Evaluator appears as a tab in the debugger window area; you can select View > Debugger > EL Evaluator if it is not visible.

Enter an EL expression in the input field. When you enter `#{` in the field, a discovery function helps you select the correct expression to evaluate, and auto-completion also facilitates expression entry. You can evaluate several EL expressions at the same time by separating them with semicolons. After you have entered the expression or expressions, click Evaluate. If the EL expression no longer applies within the current context, the expression evaluates to `null`.

The EL Evaluator is different from the Watches window. EL evaluation occurs only when stopped at a breakpoint, and not when stopped at subsequent debugging steps.

Developing Regression Tests with JUnit

1. Download JUnit extensions.
2. Create a JUnit testcase.
   - Use wizards to create skeleton test cases and group them into suites.
   - Add code to verify queries, validation, and so on.

```
@Test (expected=AttrValException.class)
  public void testCustomerIdRequired() {
    ViewObject view = fixture1.getApplicationModule().findViewObject("Customers");
    oracle.jbo.Row r = view.createRow();
    r.setAttribute("Name", "Test Customer");
    r.validate();
  }
```

3. Run the test directly or via ant.

ORACLE

JUnit is a testing utility that enables developers to write tests to evaluate the code quality of an application. You can create JUnit test cases for nearly any kind of Java application. An extension is available for JUnit, and another extension is available for building test cases that are explicitly used to test ADF Business Components.

To create test cases, download both JUnit extensions from the Help > Check for Updates menu item. Then use the wizard in JDeveloper to generate test cases and fixtures. A simple test is created for each view object and simply tests that the view object is found. Additional tests should be added by developers (for example, to test that a mandatory field is required, as shown in the slide). Developers can run the test directly by selecting the test suite class and choosing Run from the context menu. Or, to automate tests, a target can be built in Ant to run the tests.

Summary

In this lesson, you should have learned how to:

- Define and use tools for logging and diagnostics
- Use the information in the logging window to better understand where a problem exists in the application
- Use the Oracle ADF Model Tester to debug business services
- Use JDeveloper to debug an application
- Use JUnit to help prevent regressions

Practice 17 Overview: Debugging

This practice covers the following topics:

- Using JDeveloper to set break points in your application
- Examining the application status in the various windows of the debugger when the application pauses at a breakpoint

18

Implementing Security in ADF Applications

Objectives

After completing this lesson, you should be able to:

- Explain the benefits of secure applications
- Describe the security aspects of an ADF Business Components application
- Implement ADF security
  - Authentication
  - Authorization
    - In the data model
    - In the UI for task flows and pages
- Access security information programmatically
- Use the Expression Language to extend security capabilities

ORACLE

Lesson Aim

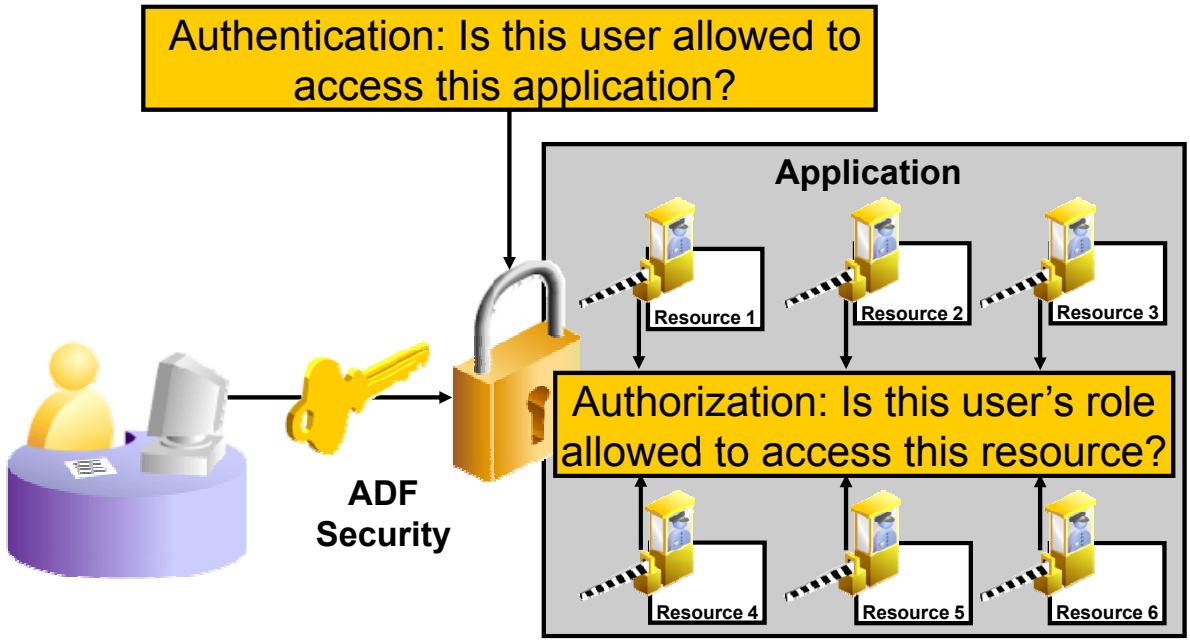
This lesson describes the methods and techniques for applying security principles to ADF Business Components applications.

Benefits of Securing Web Applications

- Web applications often connect with a single database user account. Therefore, separate application users accounts must be used.
- Identity can be used to:
  - Ensure that only authenticated users can access the application
  - Restrict access to parts of the application
  - Customize the UI
  - Provide the user name for auditing
  - Set up a Virtual Private Database (VPD)

One of the major challenges of web applications is the notion of identity when authentication is not done by the database that is providing the data. The identity of the end user is often needed for a variety of purposes (as shown in the slide).

Authentication and Authorization

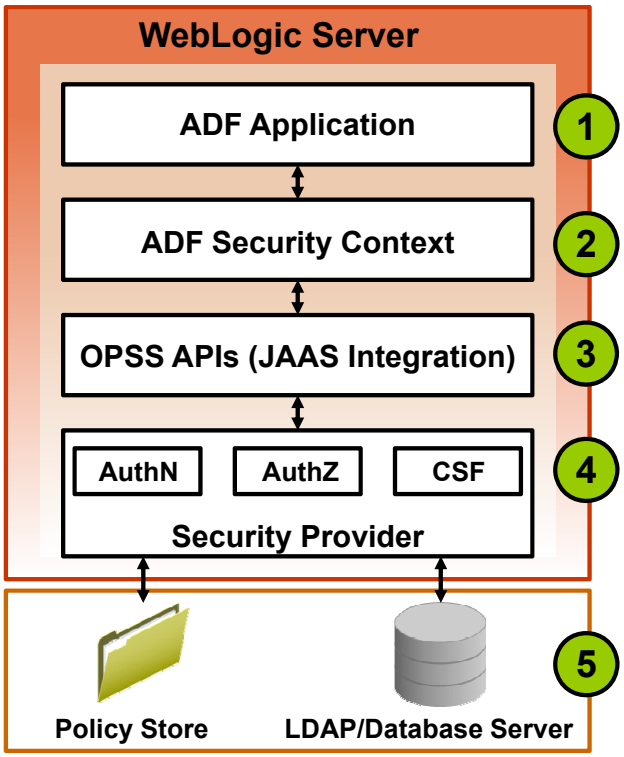


ADF Security is the preferred technology for providing authentication and authorization services to Fusion web applications. Authentication and authorization are two key aspects of security for web applications:

- **Authentication** determines which users can access the application.
- **Authorization** determines what functions users are allowed to perform after they enter the application. You can control this by granting certain functionality to the roles that are defined for the application. You specify which objects, pages, and task flows are available to each role.

ADF Security implements a Java Authentication and Authorization Service (JAAS) security model. JAAS is a standard security API that enables applications to authenticate users and enforce authorization. Although other security-aware models exist that can handle user login and resource protection, ADF Security is ideally suited to provide declarative, permission-based protection for ADF resources, such as bounded task flows, top-level web pages, and rows defined by entity objects and attributes.

ADF Security Framework and OPSS

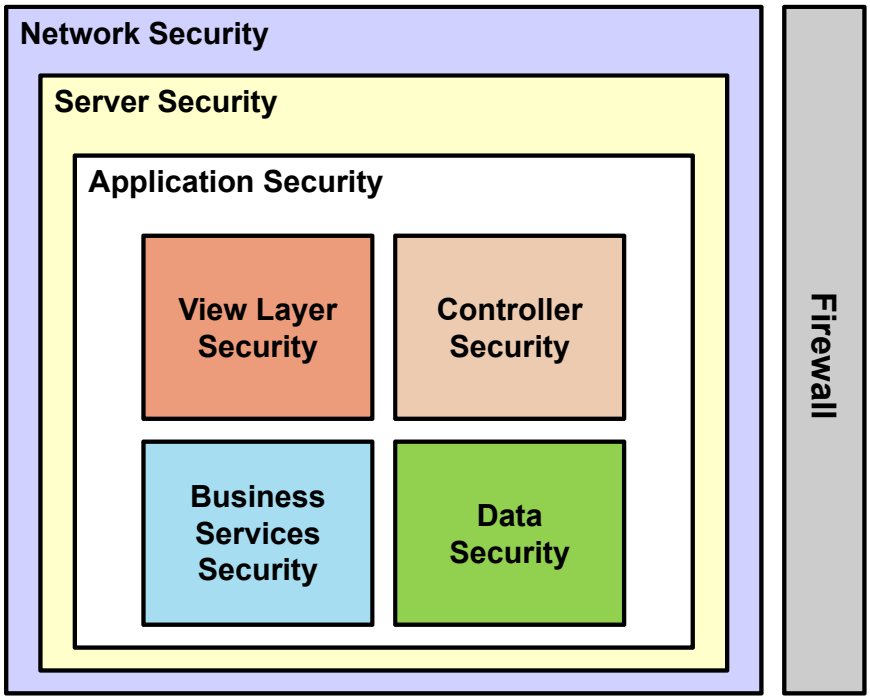


1. Sends request to authenticate or authorize user for a specific page or resource
2. Passes the request to OPSS
3. Delegates to an authentication and authorization provider
4. Redirects request to an identity management system.
5. Provides authentication and authorization services

ADF Security is built on top of the Oracle Platform Security Services (OPSS) architecture, an abstraction layer within the Oracle Fusion Middleware technology stack that insulates developers from security and identity implementation details. OPSS is a set of APIs that allow clients such as Oracle ADF to authenticate and authorize users for specific resources or pages that are protected. OPSS is a hybrid approach to web application security that combines JAAS capabilities with container-managed security.

ADF applications themselves do not perform authentication or authorization. Instead, they issue requests for authentication and authorization through the ADF Security Context. (The ADF Security Context for an application is established when you enable ADF Security on an ADF application.) OPSS delegates authentication to an authentication provider, which then redirects the request to an identify management system. The identity management system can be an LDAP server, the database, or anything else that you have configured on WebLogic Server to perform user authentication. Similarly, authorization is controlled by a policy store, which by default is a file-based store but can also be configured to be LDAP or the database server.

Securing the Layers of an ADF Application



As shown in the slide, application security is one part of an overall security strategy that includes network, server, and peripheral security. You can apply security to the following layers in an ADF application:

- In the **view** layer, you can require users to authenticate before accessing a page, and you can use the Expression Language to conditionally render (or disable) UI components based on the user's identity.
- In the **controller** layer, you can restrict access to bounded task flows and top-level pages, and you can use router activities to define navigation based on the user's identity.
- In the **business services** layer, you can restrict access to entities and attributes based on the user's identity.
- In the **data** layer, you can leverage built-in database security, such as Virtual Private Database (VPD) and proxy user access, to protect data.
- A fifth layer (not shown) also exists when you are integrating applications in a service-oriented architecture (SOA) or calling remote services such as web services.

