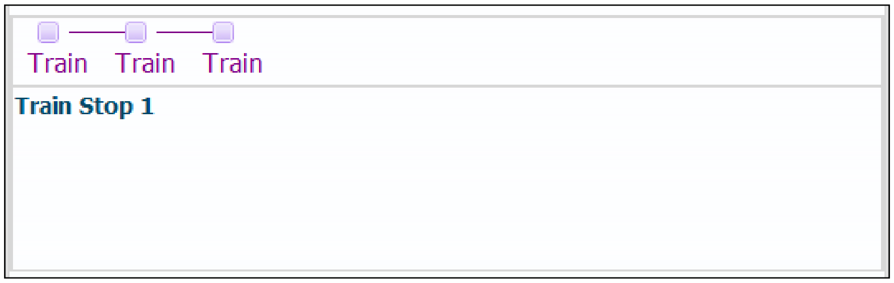
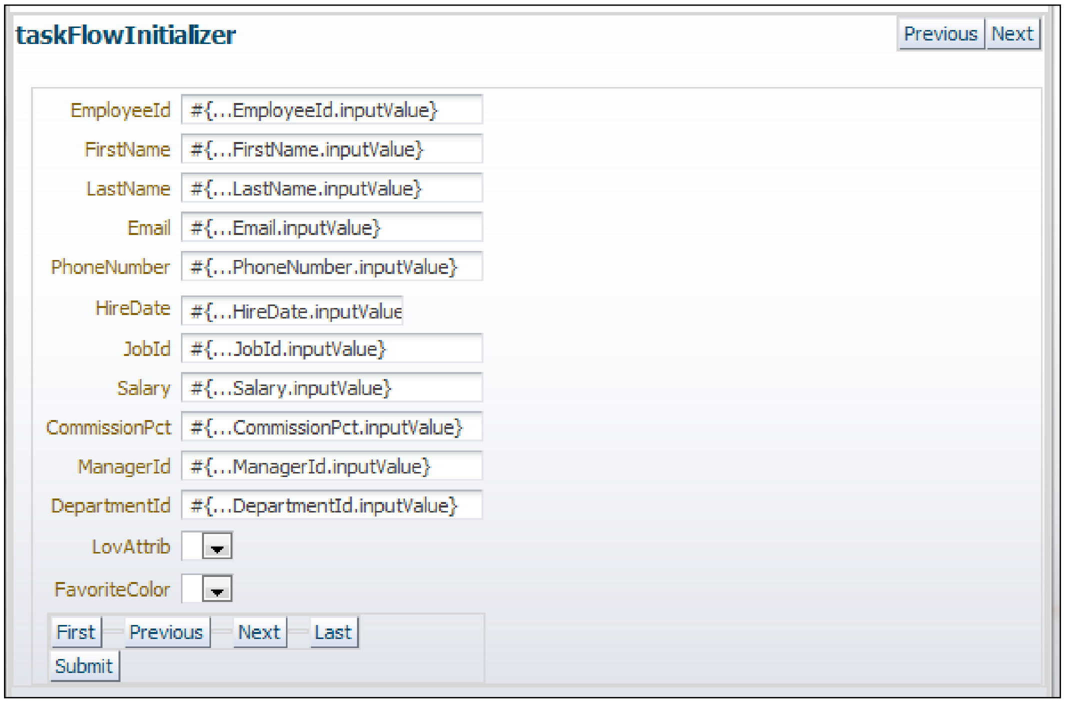
Each page should look similar to the following one:



14. Finally, modify the `taskFlowInitializerView.jspx` JSF page by adding two extra buttons called **Previous** and **Next**. Using the **Property Inspector**, set their **Action** attributes to **previous** and **next** respectively. To ensure that validation will not be raised on the page, ensure that for both buttons the `Immediate` attribute is set to `true`. The `taskFlowInitializerView.jspx` page should look similar to the following screenshot:


taskFlowInitializer

EmployeeId	#{…EmployeeId.inputValue}
FirstName	#{…FirstName.inputValue}
LastName	#{…LastName.inputValue}
Email	#{…Email.inputValue}
PhoneNumber	#{…PhoneNumber.inputValue}
HireDate	#{…HireDate.inputValue
JobId	#{…JobId.inputValue}
Salary	#{…Salary.inputValue}
CommissionPct	#{…CommissionPct.inputValue}
ManagerId	#{…ManagerId.inputValue}
DepartmentId	#{…DepartmentId.inputValue}
LovAttrib	
FavoriteColor	

How it works...

In step 1, we created a bounded task flow called `trainTaskFlow`. We indicated that the task flow will implement a train by ensuring that the **Create Train** checkbox in the **Create Task Flow** dialog was selected. Then, in step 2, we droped four view activities to the task flow, called train stops in train terminology, each one being part of the train. Notice how JDeveloper connects these train stops with a dotted line indicating that they are part of the train.

In steps 3 through 6, we combined a method call activity, called `prepare`, with the `trainStop3` view activity in a single train stop. The way we did this was by wiring the `prepare()` method call activity via a control flow rule to the `trainStop3` view activity (step 4). The `prepare()` method call activity is wired to a wildcard control flow rule called `callPrepareBeforeStop3` (in step 5). In order to ensure that the `prepare()` method call activity and the `trainStop3` view activity are combined in a single train step, we have set the outcome of the `trainStop3` train stop to `callPrepareBeforeStop3` (step 6). This ensures that at runtime the `prepare()` method call activity is executed before the `trainStop3` view activity together in a single train stop.

In steps 7 and 8, we have modified the `taskflowInitializer` task flow, which was originally developed in the *Using a task flow initializer to initialize task flow* recipe in this chapter, so that it can be used as part of the train. In particular, we added two task flow return activities, one for navigating backwards on the train and another one for navigating forward. We wired the task flow return activities to the existing `taskFlowInitializerView` view activity. Based on the specific outcomes (`previous` or `next`) originating from the `taskFlowInitializerView` view activity (see step 14), navigation on the train can be accomplished.

Once these changes were made to the `taskflowInitializer` task flow, we are able to complete the `trainTaskFlow` train, by first adding it to the train as a task flow call activity (in step 9) and then wiring it to the train by adding the relevant control flow cases (step 11). In step 10, we just adjusted the task flow call train stop position in the train.

The rest of the recipe steps (12 through 14) deal with the creation and modification of the JSF pages related to the view activities participating in the train task flow. In steps 12 and 13, we created the JSF pages corresponding to the four view activity train stops. In each page, we added an `af:train` ADF Faces component to allow for the navigation over the train. Finally, in step 14, we made the necessary changes to the existing `taskFlowInitializerView.jspx` page to be able to hook it to the train. Specifically, we added two buttons, called `Previous` and `Next`, and we set their actions appropriately (to `previous` and `next` respectively), to allow for the `taskflowInitializer` task flow to return to the calling `trainTaskFlow` task flow (see step 8).

To run the train, right-click on the `trainTaskFlow` task flow in the **Application Navigator** and select **Run** or **Debug**.

There's more...

Each train stop can be dynamically configured at runtime using EL expressions to allow for a number of options. These options are available in the **Property Inspector** for each train stop selected in the train task flow during development. They are briefly explained as follows:

- `Outcome`: Used in order to combine multiple activities preceding the view or task flow call activity in a single train stop. This was demonstrated in step 6 where we combined a method call activity with a view activity in a single train stop.
- `Sequential`: When set to `false`, the train stop can be selected even though a previous train stop has not been visited yet.
- `Skip`: When set to `true`, the train stop will be skipped. At runtime a skipped train stop will be shown as disabled and you will not be able to select it.
- `Ignore`: When set to `true`, the train stop will not be shown.

By dynamically setting these attributes at runtime, you can effectively create multiple trains out of a single train definition.