Remember, however, that application security is part of an overall perimeter security that includes physical server protection (such as firewalls) and network security. You need to implement an overall security strategy that addresses all potential vulnerabilities.

Steps for Implementing ADF Security

To implement ADF Security in a Fusion web application:

1. Configure security for the application.
2. Create test users and add them to groups (enterprise roles).
3. Create application roles and add users and groups to those roles.
4. Define security policies to grant access to ADF resources such as:
   - Bounded task flows
   - Page definition files (for pages in unbounded task flows)
   - Entity objects and attributes

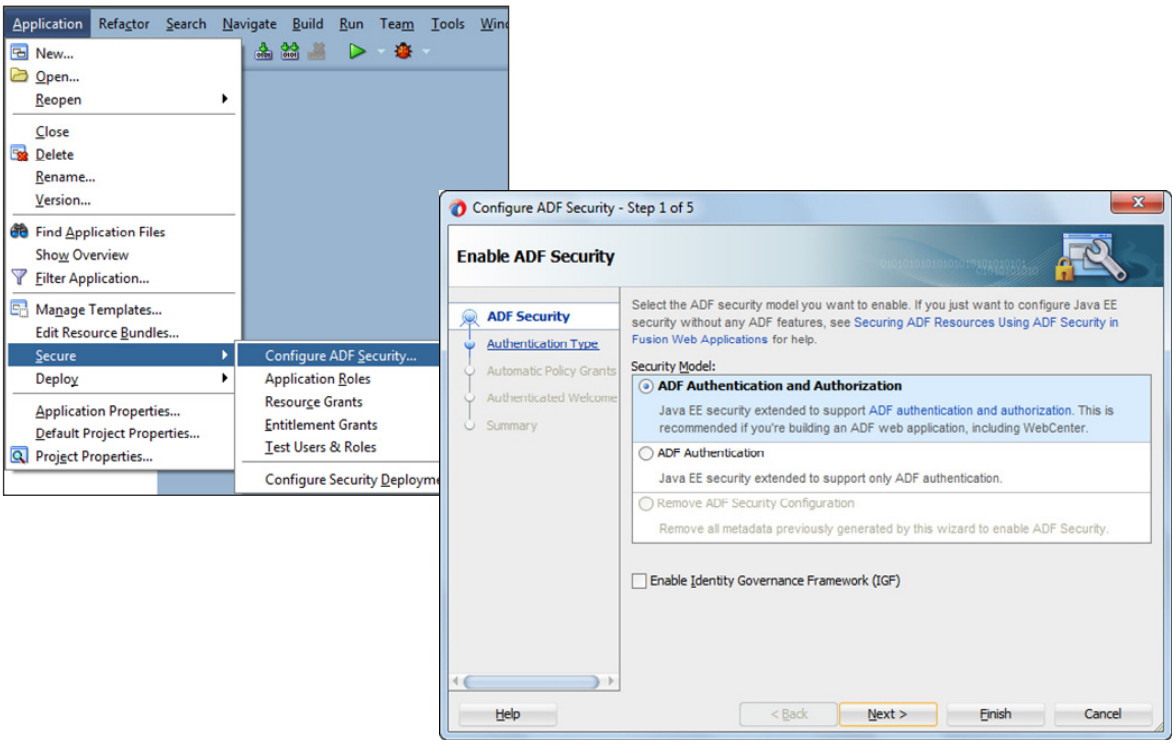
ORACLE

ADF Security is well integrated in the JDeveloper IDE, enabling you to define security on ADF resources declaratively. You enable ADF security for an application by running the Configure ADF Security Wizard. The wizard configures ADF Security for the entire Fusion web application, so that any web page associated with an ADF security-aware resource is protected by default.

For testing purposes, you can create test users and add them to groups (known as *enterprise roles*) that simulate accounts that are available in the production environment. Before you can grant users access to specific resources, you must create application roles that are mapped to enterprise groups. OPSS supports the mapping of application roles to enterprise groups in the policy store configured for the WebLogic Server domain.

After you configure ADF Security, you grant users access rights to specific security-aware ADF resources through their enterprise role membership. These access rights are known as *security policies*. Because ADF Security is based on JAAS, security policies identify the principal (the user or application role), the ADF resource, and the permission. You can define security policies in the Fusion web application for the following ADF security-aware resources: ADF bounded task flows, page definition files, and entity objects and attributes. Ultimately, it is the security policy on the ADF resource that controls the user's ability to enter a task flow, view a web page, or access data.

Configuring ADF Security



The Configure ADF Security Wizard enables you to configure ADF Authentication and Authorization or to configure only ADF Authentication. A third option enables you to remove all ADF security.

To invoke the wizard, in the Applications window, select the view-controller project that contains the pages you want to protect, and then select Application > Secure > Configure ADF Security.

On the first page of the wizard, choose the type of security to configure:

- Choose ADF Authentication and Authorization to define fine-grained security policies that ADF Security enforces.
- Choose ADF Authentication to enable support for dynamic authentication without implementing the fine-grained authorization of resources available with ADF Authorization.

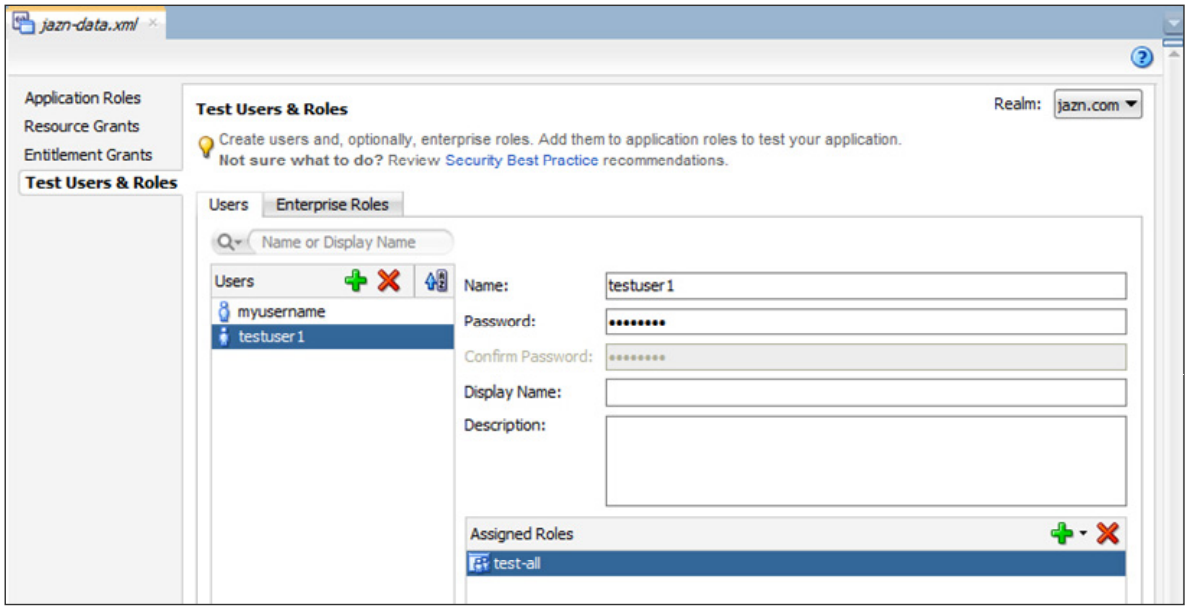
When you select only ADF Authentication, the container (which manages authentication) requires the user to log in the first time a page in the application is accessed. The container also redirects the user to the requested page when the user has sufficient access rights as defined by security constraints. In this case, the developer is responsible for defining security constraints for webpages; otherwise, all pages are accessible by the authenticated user.

On the remaining pages of the wizard, you specify additional security settings, such as:

- The authentication type for the web project. The authentication type specifies how the user credentials are received (through the browser's native login dialog box, a client certificate, a form, and so on)
- Whether to lock access to all objects until access is granted to objects manually, or to grant "view" access to a special test-all role
- Whether to redirect to a specific Welcome page when a user is successfully authenticated

Creating Test Users and Enterprise Roles

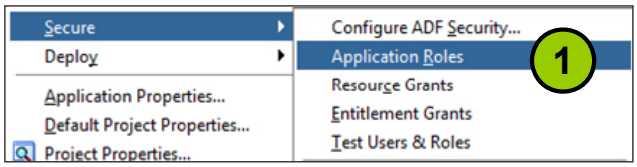
Create test users and (optionally) add them to enterprise roles:



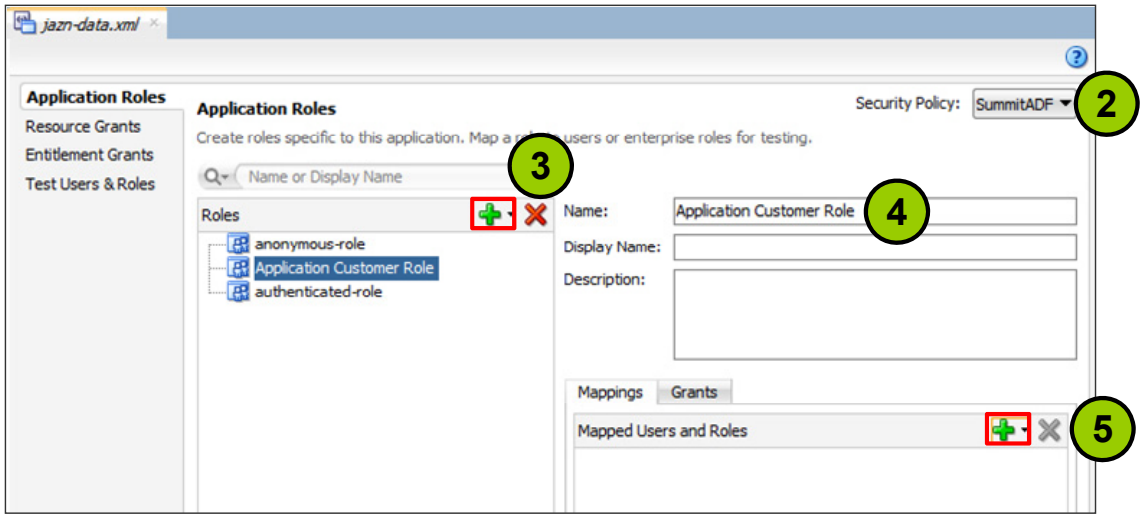
You can seed the identity store of your application with a temporary set of users and enterprise roles to simulate the actual users' experience in your production environment. Enterpriser roles simply enable you to group users by role.

Because you typically configure the deployment options in JDeveloper to prevent migrating the identity store to a staging environment, the enterprise roles that you create in the `jazn-data.xml` file are for convenience only. For the WebLogic platform, however, an enterprise role can be used to generate a WebLogic group at deployment time.

Creating Application Roles



Adds application roles to the policy store defined by `jazn-data.xml`



You create application roles to represent the policy requirements of the application and to define groups of users with the same set of permissions. The application roles that you create in the application policy store are specific to your application. For example, you might have an application that is used by both customers and employees. You can define two separate application roles—Application Customer Role and Application Employee Role—to specify two different sets of permissions.