For more information about train task flows, check out the section *Using Train Components in Bounded Task Flows* in the *Fusion Developer's Guide for Oracle Application Framework* which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*
- *Using a task flow initializer to initialize task flow*, in this chapter

7 Face Value: ADF Faces, JSF Pages, and User Interface Components

In this chapter, we will cover:

- Using an af:query component to construct a search page
- Using an af:pop-up component to edit a table row
- Using an af:tree component
- Using an af:selectManyShuttle component
- Using an af:carousel component
- Using an af:poll component to periodically refresh a table
- Using page templates for pop-up reuse
- Exporting data to a client file

Introduction

ADF Faces Rich Client Framework (ADF RC) contains a plethora (more than 150) of AJAX-enabled JSF components that can be used in your JSF pages to realize **Rich Internet Applications (RIA)**. ADF RC hides the complexities of using JavaScript, and declarative partial page rendering allows you to develop complex pages using a declarative process. Moreover, these components integrate with the **ADF Model layer (ADFm)** to support data bindings and model-driven capabilities, provide support for page templates, and reusable page regions. In JDeveloper, ADF Faces components are made available through the **Component Palette**. For each component, the available attributes can be manipulated via the **Property Inspector**.

Using an af:query component to construct a search page

The `af:query` (or query search form) ADF Faces user interface component allows for the creation of search forms in your ADF Fusion web application. It is a model-driven component, which means that it relies on the model definition of named view criteria. This implies that changes made to the view criteria are automatically reflected by the `af:query` component without any additional work. This fact, along with the JDeveloper's declarative support for displaying query results in a table (or tree table) component, makes constructing a search form a straightforward task.

In this recipe, we will cover the creation of a query search form and the display of search results in a table component.

Getting ready

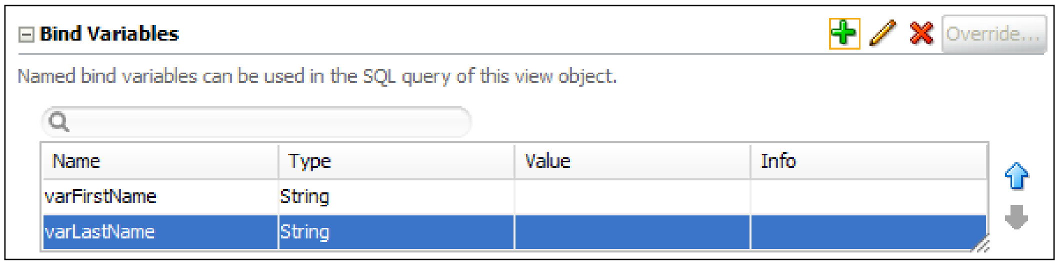
You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. We will be using the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspace*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundation*s.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `HRComponents` workspace and locate the `Employees` view object in the **Application Navigator**. Double-click on it to open its definition.
2. Go to the **Query** section and add two bind variables of type **String**, named `varFirstName` and `varLastName`. Ensure that the **Required** checkbox in the **Bind Variable** dialog is unchecked for both these variables. Also, in the **Control Hints** tab of the **Bind Variable** dialog, ensure that the **Display Hint** is set to **Hide**.

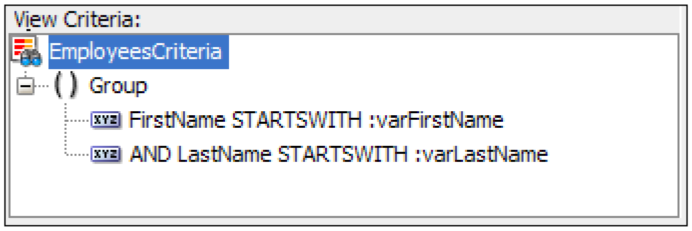


Bind Variables

Named bind variables can be used in the SQL query of this view object.

Name	Type	Value	Info
varFirstName	String		
varLastName	String		

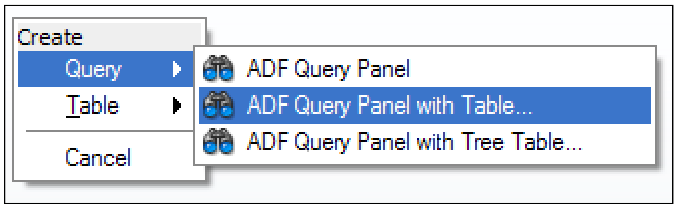
3. While in the **Query** section, add named view criteria for the `FirstName` and `LastName` attributes using the bind variables `varFirstName` and `varLastName` respectively. For both criteria items, ensure that the **Ignore Case** and **Ignore Null Values** checkboxes are checked and that the **Validation** is set to **Optional**.



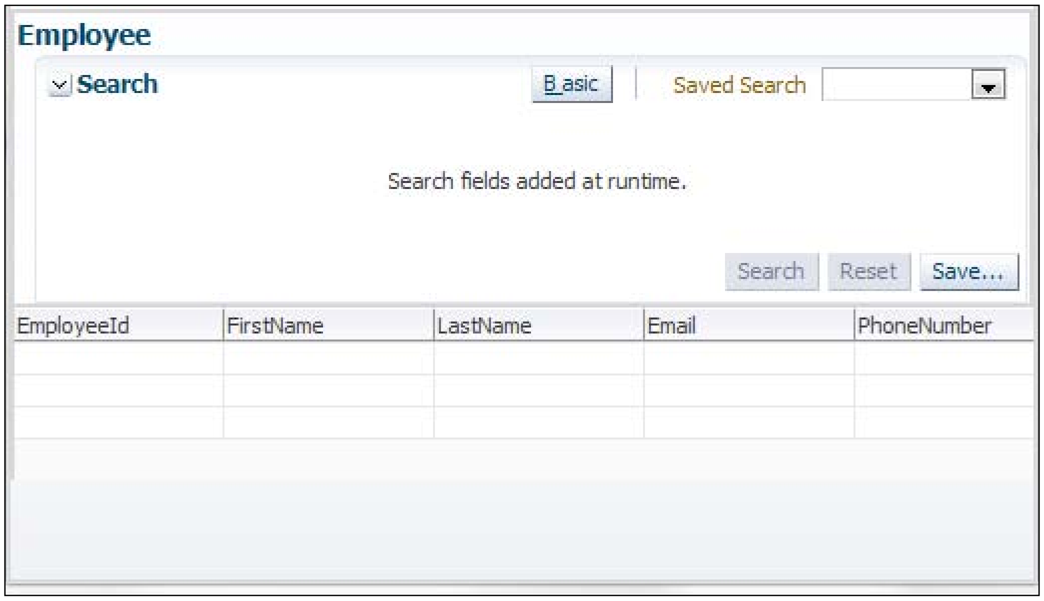
View Criteria:
EmployeesCriteria
( ) Group
FirstName STARTSWITH :varFirstName
AND LastName STARTSWITH :varLastName

4. Rebuild and redeploy the `HRComponents` workspace as an ADF Library JAR.
5. Open the `MainApplication` workspace and ensure that the `HRComponents` and `SharedComponents` ADF Library JARs are added to the ViewController project.
6. Create a bounded task flow called `queryTaskFlow` and add a view activity called `queryView`. Ensure that the task flow is not created with page fragments.
7. Double-click on the `queryView` activity in the task flow to bring up the **Create JSF Page** dialog. Proceed with creating a **JSP XML** page called `queryView.jspx` using any one of the pre-defined layouts.
8. In the **Application Navigator**, expand the **Data Controls** section and locate the `EmployeesCriteria` view criteria under the **HrComponentsAppModuleDataControl** | **Employees** | **Named Criteria** node. Drag-and-drop the `EmployeesCriteria` view criteria onto the page.

9. From the **Create** context menu, select **Query** | **ADF Query Panel with Table...**.


Create
Query
Table
Cancel
ADF Query Panel
ADF Query Panel with Table...
ADF Query Panel with Tree Table...

10. JDeveloper will bring up the **Edit Table Columns** dialog. Click **OK** to accept the default settings for now. When previewing the page in the browser, you should see something similar to the following screenshot:



11. In the **Structure** window, locate and select the **af:table** component. Then, in the **Table Property Inspector**, click on the **Edit Component Definition** button (the pen icon). In the **Edit Table Components** dialog, adjust the table definition by removing any columns indicating row selection and enabling sorting or filtering. Adjust the table's width by specifying the width in pixels in the **Style** section of the **Table Property Inspector**.

How it works...

In steps 1 through 3, we have updated the `Employees` view object, which is part of the `HRComponents` workspace, by adding named view criteria to it. Based on the earlier mentioned criteria, we subsequently (in steps 8 and 9) associate an `af:query` component to create the search page. The view criteria comprises two criteria items, one for the employee's first name and another for their last name. We have based the criteria items on corresponding bind variables created in step 2. Note the view criteria item settings that are used in step 3. For both name criteria the search is case-insensitive; null values are ignored, which means that the search will yield results when no data is specified; and that both are optional. Also note that we have based both of the criteria items on the **Starts with** operation and that the `AND` conjunction is used for the criteria items.

In steps 4 and 5, we redeployed the `HRComponents` workspace to an ADF Library JAR, which we then add to the `MainApplication` ViewController project. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project.

In step 6, we created a bounded task flow called `queryTaskFlow` and added a single view activity. Then (in step 7), we created a JSF page for the view activity.

To add search capability to the page, we have located the view criteria added earlier to the `Employees` view object. We have done this by expanding the **Named Criteria** node under the `Employees` view object node of the `HrComponentsAppModuleDataControl` data control. This data control is added to the list of available data controls once we add the `HRComponents` ADF Library JAR to our workspace in step 5. JDeveloper supports the creation of databound search pages declaratively by presenting a context menu of choices when dropping view criteria onto the page, as in step 9. From the context menu that is presented, we had chosen **ADF Query Panel with Table...**, which created a query panel with an associated results table. If you take a look at the page's source, you will see a code snippet similar to the following:

```
<af:panelGroupLayout layout="vertical" ...
  <af:panelHeader ...
    <af:query value="#{bindings.EmployeesCriteriaQuery.
      queryDescriptor}"
    queryListener="#{bindings.EmployeesCriteriaQuery.processQuery}"
    queryOperationListener="#{bindings.EmployeesCriteriaQuery
      .processQueryOperation}"
    resultComponentId="::resId1" ...
  </af:panelHeader>
  <af:table id="resId1" value="#{bindings.Employees.collectionModel}"
    var="row" ....
  <af:column ...
  </af:table>
</af:panelGroupLayout>
```

As you can see, JDeveloper wraps the `af:query` and `af:table` components in an `af:panelGroupLayout`, arranged vertically. Also, note that this simple drag-and-drop of the view criteria onto the page in the background creates the corresponding search region and iterator executables, along with the tree binding used by the `af:table` component and the necessary glue code to associate the search region executable and tree binding to the iterator. It also associates the `af:query` component with the table component that will be used to display the search results. This is done by specifying the table component's identifier (`resId1` in the previous sample code) in the `af:query` `resultComponentId` attribute.

Finally, notice in steps 10 and 11 some of the possibilities that are available in JDeveloper to declaratively manipulate the table, either through the **Edit Table Columns** dialog or the **Property Inspector**.

There's more...

In addition to the `af:query` component, ADF Faces supports the creation of model-driven search pages using the `af:quickQuery` (Quick Query) component. You can create a search page using an `af:quickQuery` by dragging the **All Queriable Attributes** item under the view object **Named Criteria** node in the **Data Controls** window and dropping it on the page and selecting any of the **Quick Query** options in the **Create** context menu. The **All Queriable Attributes** node represents the implicit view object criteria that are created for each `Queryable` view object attribute.

For information about creating databound search pages, refer to the *Creating ADF Databound Search Forms* chapter in the *Fusion Developer's Guide for Oracle Application Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*

Using an af:pop-up component to edit a table row

An `af:popup` component can be used in conjunction with an `af:dialog` to display and edit data within a page on a separate pop-up dialog. The pop-up is added to the corresponding JSF page, and can be raised either declaratively using an `af:showPopupBehavior` or programmatically by adding dynamic JavaScript code to the page.

In this recipe, we will expand the functionality introduced in the previous recipe, to allow for the editing of a table row. The use case that we will demonstrate is to raise an edit form inside a pop-up dialog by double-clicking on the table row. The changes made to the data inside the dialog are carried over to the table.

Getting ready

This recipe relies on having completed the *Using an af:query component to construct a search page* recipe in this chapter.

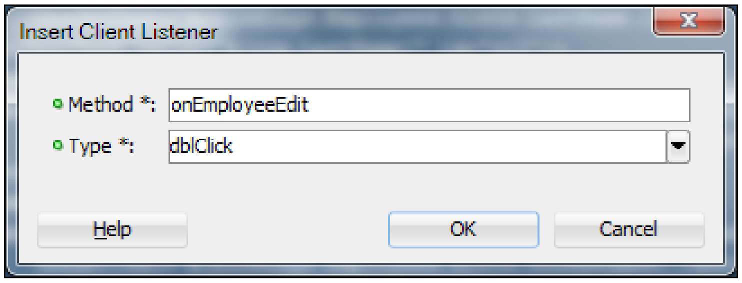
How to do it...

1. Open the `MainApplication` workspace. Locate the `queryTaskFlow` in the **Application Navigator** and double-click on it to open it.
2. Go to the task flow **Overview** | **Managed Beans** section and add a managed bean called `QueryBean`. Specify a class for the managed bean, then use the **Generate Class** selection in the **Property Menu**—located next to the **Managed Bean Class** property in the **Property Inspector** to create the managed bean class.
3. Double-click on the managed bean Java class in the **Application Navigator** to open it in the Java editor. Add the following method to it:

```
public void onEmployeeEdit(ClientEvent clientEvent) {
  FacesContext facesContext = FacesContext.getCurrentInstance();
  ExtendedRenderKitService service =
    Service.getRenderKitService(facesContext,
    ExtendedRenderKitService.class);
  service.addScript(facesContext,
    "AdfPage.PAGE. findComponentByAbsoluteId(
    'editEmployee').show();");
}
```

4. Locate the `queryView.jspx` JSF page in the **Application Navigator** and double-click on it to open it.
5. Locate and select the **af:table** component in the **Structure** window. Right-click on it and select **Insert Inside af:table** | **ADF Faces** | **Client Listener**.

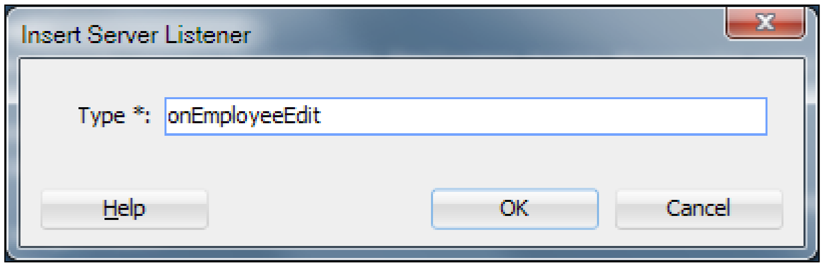
6. In the **Insert Client Listener** dialog, enter `onEmployeeEdit` for the **Method** and select **dblClick** for the **Type**.