To create an application role:

1. From the Application menu, select Secure > Application Roles.
2. On the Application Roles page of the `jazn-data.xml` overview editor, make sure that the policy store for your application is selected in the Security Policy list. The policy store that JDeveloper creates in the `jazn-data.xml` file is automatically based on the name of your application.
3. In the Roles area, click the New icon and add the new role. You can create nested roles by selecting a role and then adding a new role to it, or you can group related roles by adding role categories.
4. In the Name field, enter the name of the role and click any other field to add the application role to the policy store.

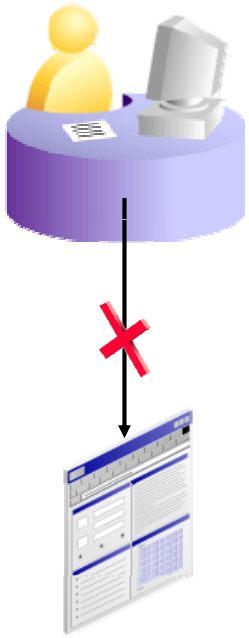
5. If you have already set up test users in the identity store, you can select an application role and, on the Mappings tab, add users or enterprise roles to the application role. (You use enterprise roles to group users and other enterprise roles for grants or inclusion in application roles.)

At run time, the application role to which a user is assigned determines access rights. Therefore, before you can define security policies for ADF resources, the policy store must contain the application roles to which you intend to issue grants (you learn how to issue grants next). The application role can be one that you define (such as Application Customer Role), or it can be one of the two built-in application roles defined by OPSS: authenticated-role or anonymous-role.

When you use the overview editor to create an application role, JDeveloper adds the role to the policy store defined by the `jazn-data.xml` file, which appears in the `Descriptors/META-INF` node of the Application Resources panel.

Defining Security Policies

- Authorization relies on security policies in the policy store.
- When authorization is enabled, security-aware resources are inaccessible (by default) until you define explicit grants.
- You can define security policies to grant access to:
  - Bounded task flows
  - Page definition files (for pages in unbounded task flows)
  - Entity objects and attributes
  - Other resources



Protected by default

Authorization relies on a policy store that is accessed at run time and that contains permissions that grant privileges to execute predefined actions, like `view`, on a specified object. Initially, after you run the Configure ADF Security Wizard, the policy store defines no grants. If you chose to enable ADF Authentication and Authorization when you ran the wizard, any web pages in your application that rely on the ADF security-aware resources will be inaccessible to users. To make the pages accessible, use JDeveloper to define explicit grants for the resources that you want to permit users to access.

You can define security policies in Fusion web applications for the following ADF security-aware resources:

- **ADF bounded task flows:** You can protect the entry point to the task flow, which in turn controls the user's access to the pages contained by the flow.
- **ADF page definition files:** You can protect specific pages in an unbounded task flow. The unbounded task flow itself is not an ADF Security–aware component and thus does not participate in authorization checks. Therefore, you can protect the pages of an unbounded task flow by defining grants for the page definition files associated with the pages.
- **ADF entity objects and attributes:** You can protect data access through entity objects and attributes.

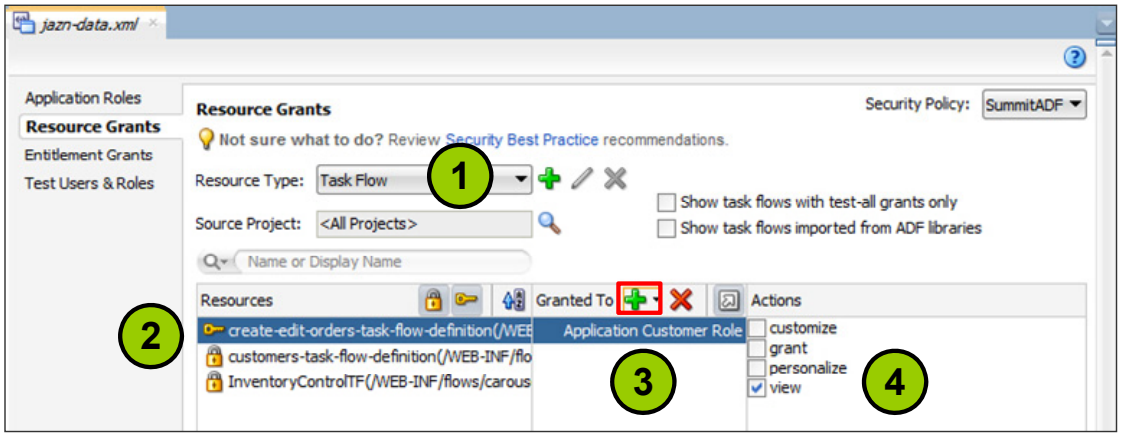
You can protect other resources (such as menu items, buttons, and tabs) by defining custom resource permissions and using the Expression Language to conditionally render or disable the component based on whether the user has been granted access to the resource.

Note: Resource permissions are beyond the scope of this course. For more information about resource permissions, see the section on enabling security in *Developing Fusion Web Applications with Oracle Application Development Framework:*

http://docs.oracle.com/middleware/1212/adf/ADFFD/adding_security.htm

Granting Access to Bounded Task Flows

- Prevents unauthorized access to secured task flows
- Provides developers with the ability to:
  - Secure a bounded task flow as a logical entity
  - Write security-aware bounded task flows and pages



Bounded task flows are secured by default when ADF Security is enabled. To make a bounded task flow accessible, you must grant the `view` permission to any application roles that need to read and execute the bounded task flow. When you define the security policy at the level of the bounded task flow, you protect the flow's entry point; all pages within that flow are then secured by the policy that it defines.

Securing task flows can enable simpler security administration by allowing security to be controlled in terms of a unit of application functionality, rather than having to deal with all the individual pages that may arise in the implementation of that application function.

To grant view permissions on bounded task flows, use the overview editor for security policies (select Secure > Resource Grants from the Application menu):

1. On the Resource Grants page of the overview editor, select Task Flow from the Resource Type list. The overview editor displays all the task flows that your application defines.
2. In the Resources column, select the task flow for which you want to grant access rights. Notice that a lock icon appears next to resources that have no grants. The lock icon indicates that the resource is inaccessible until you define a grant. You can click the key icon in the toolbar to hide resources that already have grants.

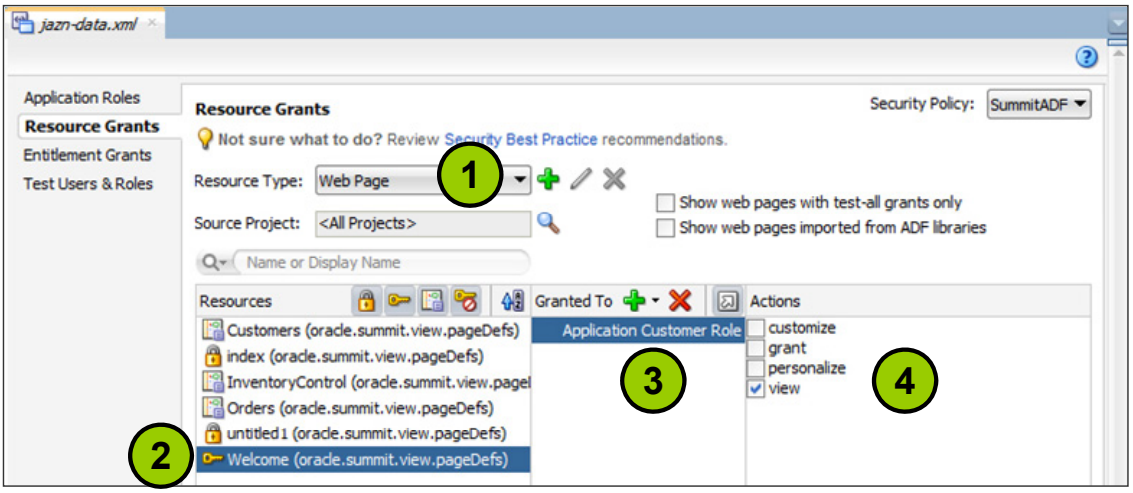
3. In the Granted To column, click the Add Grantee icon (the green plus sign) and select Add Application Role. Then select the application role that will be a grantee of the permission. You can choose an application role that you created, or you can choose one of the built-in OPSS application roles: anonymous-role and authenticated-role. If the role you want to use does not yet exist, you can save time by creating the new application role from this dialog box.
4. Back on the Resource Grants page, leave the `view` action selected in the Actions column. The `view` action is the only action that is currently supported for Fusion web applications. Do not select the `customize`, `grant`, or `personalize` actions; these are reserved for use with WebCenter.

You can repeat these steps to make additional grants as needed. The same task flow definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file.

After you grant access to a bounded task flow, you can use the Expression Language (EL) to check whether the user is authorized to access a task flow. You learn how to do this later in this lesson.

Granting Access to Pages

- Determines whether the user is allowed to view a page
- Is not needed on pages in secured task flows



ADF page definitions are secure by default when ADF Security is enabled. You can make individual pages in an unbounded task flow accessible by granting permissions on the ADF page definition. To make a page accessible to authenticated users, you must grant the `view` permission to any application roles that need to view the page.

Tip: Page-level security is not checked for pages within a bounded task flow. If you want to provide a higher level of security for a page within a bounded task flow, you must wrap the page in a nested bounded task flow and grant permissions on the nested task flow.

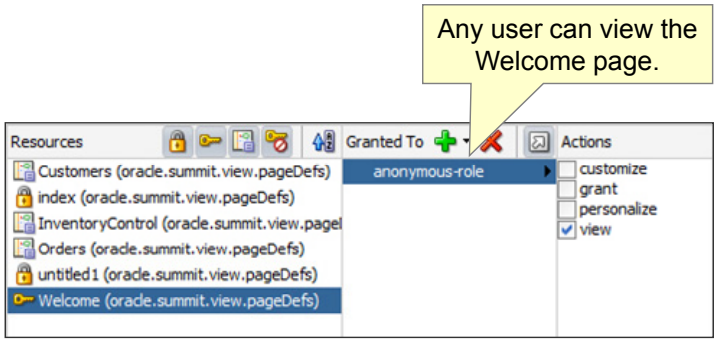
To grant `view` permissions on a page definition file, use the overview editor to edit the `jazn-data.xml` file (select Secure > Resource Grants from the Application menu):

1. On the Resource Grants page of the overview editor, select Web Page from the Resource Type list. The overview editor displays all the pages for which there are page definition files. For pages that have no bindings, you must generate an empty page definition file before you can secure the page.
2. In the Page Definition column, select the page.
3. In the “Granted To” column, click the Add Grantee icon and add one or more application roles.
4. Back on the Resource Grants page, in the Actions column, leave the `view` action selected.

Making ADF Resources Public

Use prebuilt OPSS application roles to make ADF resources public:

- Use `anonymous-role` to grant access to any user.
- Use `authenticated-role` to grant access to authenticated users only.



Resource Grants Page of `jazn-data.xml`

You probably have some pages in your application that you want to make available to all users, regardless of their specific access privileges. For example, you might have a Welcome page that should be accessible to all visitors to the site. You might also have some corporate web pages that should be available only to users who have identified themselves through authentication. In both cases, the pages are public in the sense that the ability to view the pages is not defined by the users' specific permissions.

In the ADF security model, you can use one of the following prebuilt OPSS application roles to make a resource public:

- Use `anonymous-role` to grant access privileges to both known (authenticated) and anonymous users.
- Use `authenticated-role` to grant access privileges to authenticated users only.

Granting Access to Business Services

- Prevents unauthorized access to entity objects or attributes
- Enables developers to:
  - Secure access to an entire entity object or only certain attributes
  - Specify the actions that members of a role can perform on entity objects or attributes
- Involves a two-step process:
  - In the model layer, enable security on specific entity objects and attributes.
  - In `jazn-data.xml`, grant privileges to specific application roles.