Insert Client Listener

Method *: onEmployeeEdit

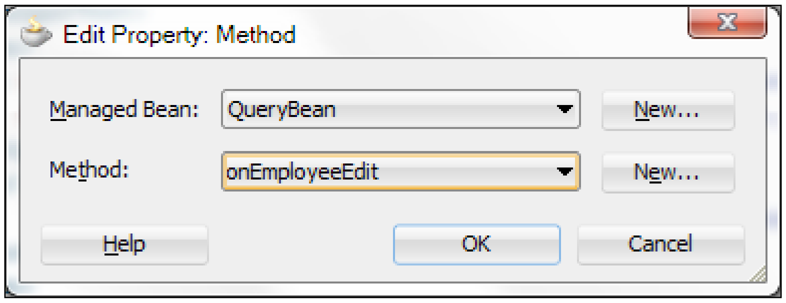
Type *: dblClick

7. Right-click on the `af:table` component in the Structure window and this time select **Insert Inside af:table | ADF Faces | Server Listener**. For the **Type** field in the **Insert Server Listener** dialog type `onEmployeeEdit`.


Insert Server Listener

Type *: onEmployeeEdit

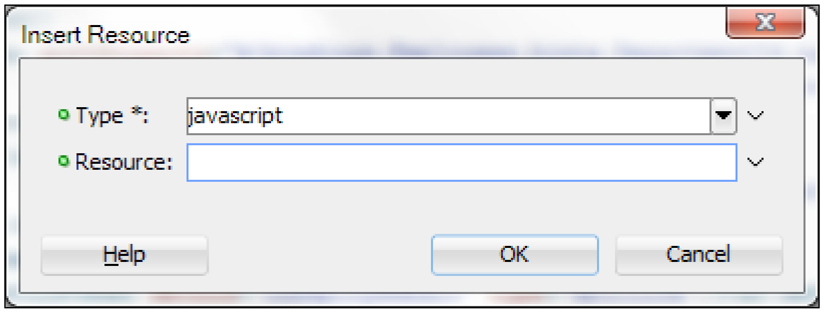
8. Use the **Property Inspector** to set the `af:serverListener` **Method** property to the **onEmployeeEdit** method of the **QueryBean**.


Edit Property: Method

Managed Bean: QueryBean

Method: onEmployeeEdit

9. Right-click on the `af:document` in the **Structure** window and select **Insert Inside af:document | ADF Faces...**. Then select **Resource** from the **Insert ADF Faces Item** dialog.
10. In the **Insert Resource** dialog, specify **javascript** for the **Type** and click **OK**.


Insert Resource

Type *: javascript

Resource:

11. Locate the `af:resource` in the `queryView` page and add the following JavaScript code to it:

```
function onEmployeeEdit(event){
  var table = event.getSource();
  AdfCustomEvent.queue(table, "onEmployeeEdit",{}, true);
  event.cancel();
}
```

12. Locate a **Popup** component in the **Component Palette** and drop it on the page, inside the `af:form` tag.
13. Right-click on the `af:popup` component in the **Structure** window and select **Insert Inside af:popup | Dialog** from the context menu.
14. Locate the `Employees` collection under `HrComponentsAppModuleDataControl` in the **Data Controls** window and drop it on the `af:dialog` component in the page. From the **Create** menu, select **Form | ADF Form...**. Adjust the fields you want to display in the **Edit Form Fields** dialog and click **OK**.
15. Using the **Property Inspector** for the `af:popup` component, change the **Id** property to `editEmployee` and the **ContentDelivery** to **lazyUncached**.
16. Finally, adjust the results table **PartialTriggers** by adding the `af:dialog` identifier to it.

How it works...

In steps 1 and 2, we have updated the `queryTaskFlow` task flow definition, which was introduced in the *Using an af:query component to construct a search page* recipe in this chapter, by adding a managed bean definition and generating the bean class. In step 3, we have added a method to the managed bean called `onEmployeeEdit()`. This method is used as an event listener for the `af:serverListener` that is added to the `af:table` component in step 8. It is used to programmatically show the `editEmployee` `af:popup`. The `editEmployee` pop-up is added to the page in step 13. The pop-up is shown programmatically by infusing the page with dynamic JavaScript code using the `addScript()` method implemented by the `ExtendedRenderKitService` interface. The JavaScript code that is added is specified as an argument to the `addScript()` method. In this case the code is as follows:

```
AdfPage.PAGE.findComponentByAbsoluteId('editEmployee').show();
```

This piece of JavaScript code locates the `editEmployee` component in the page and displays it.

The pop-up is invoked by double-clicking on a table row. In order to accomplish this behavior, a combination of an `af:clientListener` and an `af:serverListener` tag is used. We add these components in steps 6 and 7 respectively.

When we added the `af:clientListener` tag in step 6, we indicated that a JavaScript method called `onEmployeeEdit()` will be executed when we double-click on a table row. This JavaScript method is added directly to the page in steps 9 through 12. The JavaScript `onEmployeeEdit()` method is shown as follows:.

```
function onEmployeeEdit(event){
  var table = event.getSource();
  AdfCustomEvent.queue(table, "onEmployeeEdit",{}, true);
  event.cancel();
}
```

The method retrieves the table component from the event and queues a custom event to the table component called `onEmployeeEdit`. This indicates the `af:serverListener` that was added in step 7.

Back in step 7, when we added the `af:serverListener` to the `af:table`, we identified the serverListener of type `onEmployeeEdit` and indicated that the backing bean `QueryBeanonEmployeeEdit` method will be executed upon its activation. This is the method implemented in step 3 that programmatically raises the pop-up.

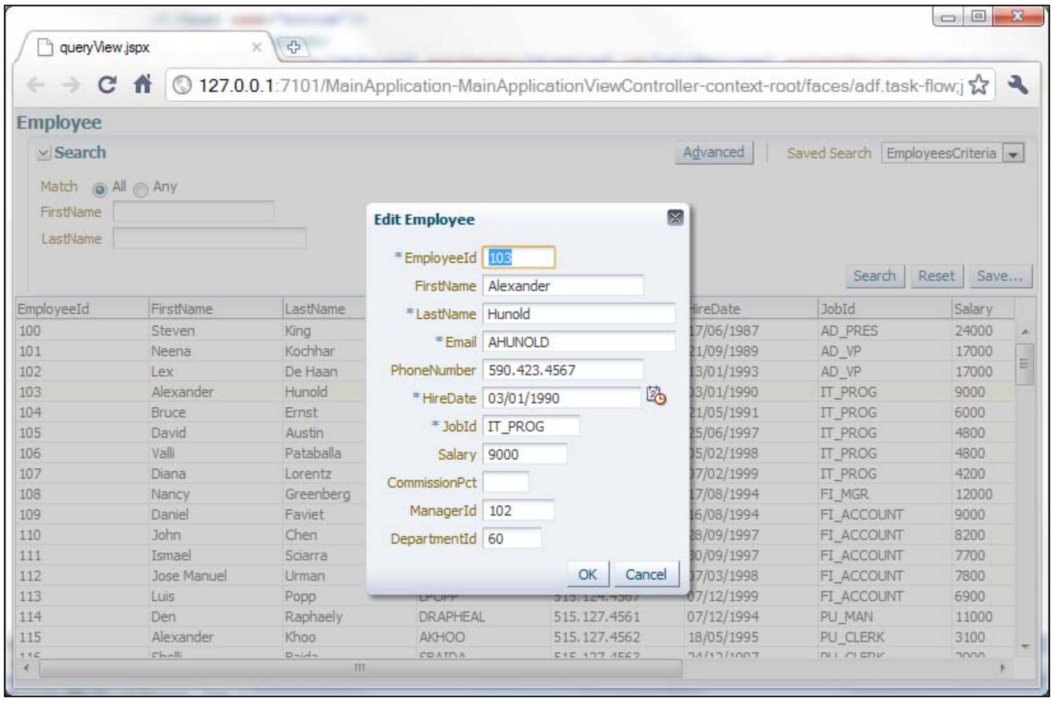
We mentioned earlier that the JavaScript code for the `af:clientListener onEmployeeEdit` method was added in steps 9 through 11. JavaScript is added directly on the page by adding an `af:resource` component of type `javascript` to the `af:document`. The actual page code looks similar to the following:

```
<af:document ...
  <af:resource type="javascript">
  function onEmployeeEdit(event){
    var table = event.getSource();
    AdfCustomEvent.queue(table, "onEmployeeEdit",{}, true);
    event.cancel();
  }
  </af:resource>
</af:document>
```

The pop-up is added to the page in steps 12 through 14 using a combination of the `af:popup` and `af:dialog` components. In step 14, we dropped the `Employees` collection from the **Data Controls** right on the `af:dialog` as an editable form. Since the collection is bound to the page's table, we will be editing the same data.

Finally, note the adjustments that we have made to the pop-up and table components in steps 15 and 16. First we changed the pop-up identifier to `editEmployee`. This is necessary since we specify pop-up in step 3 by name: `AdfPage.PAGE. findComponentByAbsolut eId('editEmployee').show()`. Then we set the pop-up's `contentDelivery` attribute to `lazyUncached`. This attribute indicates how the pop-up content is delivered to the client. The `lazyUncached` content delivery setting is used because the data delivered to the pop-up component from the server will change as we double-click in different rows on the table. The `PartialTriggers` settings indicate how the related page components are refreshed. In this case, we want changes made to the data in the pop-up, to be mirrored in the table. We can accomplish this by adding the dialog's identifier to the table's list of partial triggers.

To run the recipe, right-click on the `queryTaskFLow` in the **Application Navigator** and select **Run** or **Debug**. When the page is displayed, click **Search** to perform a search. Double-click on a row in the results table to show the **Edit Employee** dialog. Any changes you make are saved by clicking **OK** on the **Edit Employee** dialog. If you click **Cancel**, the changes are dismissed. The table is updated to match the adjusted data.


queryView.jspx
127.0.0.1:7101/MainApplication-MainApplicationViewController-context-root/faces/adf.task-flow;j
Employee
Search
Advanced
Saved Search EmployeesCriteria
Search
Reset
Save...
OK
Cancel

Match [✓ All] Any

FirstName

LastName

Edit Employee

* EmployeeId	103
FirstName	Alexander
* LastName	Hunold
* Email	AHUNOLD
PhoneNumber	590.423.4567
* HireDate	03/01/1990
* JobId	IT_PROG
Salary	9000
CommissionPct	
ManagerId	102
DepartmentId	60

EmployeeId	FirstName	LastName			HireDate	JobId	Salary
100	Steven	King			17/06/1987	AD_PRES	24000
101	Neena	Kochhar			21/09/1989	AD_VP	17000
102	Lex	De Haan			13/01/1993	AD_VP	17000
103	Alexander	Hunold			03/01/1990	IT_PROG	9000
104	Bruce	Ernst			21/05/1991	IT_PROG	6000
105	David	Austin			25/06/1997	IT_PROG	4800
106	Valli	Pataballa			05/02/1998	IT_PROG	4800
107	Diana	Lorentz			07/02/1999	IT_PROG	4200
108	Nancy	Greenberg			17/08/1994	FI_MGR	12000
109	Daniel	Faviet			16/08/1994	FI_ACCOUNT	9000
110	John	Chen			28/09/1997	FI_ACCOUNT	8200
111	Ismael	Sciarra			30/09/1997	FI_ACCOUNT	7700
112	Jose Manuel	Urman			07/03/1998	FI_ACCOUNT	7800
113	Luis	Popp	LPOPP	515.124.4567	07/12/1999	FI_ACCOUNT	6900
114	Den	Raphaely	DRAPHEAL	515.127.4561	07/12/1994	PU_MAN	11000
115	Alexander	Khoo	AKHOO	515.127.4562	18/05/1995	PU_CLERK	3100

There's more...

Note that the page does not implement a commit or rollback functionality, so changes done to the table's data are not committed to the database. To rollback the changes for now, just refresh the browser; this will re-fetch the data from the database and re-populate the results table. Also, note the functionality of the **Search** and **Reset** buttons. The **Search** button populates the results table by searching the database, while at the same time preserving any changed records in the entity cache. This means that your changes still show in the table after a new search. The behavior of the **Reset** button does not refresh by default in the results table. We will cover how to accomplish this in the recipe *Using a custom af:query operation listener to clear both the query criteria and results* in *Chapter 8*, *Backing not Baking: Bean Recipes*.

Moreover, note that this recipe shows how to launch a pop-up component programmatically using the `ExtendedRenderKitService` class by infusing dynamic JavaScript code into the page. It is this infused JavaScript code that actually shows the pop-up. Another approach to programmatically launching a pop-up is to bind the `af:popup` component to a backing bean as an `oracle.adf.view.rich.component.rich.RichPopup` object, then use its `show()` method to display the pop-up. For more information about this technique, take a look at the section *Programmatically invoking a Pop-up* in the *Web User Interface Developer's Guide for Oracle Application Development Framework* which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Using an af:query component to construct a search page*, in this chapter

Using an af:tree component

The ADF Faces Tree component (`af:tree`) can be used to display model-driven master-detail data relationships in a hierarchical manner. In this case, the parent node of the tree indicates the master object, while the child nodes of the tree are the detail objects.

In this recipe, we will demonstrate the usage of the `af:tree` component to implement the following use case: Using the `HR` schema, we will create a JSF page that presents a hierarchical list of the departments and their employees in a tree. As you navigate the tree, the detailed department or employee information will be displayed in an editable form. The recipe makes use of a custom selection listener to determine the type of the tree node (department or employee) being clicked. Based on the type of node, it then displays the department or the employee information.

Getting ready

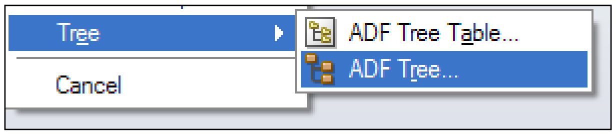
You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces* recipe, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

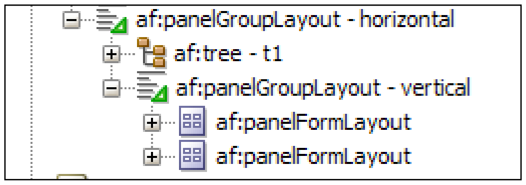
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

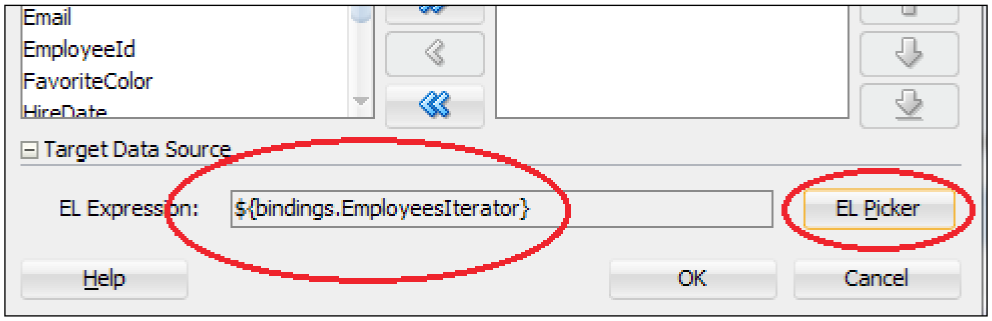
1. Ensure that the `HRComponents` and the `SharedComponents` ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `treeView.jspx`. Use any of the predefined quick start layouts.
3. Expand the **Data Controls** section in the **Application Navigator** and locate the `Departments` collection under the `HrComponentsAppModuleDataControl` data control. Drag-and-drop it on the `treeView.jspx` page.
4. From the **Create** menu, select **Tree** | **ADF Tree...**.