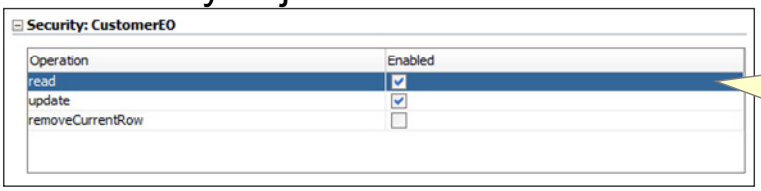
Unlike security-aware resources in the view layer, entity objects and attributes in the model layer are not protected by default when ADF Security is enabled. To prevent unauthorized access, you must enable security on specific entity objects and attributes, and then authorize access to the objects by granting privileges to one or more application roles (or users) in the `jazn-data.xml` file.

When you secure an entity object, you effectively secure any view objects that rely on that entity object. As a result, entity objects that you secure define an even broader access policy that applies to all UI components bound to this set of view objects.

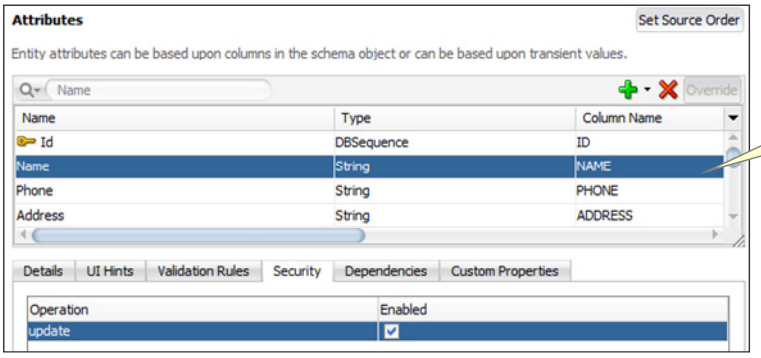
Enabling Security on Entity Objects and Attributes

In the overview editor of an entity object, enable security for:

- Entire entity objects



- Individual attributes



You enable ADF Security on an entity object by specifying the operations that are allowed on the object and its attributes.

To enable security on the entire entity object:

1. In the Applications window, double-click the entity object that you want to secure.
2. In the overview editor, click the General tab.
3. On the General page, expand the Security section, and then select the operations that you want to secure for the entity object.

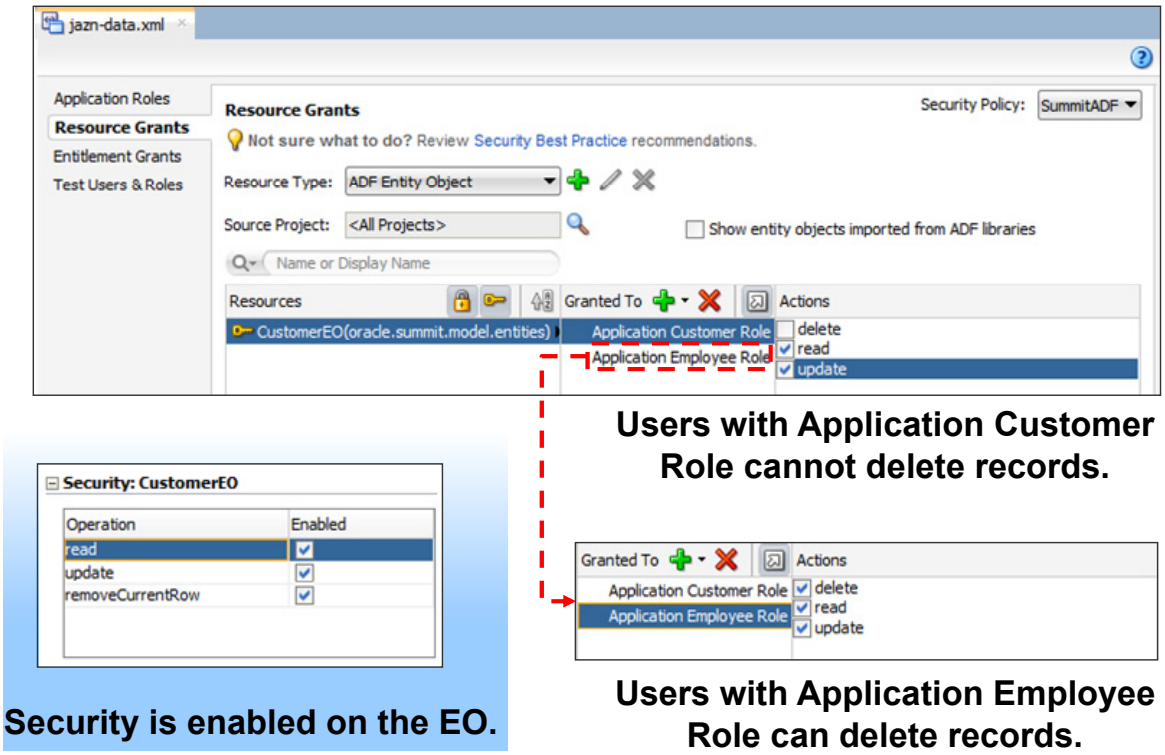
To enable security on individual attributes:

1. In the overview editor, click the Attributes navigation tab.
2. On the Attributes page, select the attribute to secure, and then click the Security tab and select the update operation.

The entity objects and attributes are not secured until you explicitly define policy grants in the `jazn-data.xml` file.

You can also access security settings by right-clicking entities and attributes in the Structure window.

Granting Privileges on Entity Objects and Attributes



After enabling security in the model layer, you must grant privileges to one or more application roles (or users) in the `jazn-data.xml` file.

To grant view privileges on an entity object or attribute:

1. On the Resource Grants page of the overview editor, select ADF Entity Object or ADF Entity Object Attribute from the Resource Type list. The overview editor displays all entity objects or attributes for which you have enabled security.
2. In the Resources column, select the entity object or attribute that you want to secure.
3. In the Granted To column, click the Add Grantee icon and add one or more application roles.
4. In the Actions column, select the actions that you want to grant to the selected role. You can select an action only if you have configured the corresponding operation in the entity object or attribute. For example, you can select the delete action only if you have enabled security for the removeCurrentRow operation.

The slide shows security that is enabled for the CustomerEO entity object. The read, update, and removeCurrentRow operations are all enabled for security. In the `jazn-data.xml` file, notice that two application roles are granted access to the entity object: The Application Customer Role is granted the ability to read and update records, but not to delete them. The Application Employee Role, however, is granted the ability to perform all available actions.

Application Authentication at Run Time

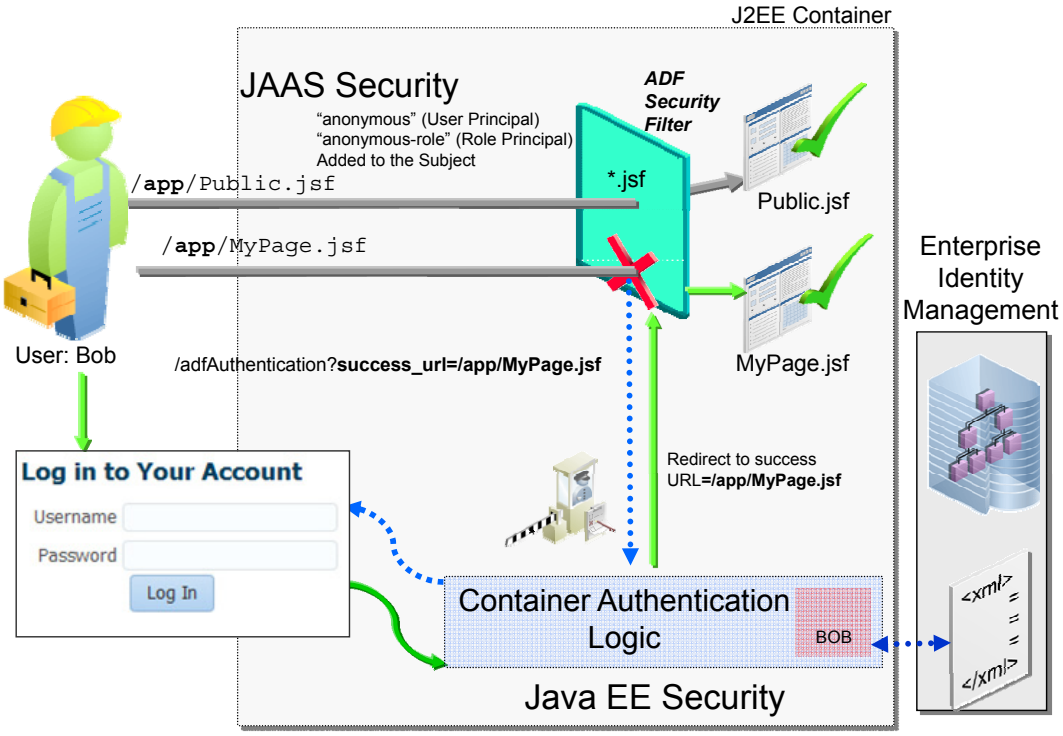
Two types of authentication

- **Implicit:** Authentication happens automatically when the user accesses a secured page.
- **Explicit:** The user is first directed to a public page and then navigates to a login page.

When you use ADF Security in your application, you can choose one of two authentication types:

- **Implicit:** Direct a user to a secured page, and authentication happens automatically.
- **Explicit:** Direct a user to a public page first, and then have the user navigate to a login page.

ADF Security: Implicit Authentication



In an implicit authentication scenario, authentication is triggered automatically if an unauthenticated user tries to access a page.

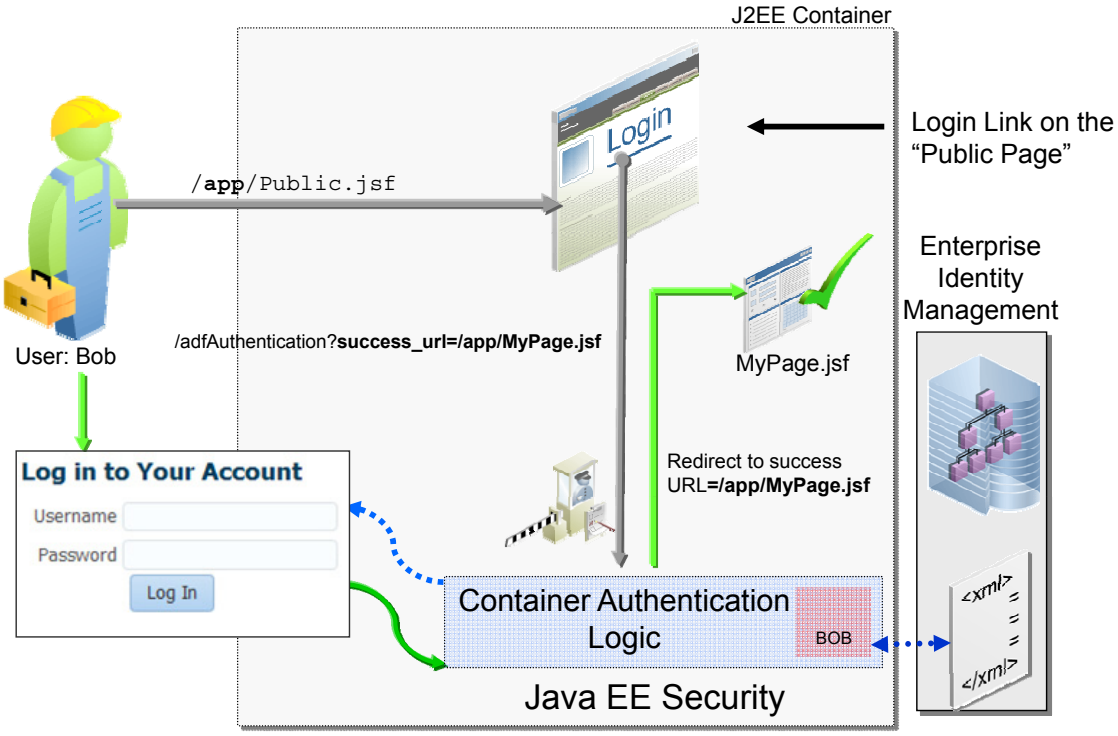
If the requested page is public (defined as viewable by users with `anonymous-role`), the user is allowed to access the public page. For example, Bob is allowed to access `Public.jsf`.