5. In the **Tree Level Rules** section of the **Edit Tree Binding** dialog, click on the **Add Rule** button (the green plus sign icon) and add the `Employees` collection. Also adjust the attributes in the **Display Attributes** list so that the `DepartmentName` is listed for the `Departments` rule and the `LastName` and `FirstName` are listed for the `Employees` rule. Click **OK** to proceed.
6. Right-click on the `af:tree` component in the **Structure** window and select **Surround With...**. From the **Surround With** dialog, select the **Panel Group Layout** and click **OK**. Using the **Properties Inspector**, set the **Valign** and **Layout** attributes to **top** and **horizontal** respectively.
7. In the **Data Controls** section, locate the `Departments` and `Employees` collections under the `HrComponentsAppModuleDataControl` data control and drop them inside the **panelGroupLayout**. In both cases, select **Form** | **ADF Form...** from the **Create** menu.
8. For each of the `af:panelFormLayout` components created previously for the `Departments` and `Employees` collections, set their **Visible** property to **false** and bind them to a backing bean called `TreeBean`. If needed, create the `TreeBean` backing bean as well.
9. Surround both of the `af:panelFormLayout` components created previously for the `Departments` and `Employees` collections with the same `af:panelGroupLayout`. Using the **Property Inspector**, set the **Layout** attribute of the new `af:panelGroupLayout` to **vertical**. Also, ensure that you specify the tree's identifier in the **PartialTriggers** attribute of the `af:panelGroupLayout`. Your components in the Structure window should look similar to the following screenshot:

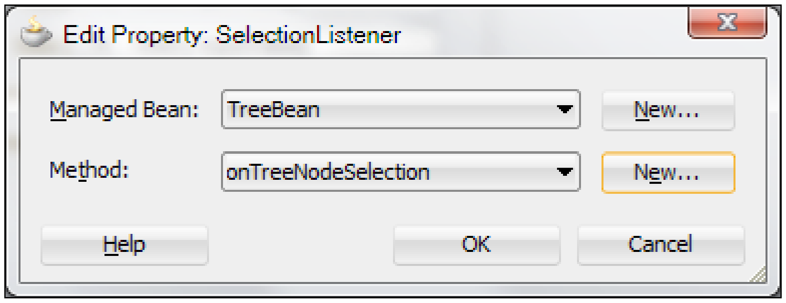

af:panelGroupLayout - horizontal
af:tree - t1
af:panelGroupLayout - vertical
af:panelFormLayout
af:panelFormLayout

10. With `af:tree` selected in the **Structure** window, click on the **Edit Component Definition** button (the pen icon) in the **Property Inspector** to open the **Edit Tree Binding** dialog. With the `Employees` rule selected in the **Tree Level Rules**, expand the **Target Data Source** section at the bottom of the dialog. Use the **EL Picker** button and select the `EmployeesIterator` under **the ADF Bindings** | **bindings** node. The **EL Expression** `${bindings.EmployeesIterator}` should be added, as shown in the following screenshot:



EL Expression: ${bindings.EmployeesIterator}

11. While at the `af:tree` **Property Inspector**, change the **SelectionListener** to a newly created selection listener in the `TreeBean` backing bean.



Managed Bean: TreeBean

Method: onTreeNodeSelection

12. Locate the `TreeBean.java` in the **Application Navigator** and double-click on it to open it in the Java editor. Add the following code to the `onTreeNodeSelection()` selection listener:

```
// invoke default selection listener via bindings
invokeMethodExpression(
  "#{bindings.Departments.treeModel.makeCurrent}",
  Object.class, new Class[] { SelectionEvent.class},
  new Object[] { selectionEvent});
// get the tree component from the event
RichTree richTree = (RichTree)selectionEvent.getSource();
// make the selected row current
RowKeySet rowKeySet = richTree.getSelectedRowKeys();
Object key = rowKeySet.iterator().next();
richTree.setRowKey(key);
// get the tree node selected
JUCtrlHierNodeBinding currentNode =
   (JUCtrlHierNodeBinding)richTree.getRowData();
// show or hide the department and employee information
// panels depending the type of node selected
this.departmentInfoPanel.setVisible(
  currentNode.getCurrentRow() instanceof DepartmentsRowImpl);
this.employeeInfoPanel.setVisible(
  currentNode.getCurrentRow() instanceof EmployeesRowImpl);
```

13. Add the following `invokeMethodExpression()` helper method to the `TreeBean.java`:

```
private Object invokeMethodExpression(String expression,
  Class returnType, Class[] argTypes, Object[] args) {
  FacesContext fc = FacesContext.getCurrentInstance();
  ELContext elContext = fc.getELContext();
  ExpressionFactory elFactory =
    fc.getApplication().getExpressionFactory();
  MethodExpression methodExpression =
    elFactory.createMethodExpression(elContext,
    expression, returnType, argTypes);
  return methodExpression.invoke(elContext, args);
}
```

How it works...

Since we will be using business components from the `HRComponents` workspace, in step 1 we have ensured that the `HRComponents` ADF Library JAR is added to the workspace's ViewController project. This can be done either through the **Resource Palette** or using the **Project Properties | Libraries and Classpath** dialog. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project. Then, we proceed with the creation of the JSF page (step 2).

In steps 3 through 5, we added the Tree component to the page. The tree is comprised of two nodes or level rules: the parent node represents the departments, and is set up by dragging and dropping the `Departments` collection of the `HrComponentsAppModuleDataControl` data control onto the page as an ADF Tree component. The child nodes represent the department employees and are set up in step 5 by adding a rule for the `Employees` collection. The rules control the display order of the tree. The tree binding populates the component starting at the top of the tree level rules list and continues until it reaches the last rule.

In steps 6 through 9, we dropped the `Departments` and `Employees` collections on the page as editable forms (`af:panelFormLayout` components) and rearranged the page in such a way that the tree will be displayed on the left-hand side of the page, while the department or employee information will be displayed on the right-hand side. We also bound the department and employee `af:panelFormLayout` components in a backing (in step 8), so that we will be able to dynamically show and hide them depending on the currently selected node (see step 12). For this to work, we also need to do a couple more things:

- Set the `af:panelGroupLayout` component's (used to vertically group the department and employee `af:panelFormLayout` components) `partialTriggers` attribute to the tree's identifier (in step 9)
- Setup the tree's target data source for the `Employees` rule, so that the `Employees` iterator is updated based on the selected node in the tree hierarchy (in step 10)

Finally, in steps 11 through 13, we created a custom selection listener for the tree component, so that we are able to dynamically show and hide the department and employee forms depending on the tree node type that is selected. The custom selection listener is implemented by the backing bean method called `onTreeNodeSelection()`. If we look closer at this method, we will see that first we invoke the default tree selection listener with the expression `#{bindings.Departments.treeModel.makeCurrent}`. In order to do this, we use a helper method called `invokeMethodExpression()`. Then, we obtain the currently selected node from the tree by calling `getRowData()` on the `oracle.adf.view.rich.component.rich.data.RichTree` component (obtained earlier from the selection event). Finally, we dynamically change the visible property of the department and employee `af:panelFormLayout` components, depending on the type of the selected node. We do this by calling `setVisible()` on the bound department and employee `af:panelFormLayout` components.

There's more...

Note that when adding an `af:tree` component to the page, a single iterator binding is added to the page definition for populating the root nodes of the tree. The accessors specified in the tree level rules, which return the detailed data for each child node, are indicated by the `nodeDefinition` XML nodes of the `tree` binding in the page definition.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Using an af:selectManyShuttle component

The `af:selectManyShuttle` ADF Faces component is a databound model-driven component that can be used to select multiple items from a given list. Using a set of pre-defined buttons, you move the selected items from an available items list to a selected items list. Upon completion of the selection process, you can programmatically retrieve and process the selected items.