If the requested page (such as `MyPage.jsf`) is secured, the request is redirected to the container, and the container invokes the configured login mechanism. The example shown in the slide uses form-based login, in which the user enters credentials and posts the form back to the Java EE container, which is responsible for authenticating the user.

If the login is successful and ADF Authorization is enabled, another check is performed to verify that the authenticated user has `view` access to the requested page. Users who are authorized to access the page are then forwarded to the requested page.

In a later slide, you learn more about how users are authorized.

ADF Security: Explicit Authentication

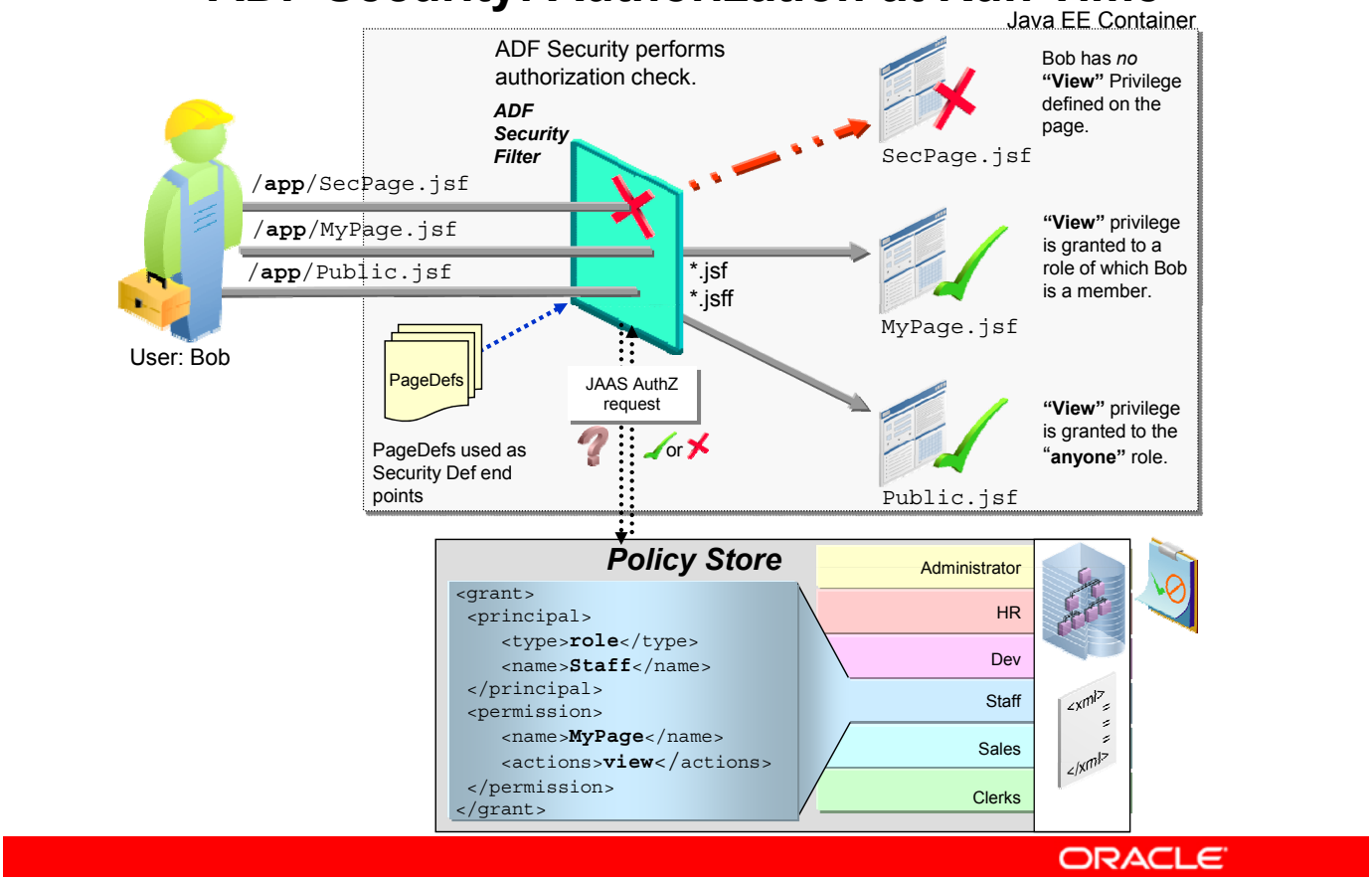


In an explicit authentication scenario, you might have a public page with a login link that, when it is clicked, triggers an authentication challenge to log in the user. The login link may optionally specify some other target page that should be displayed after successful authentication (assuming that the authenticated user has the correct privileges to access the page).

In this scenario, the request to authenticate is redirected to the container's login component. If login is successful and ADF Authorization is enabled, another check is performed to verify that the authenticated user has `view` access to the requested page. If the user is authorized to access the page, the user is forwarded to the requested page.

You learn more about how users are authorized in the following slides.

ADF Security: Authorization at Run Time



As you learned earlier, the security implementation in a Fusion web application is based on JAAS policies, which are held in a policy store (independent of the application) and accessed at run time. The policy determines whether a user has access to the defined web resource and what granular permissions (actions) have been granted to that user.

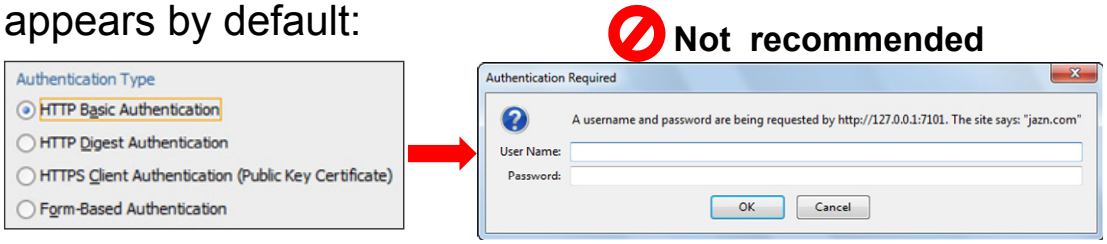
In this case, the user Bob is a member of the enterprise role Staff in the identity management solution.

Assuming that Bob is already logged in and successfully authenticated, when he tries to access `mypage.jsf`, the Oracle ADF Security enforcement logic intercepts the request and checks the page definition of that page to see whether permission is required. In the example in the slide, the `view` privilege on the requested page has been granted to the Staff role; for Bob, therefore, the `mypage.jsf` appears.

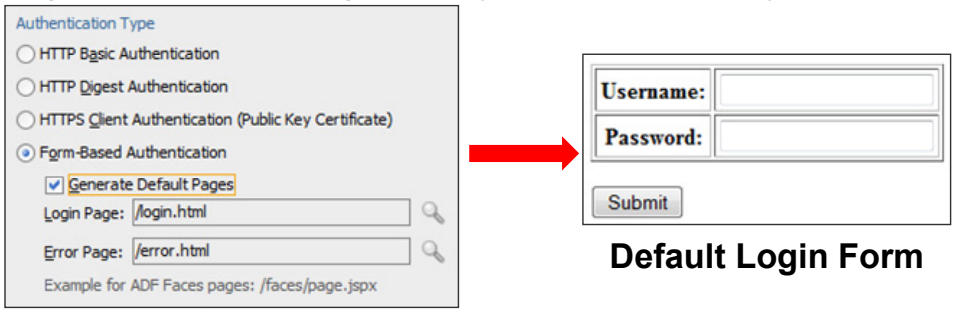
However, when Bob tries to access `SecPage.jsf` later, permission is required and Bob does not belong to a role that has access to the resource; therefore, a security error appears.

Implementing a Login Page for Implicit Authentication

- With basic authentication, the browser's login dialog box appears by default:



- With form-based authentication, you can generate default login and error pages, or you can create your own pages:



Default Login Form

In the Configure ADF Security Wizard, you choose the type of authentication that the application should use. The authentication type determines the kind of login page that is displayed (by default) if a user who is not yet authenticated attempts to access a protected web page. This is *implicit authentication*. After the user successfully logs in, another check is performed to verify whether the authenticated user has `view` access granted on the requested page's ADF security-aware resource.

Authentication Types

- **HTTP Basic Authentication:** Browser authentication is used. The user name and password are not encrypted. This is useful for testing authentication without the need for a custom login page.
- **HTTP Digest Authentication:** Browser authentication is used. The user name and password are encrypted, and for this reason it is more secure than basic authentication.
- **HTTPS Client Authentication (Public Key Certificate):** This strong authentication mechanism uses a certificate and transmits over SSL.

- **Form-Based Authentication:** The developer can specify a login page and an error page to display when login fails. If you select the Generate Default Pages check box, JDeveloper generates a simple HTML login page and error page for you. The generated login page supports container-managed authentication. You can create a custom page to replace the default login page, but note that the Java EE container expects a form that relies on the j_security_check mechanism to handle user-submitted `j_username` and `j_password` input.

  If you want to specify a custom page that uses ADF Faces components, you must use `/faces/` in the path (for example, `/faces/login.jsf`). You must also write additional code in a managed bean to handle login attempts. The steps for creating a custom login page and supporting code are described in the slides that follow.

  Another approach for implementing a custom page is to embed the HTML form with `j_security_check`, `j_username`, and `j_password` within a JSF page. You can do this by using the `f:verbatim` tag to enclose the HTML tags or by adding the HTML to the JSF Facelets page. This approach enables you to include the login form without writing extra code, but it does not integrate the login request with the JSF life cycle (so there is no partial login submit). Furthermore, skinning cannot be applied to the login fields at run time.

Creating a Custom Login Page That Uses ADF Faces components

To create a custom login page:

1. Create a managed bean that handles login attempts.
2. Create a JSF login page that uses ADF Faces components.
3. Configure ADF Security to use form-based authentication.

The default login form that is generated for you when you run the Configure ADF Security wizard is provided as a convenience for testing your application in JDeveloper. The default form does not enable you to use ADF Faces components to match the user interface of the application. You can replace the default form with a custom login page that uses ADF Faces components. The page that you create must explicitly call the ADF authentication servlet from a managed bean.

To explicitly handle user authentication:

1. Create a managed bean to handle login attempts by users.
2. Create a JSF login page that uses ADF Faces components.
3. Ensure that ADF Security is configured to use form-based authentication. If you want to use the custom page as the default login page, specify the path to the page when you configure authentication. For example, for the login page, specify `/faces/CustomLoginPage.jsf`.

These steps are covered in detail in the slides that follow.

Step 1: Creating a Managed Bean for Login

To create a login bean:

1. Create a Java class and register it as a managed bean in the `adfc-config.xml` file.
2. In the Java code, do the following:
   - Add two private fields, such as `_username` and `_password`, and generate accessors.
   - Add a `doLogin()` method that handles login attempts.
   - Add a `redirect()` method that redirects the HTTP response to a specific URL.
   - Add a `showError()` method that handles errors.

ORACLE

Before you create a login page that contains ADF Faces components, you must create a managed bean to handle login attempts. This slide describes how to create a login bean that handles authentication programmatically by using the Servlet 3.0–specific API.

To create the login bean, create a Java class and register it in the `adfc-config.xml` file as a managed bean in request scope. In the code for the Java class, add the following methods:

- **`doLogin()`:** Handles authentication (After successful login, the method invokes the ADF authentication servlet to redirect to a landing page.)
- **`redirect()`:** Redirects the HTTP response to a specific URL
- **`showError()`:** Handles errors

You do not need to create separate methods to handle the redirect and to show errors. You can implement the code directly in the `doLogin()` method. However, creating separate methods enables you to encapsulate the code for reuse.

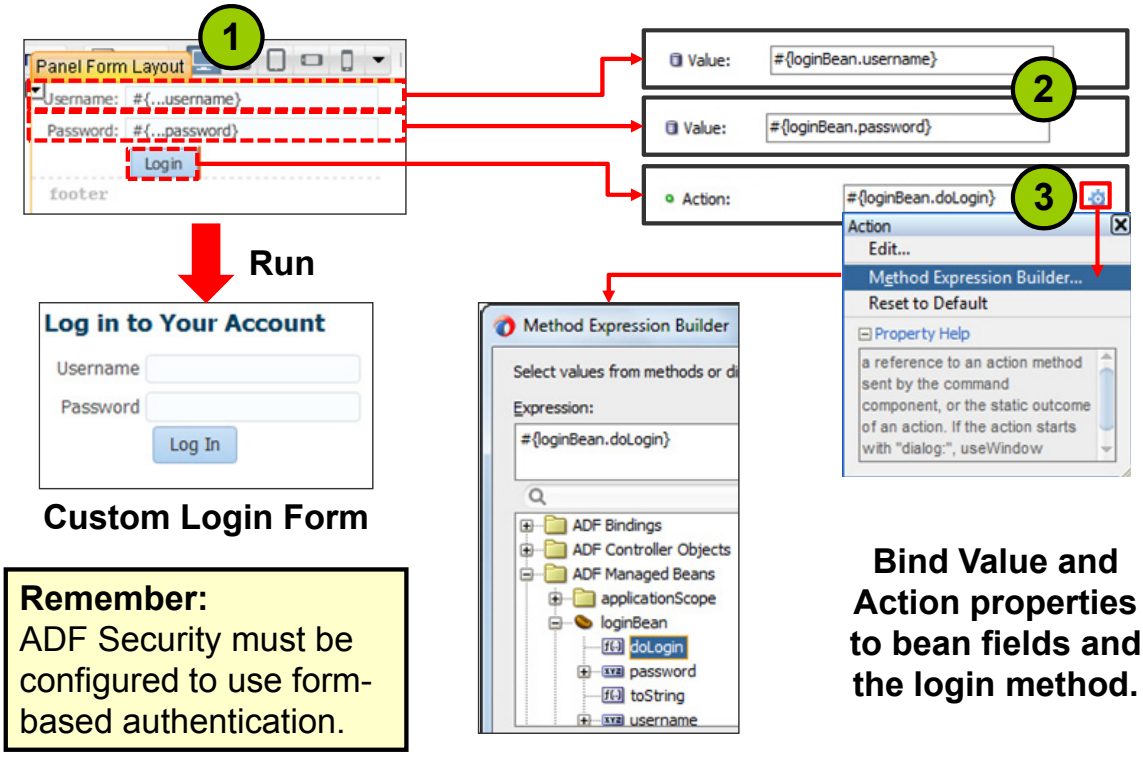
(Optional) You can add a `logout()` method if you want to trigger logout programmatically from a task flow method call activity.

The following example code shows how to implement the `doLogin()` method. The example code calls additional methods—`redirect()` and `showError()`—that are not shown in the example. The full example code for implementing login functionality in a managed bean is available in *Developing Fusion Web Applications with Oracle Application Development Framework:*

http://docs.oracle.com/middleware/1212/adf/ADFFD/adding_security.htm#BABDEICH

```
public String doLogin() {
   FacesContext ctx = FacesContext.getCurrentInstance();
   if (_username == null || _password == null) {
       showError("Invalid credentials",
             "An incorrect username or password was specified.",
             null);
   } else {
       ExternalContext ectx = ctx.getExternalContext();
       HttpServletRequest request =
         (HttpServletRequest)ectx.getRequest();
       try {
           request.login(_username, _password);
           String loginUrl = ectx.getRequestContextPath() +
                   "/faces/welcome";
           redirect(loginUrl);
       } catch (ServletException fle) {
           showError("ServletException", "Login failed.
           Please verify the username and password and try again.",
           null);
       }
   }
   return null;
}
```

Step 2: Creating a Custom Login Page with ADF Faces Components



Remember:
ADF Security must be configured to use form-based authentication.

To create a simple login page that uses ADF Faces components:

1. Create a JSF page that uses the desired layout. Remember that you can use the Panel Form Layout component to lay out multiple input components (such as input fields) in columns in a form.
2. Add two input fields, username and password, to the page and then specify text labels. For each input text field, use the Expression Builder to set the Value property to the appropriate field in the managed bean. For example, set the Value property of the input text component for the user name to `#{loginBean.username}`. Similarly, set the Value property of the password field to `#{loginBean.password}`.
3. Add a button to the form. For the Action property, use the Expression Builder to specify an expression that calls the `doLogin()` method in the managed bean. Example: `#{loginBean.doLogin}`

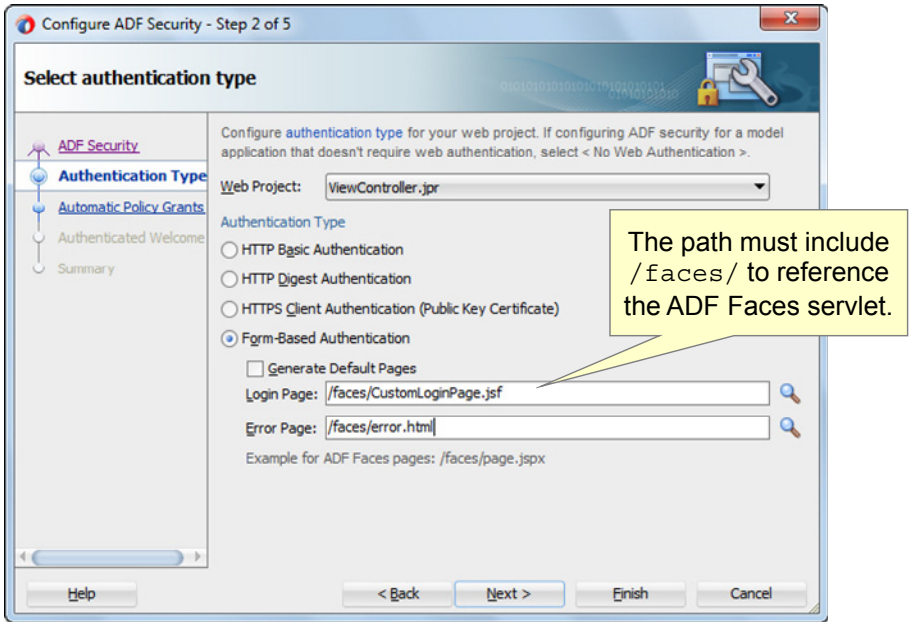
Make sure that the login page itself is accessible by granting `view` permission to `anonymous-role`.

Note: For information about implementing the login link, see *Developing Fusion Web Applications with Oracle Application Development Framework:*

http://docs.oracle.com/middleware/1212/adf/ADFFD/adding_security.htm#BABDEICH

Step 3: Configuring ADF Security to Use a Custom Login Page

Select Form-Based Authentication and specify the custom page:



Finally, make sure that ADF Security is configured to use form-based authentication and specify the custom login page. Note that the path to the login page includes `/faces/`. For example, the complete path is `/faces/CustomLoginPage.jsf`, even though the file does not physically reside in a directory called `faces`. You must include this reference to the ADF Faces servlet in the path so that the custom login page can be part of the ADF Faces life cycle.

Programmatically Accessing ADF Security

Is ADF security turned on?

```
if (ADFContext.getCurrent().getSecurityContext().isAuthorizationEnabled())
    { … }
```

Is the user logged on?

```
public boolean isAuthenticated() {
    return ADFContext.getCurrent().getSecurityContext().isAuthenticated(); }
```

Who is the user?

```
public String getCurrentUser() {
    return ADFContext.getCurrent().getSecurityContext().getUserName(); }
```

Is the user in a specified role?

```
public boolean isUserInRole(String role) {
  return ADFContext.getCurrent().getSecurityContext().isUserInRole(role); }
```

ORACLE

You can use methods of the ADF Security context to obtain information about users and roles. Because the enforcement of Oracle ADF Security can be turned on and off at the container level independently of the application, you should determine whether Oracle ADF Security is enabled before making permission checks. You can evaluate the `isAuthorizationEnabled()` method to achieve this.

It is not possible to check whether the user principal is null to determine whether the user has logged on or not, because it is either `anonymous` for unauthenticated users or the actual user name for authenticated users. You can use the `isAuthenticated()` method to determine whether the user has authenticated.

You can determine the current user name (either `anonymous` for unauthenticated users or the actual user name for authenticated users) with the `getUserName()` method.

You can use the `isUserInRole()` method to determine whether the user is a member of a specified role. It is a good idea to make user and role information available throughout the application by using session-scoped beans.

You can also access the ADF security context from ADF Business Components. For example, you can access the security context from a Groovy expression to set the value of a bind variable to the name of the authenticated user. This enables you to filter the query and return only data that belongs to the authenticated user.

Using the Expression Language to Extend Security Capabilities

You can integrate the Expression Language in two ways:

- By using built-in global security expressions:

```
<af:link action="accounts"
rendered="#{securityContext.userInRole['admin']}"
text="Manage Accounts"/>
```

- By using a security proxy bean:

```
<af:link action="accounts"
rendered="#{userInfo.admin}"
text="Manage Accounts"/>
```

Covered in a later slide

You can use expressions to resolve properties such as `Rendered`, `ReadOnly`, and `Disabled` at run time. The built-in global security expressions, which are shown in the first example in the slide, enable such access. Alternatively, you can write your own beans, as in the second example.

Using Global Security Expressions

Expression	Purpose
`#{securityContext.userName}`	User name of the authenticated user
`#{securityContext.userInRole ['role list']}`	Is the user in *any* of these roles?
`#{securityContext.userInAllRoles ['role list']}`	Is the user in *all* of these roles?
`#{securityContext. userGrantedPermission ['permission']}`	Does the user have this permission granted?
`#{securityContext.taskflowViewable ['target']}`	Does the user have view permission on the target task flow?
`#{securityContext.regionViewable ['target']}`	Does the user have view permission on the target region?

ORACLE

Global security expressions are available without installing extensions. You can use them on any page, even if the page does not have a page definition file.

Using a Security Proxy Bean

A managed bean can expose a Boolean property that the UI expressions can consume, as in the following example:

- You might have a `UserInfo` bean with a method that returns `true` if the user role is `admin`:

```
public boolean isAdmin() {
  return (ADFContext.getCurrent().getSecurityContext().isUserInRole("admin"));}
```

- You can then use the following expression in the rendered or disabled property of a component:

```
#{userInfo.admin}
```

Tip: Checking for multiple roles can be a problem. You could actually write many convenience methods.