In this recipe, we will go over the steps to declaratively create an `af:selectManyShuttle` component in a pop-up dialog and programmatically retrieve the selected items.

Getting ready

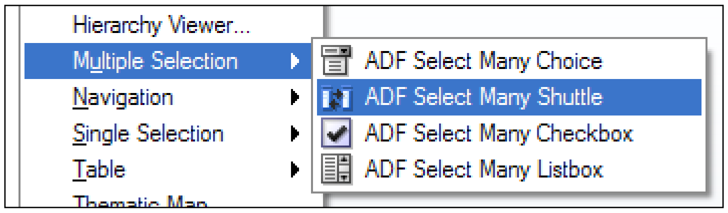
You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

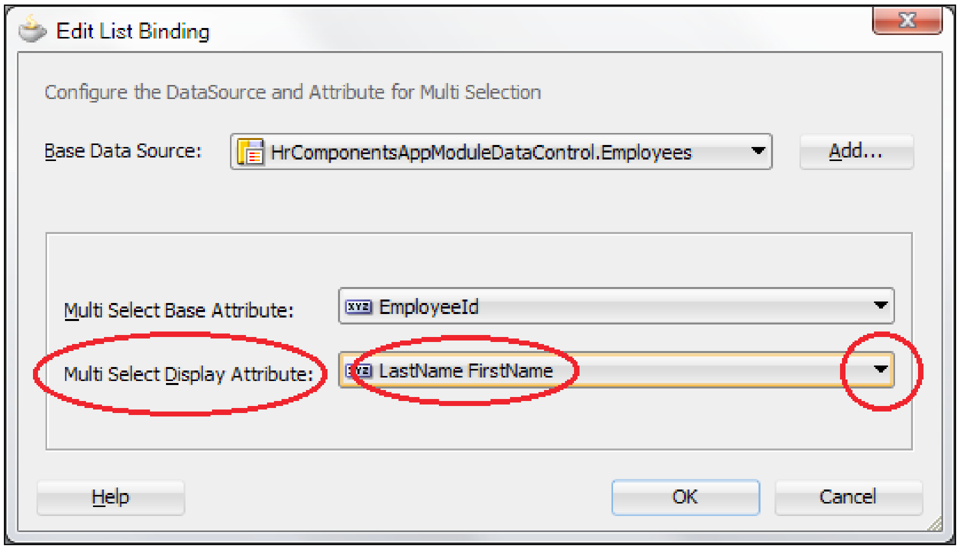
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Ensure that the `HRComponents` and the `SharedComponents` ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `selectManyShuttleView.jspx`. Use any of the predefined quick start layouts.
3. Using the **Component Palette**, add a **Popup** component (`af:popup`) to the page. Also add a **Dialog** component (`af:dialog`) inside the pop-up.
4. Expand the **Data Controls** section in the **Application Navigator** and locate the `Employees` collection under the `HrComponentsAppModuleDataControl` data control. Drag-and-drop it on the `selectManyShuttleView.jspx` page inside the dialog.
5. From the **Create** menu, select **Multiple Selection** | **ADF Select Many Shuttle...**.



6. In the **Edit List Binding** dialog, use the **Select Multiple...** selection from the **Multi Select Display Attribute** dropdown and select the `LastName` and `FirstName` attributes.


Edit List Binding

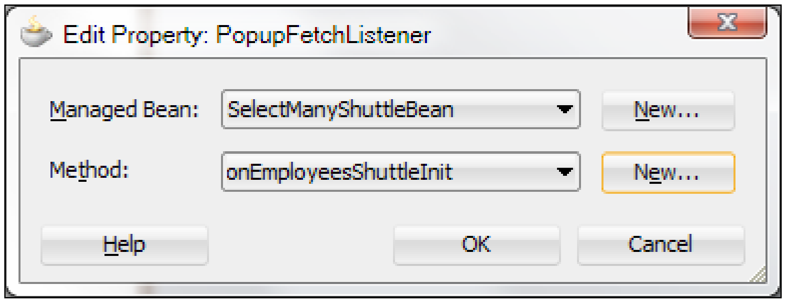
Configure the DataSource and Attribute for Multi Selection

Base Data Source: HrComponentsAppModuleDataControl.Employees

Multi Select Base Attribute: EmployeeId

Multi Select Display Attribute: LastName FirstName

7. Select `af:popup` in the **Structure** window. Using the **Property Menu** next to the **PopupFetchListener** attribute in the **Property Inspector**, select **Edit...** to add a pop-up fetch listener. When presented with the **Edit Property: PopupFetchListener** dialog, create a new managed bean called `SelectManyShuttleBean` and a method called `onEmployeesShuttleInit`. While in the **Property Inspector**, also change the **ContentDelivery** attribute to **lazyUncached**.



Managed Bean: SelectManyShuttleBean
Method: onEmployeesShuttleInit

8. Open the `SelectManyShuttleBean` bean in the Java editor and add the following code to the `onEmployeesShuttleInit()` method:

```
JUCtrlListBinding employeesList =
  (JUCtrlListBinding)ADFUtils.findCtrlBinding("Employees");
employeesList.clearSelectedIndices();
```

9. Select `af:dialog` in the **Structure** window. Using the **Property Menu** next to the **DialogListener** attribute in the **Property Inspector**, select **Edit...** and add a dialog listener. In the **Edit Property**: **DialogListener** dialog, use the `SelectManyShuttleBean` and add a new method called `onSelectManyShuttleDialogListener`.

10. Add the following code to the `onSelectManyShuttleDialogListener()` method of the `SelectManyShuttleBean` managed bean:

```
if (DialogEvent.Outcome.ok.equals(dialogEvent.getOutcome())) {
  JUCtrlListBinding employeesList =
    (JUCtrlListBinding)ADFUtils.findCtrlBinding("Employees");
  Object[] employeeIds = employeesList.getSelectedValues();
  for (Object employeeId : employeeIds) {
    // handle selection
  }
}
```

11. Finally, add a **Button** component (`af:commandButton`) to the page and a **Show Popup Behavior** component (`af:showPopupBehavior`) in it. For the Show Pop-up Behavior component setup its `PopupId` attribute to point to the pop-up created previously.

How it works...

Since we will be importing business components from the `HRComponents` workspace, in step 1 we ensured that the corresponding ADF Library JAR was added to our ViewController project. This can be done either through the **Project Properties** | **Libraries and Classpath** dialog or via the **Resource Palette**. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project.

In steps 2 and 3, we have created a new JSF page called `selectManyShuttleView.jspx` and added a pop-up to it with a dialog component in it. We will display this pop-up via the command button added in step 11.