A security proxy bean is a Java class that is usually stored in the session as a managed bean. This bean exposes security information in a form that the Expression Language can process, usually as a method called `is<something>`, which translates to a simple expression (as in the example shown in the slide).

However, when you must check for multiple combinations, the Expression Language becomes complex or you need many convenience methods. In addition, you must generate accessors for each role.

Summary

In this lesson, you should have learned how to:

- Explain the benefits of secure applications
- Describe the security aspects of an ADF Business Components application
- Implement ADF security
  - Authentication
  - Authorization
    - In the data model
    - In the UI for task flows and pages
- Access security information programmatically
- Use the Expression Language to extend security capabilities

ORACLE

Practice 18 Overview: Implementing Security

This practice covers the following topics:

- Configuring ADF security
- Creating test users and mapping users to application roles
- Granting application roles access to ADF resources
- Retrieving user information from the security context

19

Deploying ADF BC Applications

Objectives

After completing this lesson, you should be able to:

- Describe the deployment process
- Use JDeveloper to create deployment profiles and configure deployment options
- Change the context root for an application
- Deploy an application from JDeveloper

ORACLE

Lesson Aim

This lesson describes how to deploy an ADF Business Components application by using JDeveloper.

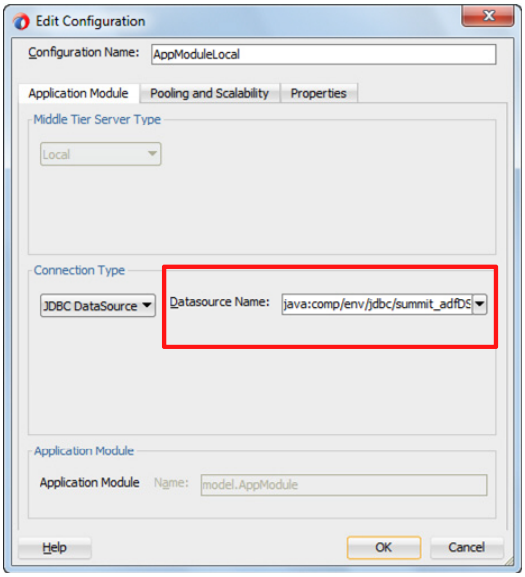
Basic Deployment Steps

1. Prepare application for deployment.
   - Example: Alter database connection.
2. Create deployment profiles.
3. Specify deployment profile properties as necessary.
   - Modify context root.
   - Specify additional deployment descriptors.
4. Prepare server for deployment.
5. Deploy the deployment profiles:
   - Create EAR file and deploy to WLS directly.
   - Create EAR file only.

Preparing an Application for Deployment

To prepare for deployment, it may be necessary to modify the application to use shared resources such as libraries, database connections, and so on.

By default, the database connection that you create when building ADF business components is created in the Application Module configuration file as a JDBC data source.



When you move from a development environment to production, you may need to modify the application to use shared resources: for example, pointing the application to a production database rather than to the database connection used during development and testing. You can change a database connection in the Model project properties under the Business Components node. The name of this connection will be defined in the application module configuration file (`bc4j.xcfg`) of type JDBC Datasource. The connection details can be different between different environments, but if the JNDI path is mapped correctly, the data source will be accessible.

You can map the JNDI data source that is specified in the Application Module configuration to another data source that is defined in the WLS domain. You do this in the `web.xml` and `weblogic.xml` deployment descriptors. You provide a name that is used in your Java code, the name of the resource as bound in the JNDI tree, and the Java type of the resource, and you indicate whether security for the resource is handled programmatically by the servlet or from the credentials associated with the HTTP request.

For example, your servlet code might refer to a data source by the name `myDataSource`:

```
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup
                                                 ("myDataSource");
```

In the `web.xml` file, the following entry would be the resource name, where the value of `<res-ref-name>` is the name that is specified in the application module configuration (`myDataSource`):

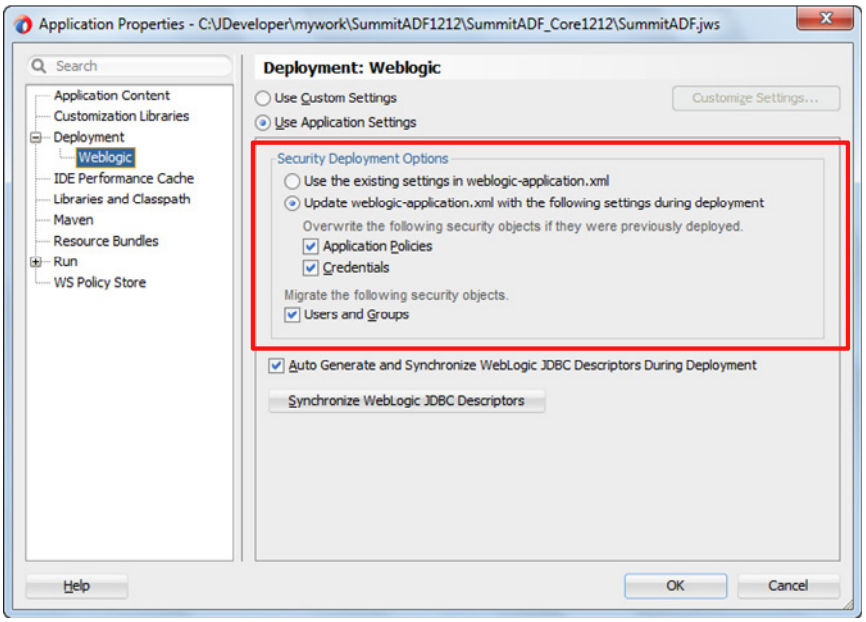
```
<resource-ref>
 . . .
    <res-ref-name>myDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>CONTAINER</res-auth>
. . .
</resource-ref>
```

In the `weblogic.xml` file, you would map the name that is specified in the `web.xml` file to the name of the actual data source in the production environment (`accountDataSource`):

```
<resource-description>
    <res-ref-name>myDataSource</res-ref-name>
    <jndi-name>accountDataSource</jndi-name>
</resource-description>
```

Setting Security Deployment Options

Set deployment options in Application Properties to specify how security settings are deployed:



You can alter the way JDeveloper handles the deployment of security settings by specifying security deployment options in the Application Properties window under Deployment > WebLogic.

By default, when you deploy to Oracle WebLogic Server, JDeveloper overwrites the security repositories at the domain level. Also, each time you run an application, JDeveloper by default overwrites the security repositories in Integrated WebLogic Server.

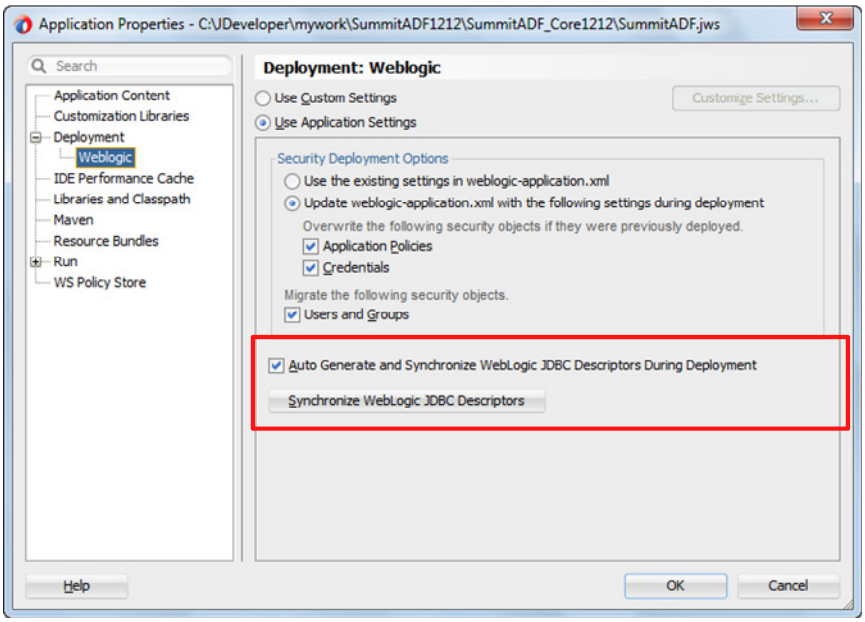
You can preserve existing application policies and credentials by deselecting the appropriate check boxes under Security Deployment Options. When you deselect these options, JDeveloper merges new policies and credentials into the domain-level stores; existing entries are preserved.

To prevent JDeveloper from making changes to the security repositories, select the "Use the existing settings in weblogic-application.xml" check box.

You can also control whether the identity store portion of the `jazn-data.xml` file is migrated to the domain-level identify store. When “Users and Groups” is selected, each time you run the application, JDeveloper migrates new user identities that you created for test purposes and updates existing user passwords in the embedded LDAP server that Integrated WebLogic Server uses as its identity store.

Synchronizing JDBC Deployment Descriptors

JDeveloper automatically generates JDBC descriptors during deployment, or you can generate them on demand:



Deployment descriptors are files that describe deployment options, such as configuration or security, for an application or module. Some common deployment descriptors include `application.xml` (used for enterprise applications) and `web.xml` (used for web applications).

You can configure JDeveloper to automatically generate and synchronize JDBC descriptors during deployment, or you can generate them on command. If the auto generate option is selected, when you run the application on Integrated WebLogic Server or deploy it to Oracle WebLogic Server, JDeveloper does the following to ensure that the application runs correctly using application-level data sources:

- Generates a file called *`connection-name`*`-jdbc.xml` for each connection in the application resources, and sets the indirect password attribute. Upon deployment, JDeveloper determines the JDBC connection password from the username in the connection file, and populates the JDBC connection password.
- Updates `weblogic-application.xml` to add each *`connection-name`*`-jdbc.xml` as a module
- Updates `web.xml` (if it exists) to have a resource reference to each JDBC JNDI name

When the auto generate option is not selected, only the `web.xml file` (if it exists) is updated when the application runs. This means that you can manually set up JDBC files in the application, or use global data sources on the server.

If you are deploying to an EAR file, you must define passwords for the data sources on the server before deploying the application.

Synchronizing JDBC Deployment Descriptors

After JDBC descriptors are synchronized, JDeveloper creates a connection file (`hr-jdbc.xml`) with information for deployment:

```
<jdbc-data-source...>
  <name>hr</name>
    <url>jdbc:oracle:thin:@host.com:1521:ORCL</url>
    <driver-name>oracle.jdbc.OracleDriver</driver-name>
    ...
      <jndi-name>jdbc/hrDS</jndi-name>
      <scope>Application</scope>
</jdbc-data-source>
```

JDeveloper also adds the connection file to `weblogic-application.xml`:

```
<module>
  <name>hr</name>
  <type>JDBC</type>
  <path>META-INF/hr-jdbc.xml</path>
</module>
```


The slide shows a connection file, `hr-jdbc.xml`, that is created for a connection called `hr`. It also shows the corresponding <module> entry in the `weblogic-application.xml` file.

Creating Deployment Profiles

Deployment profiles specify how files are packaged for deployment. Deployment profiles are artifacts that:

- Can be at the application level or project level
- Are created automatically by JDeveloper depending on the application template used. For Fusion web applications, the profiles are the following:
  - Model project JAR profile for ADF Business Components
  - ViewController project WAR profile for ADF/JSF components
  - Application EAR profile for the packaged application
- Have dependencies on one another
  - WAR profiles can have dependencies on JAR files.
  - EAR profiles can have dependencies on WAR files.

ORACLE

A deployment profile defines the way files are packaged into the archives that are deployed into the target environment. The deployment profile:

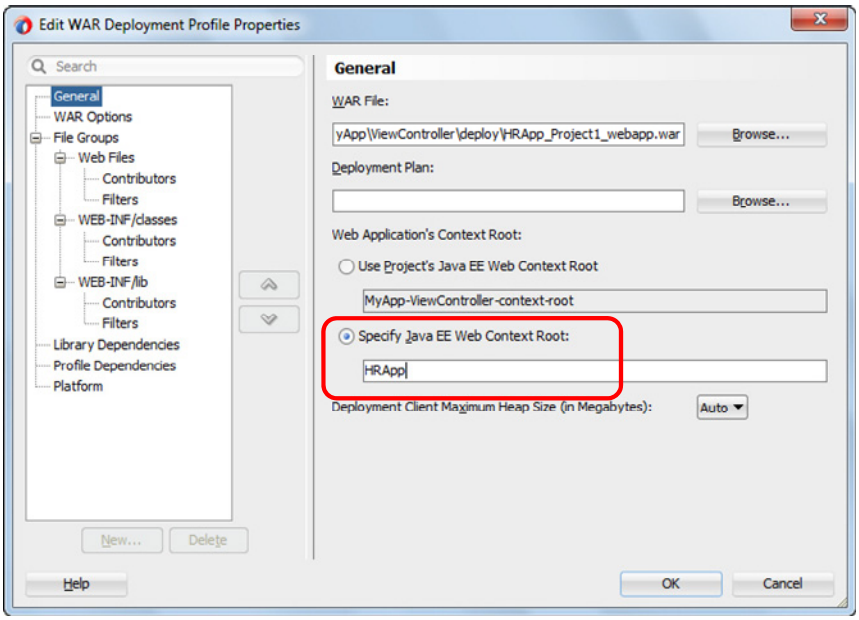
- Specifies the format and contents of the archive file to be created
- Lists the source files, deployment descriptors, and other auxiliary files to be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You can define and store deployment profiles at the application or project level, giving you more flexibility and enabling direct referencing and sharing. For example, you could create a WAR (Web Archive) profile for projects associated with the View and Controller layers and a Business Components JAR (Java Archive) for the model project. You could then create an application-level EAR (Enterprise Archive) profile that assembles the model and view-controller profiles for deployment. Additionally, Fusion web applications that include customizations deploy using a MAR (Metadata Archive) file - a compressed archive of selected metadata. For further information about deploying Fusion web applications with MDS customizations, see *Developing Fusion Web Applications with Oracle Application Development Framework.*

JDeveloper creates deployment profiles for you automatically depending on the template used to create the application. You can also create deployment profiles by using the New Gallery, or by using application properties or project properties.

Changing the Context Root for an Application

To change the default URL of a web application, modify the default context root in the WAR deployment profile properties:



The default URL for a Fusion web application is based on the application and project name of the application, for example, http://*hostname:port*/faces/*<ApplicationName>-<ProjectName>*-context-root. To change this, modify the WAR deployment profile properties to specify a different context root. For example, if you specify HRApp for the context root of a view-controller project, the URL would look something like the following:
`http://127.0.0.1:7101/HRApp/faces/Welcome.jsf`

In addition to changing the context root in the WAR file, you can also change it during deployment in the WebLogic Server Console or Oracle Fusion Middleware Control.

At design time, you can specify the Java EE application name and context root for each project in the Java EE Application page of the project properties.

Preparing the Oracle WebLogic Server (WLS) for Deployment

The Integrated WebLogic Server is preconfigured for Fusion ADF applications. To ensure that the deployment server is configured for Fusion ADF applications in the same way as WebLogic Server:

- Create a domain (optional).
- Install the ADF runtime to the domain.
- Create a global JDBC data source.
- Set the Oracle WebLogic Server credential store overwrite setting (if required):
  `-Djps.app.credential.overwrite.allowed=true`

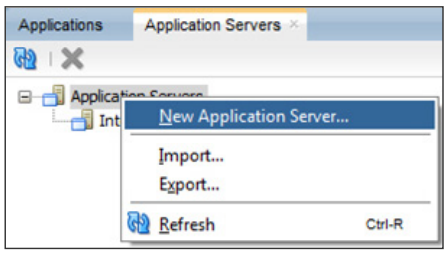
To prepare the application server for deployment, some steps need to be performed on the server, which may include creating a domain, installing the ADF Runtime libraries to the domain, creating a data source, and setting the credential store overwrite option to `true` so that database credentials will be copied to the server at deployment. For more information, see the “Administering Oracle ADF Applications” guide of the Fusion Middleware Documentation library.

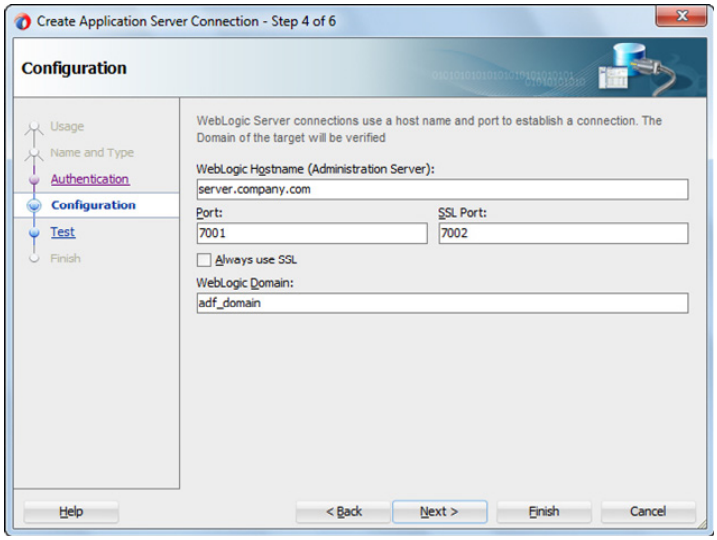
Options for Deploying an Application

- Deploy application deployment profiles to files.
  1. An application EAR file is created with nested WAR/JAR files.
  2. The EAR file is copied to the `<ApplicationDirectory>/deploy` directory.
  3. Files can then be shared with administrators for deployment via Enterprise Manager or deployment scripts.
- Deploy application deployment profiles directly to a server.
  - This is less common because it requires administrator access to the server.
- Use `ojdeploy` to generate the EAR file ("headless" deployment).
  - Run via an Ant script or the Continuous Integration solution.

Creating a Connection to an Application Server

To deploy an EAR file directly to a server from JDeveloper, create a connection in the Application Servers window and specify authentication and domain details:



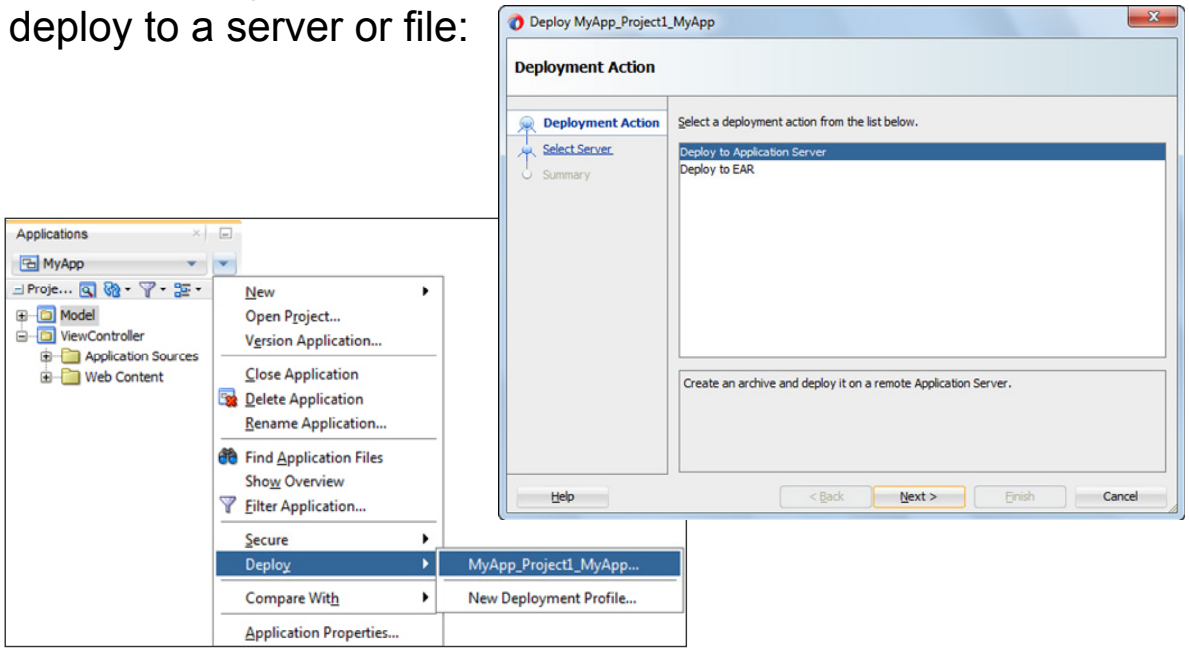


To deploy an EAR file directly to a server, create a connection to the server in the Application Servers window. Creating an application server connection in this way requires entering authentication details in the form of the username and password for the administrator account, and also specifying the hostname, port, and domain for the server. You can create the connection from either the Application Servers or Resources window, but creating the connection from the Application Servers window gives you more flexibility for dealing with remote servers.

After this step is complete, the application server will show up in the list of servers available for deployment.

Deploying the Application

Select Deploy > ProfileName from the Applications menu to deploy to a server or file:



To deploy an application, select Deploy from the Applications menu and select the profile name. Use the deployment action dialog to select whether the application will be deployed to a defined server (either the Integrated WLS or those created via the Resources window), or to deploy to an EAR, which will create the EAR in the /deploy directory of the application. Developers can then determine how the application will actually be deployed to the server. Typically, the file would be shared with the server administrator and deployed via Enterprise Manager. As previously discussed, teams may also choose to create scripts using Ant or other tools to perform this step automatically. For more information on deploying using scripts, see the “Administering Oracle ADF Applications” guide of the Fusion Middleware Documentation.

After the application is deployed, it can be accessed via: `http://<host>:port/<context root>/faces/<view_ id>` where `view_id` is the name of the default .jsf page or view activity.

Using `ojdeploy` to Build Files for Deployment

`ojdeploy` is a post-compilation processor that can be used to build a JAR file for a project or an EAR file for an application:

- Can be called via the command line or Ant
- Requires a full JDeveloper installation to operate
  - Generates necessary deployment files
  - Validates ADF metadata files

```
// Deploying a JAR file from the Model project
ojdeploy -workspace /usr/jdoe/MyApp/MyApp.jws
         -project Model
         -profile MyAppLib

// Deploying an EAR from the Application workspace
ojdeploy -workspace /usr/jdoe/MyApp/MyApp.jws
         -profile MyAppEar
```

ORACLE

`ojdeploy` is a post-compilation processor that you can use to build a JAR file for a project or an EAR file for an application without requiring you to run the JDeveloper IDE. This tool is especially useful when you need to use a batch file or script to deploy existing projects or applications.

Summary

In this lesson, you should have learned how to:

- Describe the deployment process
- Use JDeveloper to create deployment profiles and configure deployment options
- Change the context root for an application
- Deploy an application from JDeveloper