In steps 4 through 6, we declaratively added a model-driven `af:selectManyShuttle` component. We did this by dragging and dropping the `Employees` collection available under the `HrComponentsAppModuleDataControl` data control in the **Data Controls** section of the **Application Navigator**. This was added to the list of the available data controls in step 1 when the `HRComponents` ADF Library JAR was added to our project. Note in step 6 how we have modified the `Employees` collection attributes that will be displayed by the ADF Select Many Shuttle. In this case, we have indicated that the employee's last name and first name will be displayed. In the same step, we have left the **Multi Select Base Attribute** to the default `EmployeeId`, indicating the attribute that will receive the updates. The effect of adding the Select Many Shuttle is to also add a `list` binding called `Employees` to the page bindings, as shown in the following code snippet:

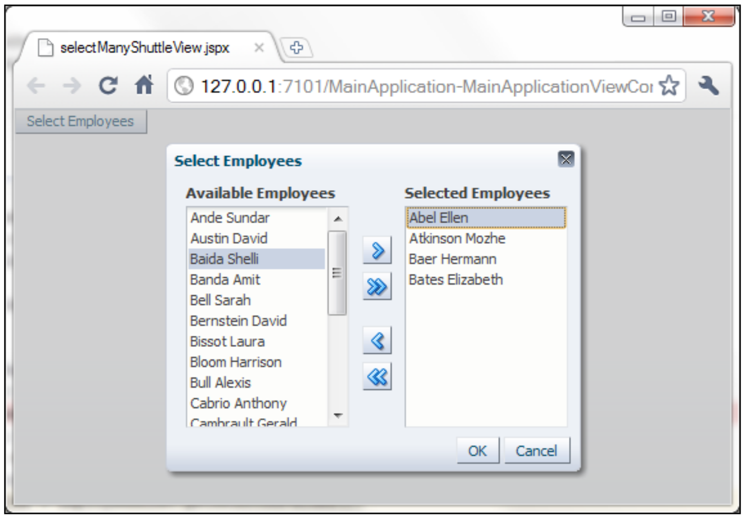
```
<bindings>
  <list IterBinding="EmployeesIterator"
    ListOperMode="multiSelect"ListIter="EmployeesIterator"
    id="Employees" SelectItemValueMode="ListObject">
    <AttrNames>
      <Item Value="EmployeeId"/>
    </AttrNames>
    <ListDisplayAttrNames>
      <Item Value="LastName"/>
      <Item Value="FirstName"/>
    </ListDisplayAttrNames>
  </list>
</bindings>
```

In steps 7 and 8, we have devised a way to initialize the shuttle's selections before the pop-up is shown. We have done this by adding a `PopupFetchListener` to the pop-up. A `PopupFetchListener` indicates a method that is executed when a pop-up fetch event is invoked during content delivery. For the listener method to be executed, the pop-up content delivery must be set to `lazyUnchached` or `lazy`. We set the pop-up content delivery to `lazyUnchached` in step 7. The `PopupFetchListener` method was called `onEmployeesShuttleInit()`. In it, we retrieve the `Employees` list binding by utilizing the `ADFUtils.findCtrlBinding()` helper method. We introduced the `ADFUtils` helper class in the *Using ADFUtils/JSFUtils* recipe in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. Once the list binding is retrieved as a `JUCtrlListBinding` object, we call `clearSelectedIndices()` on it to clear the selections. This will ensure that the selected list is empty once the pop-up is displayed.

To handle the list selections, we added a `DialogListener` to the dialog in steps 9 and 10. A `DialogListener` is a method that can be used to handle the outcome of a dialog event. In it, we first checked to see whether the **OK** button was clicked by checking for a `DialogEvent.Outcome.ok` outcome. If this is the case, we retrieve the list binding and call `getSelectedValues()` on it to retrieve a `java.lang.Object` array of the selections. In our case, since we have indicated in step 6 that the `EmployeeId` attribute will be used as the base attribute, this is an array of the selected employee identifiers. Once we have the list of selected employees (as employee identifiers), we can process it as needed.

Note in step 11 that we have added a command button with an embedded `af:showPopupBehavior` in order to show the pop-up.

To test the page, right-click on it in the **Application Navigator** and select **Run** or **Debug** from the context menu. Clicking on the command button will display the pop-up with a shuttle component displaying a list of available employees to select from, as shown in the following screenshot:



There's more...

Note that ADF Faces provides an additional shuttle component named `af:selectOrderShuttle` that includes additional buttons to allow for the reordering of the selected items.

For more information about the ADF Faces Select Many Shuttle component, take a look at the section *Using Shuttle Components* in the *Web User Interface Developer's Guide for Oracle Application Development Framework* which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Using an af:carousel component

The ADF Faces Carousel component (`af:carousel`) is a model-driven databound user interface control that you can use on your pages as an alternate way to display collections of data. As the name suggests, the data is displayed in a revolving "carousel". The component comes with predefined controls that allow you to scroll through the carousel items. Moreover, images and textual descriptions can be associated and displayed for each carousel item.

In this recipe, we will demonstrate the usage of the `af:carousel` component by declaratively setting up a carousel to browse through the employees associated with each department.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

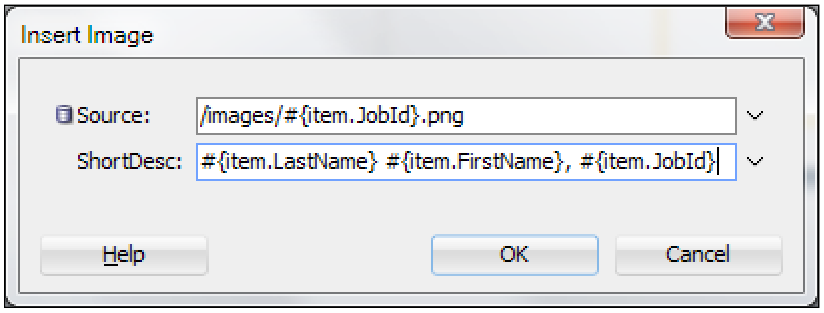
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Ensure that the `HRComponents` and the `SharedComponents` ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `carouselView.jspx`. Use any of the predefined quick start layouts.
3. Expand the **Data Controls** section in the **Application Navigator** and locate the `Departments` collection under the `HrComponentsAppModuleDataControl` data control. Drag-and-drop it on the `carouselView.jspx` page. From the **Create** menu, select **Table | ADF Read-only Table...**.
4. In the **Edit Table Columns** dialog, select the table columns and indicate **Single Row** for the **Row Selection**.
5. Drag-and-drop the `DepartmentEmployees` collection under the `Departments` collection on the `carouselView.jspx` page under the departments table. From the **Create** menu, select **Carousel**.
6. With the `af:carousel` selected in the **Structure** window, add a partial trigger to the departments table using the **Property Menu** next to the **PartialTriggers** attribute. Select **Edit...** from the property menu and in the **Edit Property: PartialTriggers** dialog add the table item to the selected items. Click **OK** to save your changes.
7. Expand the `af:carousel` component in the **Structure** window and locate the `af:carouselItem` underneath it. With the `af:carouselItem` selected in the **Structure** window, add the following to the **Text** attribute:

   ```
   #{item.LastName} #{item.FirstName}, #{item.JobId}
   ```

8. Using the **Component Palette**, locate an **Image** component and drag-and-drop it on the `af:carouselItem`. In the **Insert Image** dialog, specify `/images/#{item.JobId}.png` for the image **Source** and `#{item.LastName} #{item.FirstName}, #{item.JobId}` for the image **ShortDesc**.



Source: /images/#{item.JobId}.png

ShortDesc: #{item.LastName} #{item.FirstName}, #{item.JobId}

9. Under the `ViewController/public_html` directory create an `images` directory and add images for each employee job description. Ensure that the image filename conforms to the following naming standard: `#{item.JobId}.png`, where `#{item.JobId}` is the employee's job description. The employee's job descriptions are defined in `HR JOBS` and are identified by the `JOB_ID` column.

How it works...

Since we will be importing business components from the `HRComponents` workspace, in step 1, we ensured that the corresponding ADF Library JAR is added to our ViewController project. This can be done either through the **Project Properties** | **Libraries and Classpath** dialog or via the **Resource Palette**. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project.

In step 2, we have created a JSF page that we will use to demonstrate the `af:carousel` component. In the top part of the page, we added a table bound to the `Departments` collection. In the bottom part of the page, we added the `af:carousel` component bound to the `DepartmentEmployees` collection. As you select a department in the table, the corresponding department employees can be browsed using the carousel.

The `Departments` table was added in steps 3 and 4. The carousel was added in step 5. We simply expanded the `HrComponentsAppModuleDataControl` data control in the **Data Controls** section of the **Application Navigator** and dropped the collections on the page, making the applicable selections from the menus each time. JDeveloper proceeded by adding the components to the page and creating the necessary bindings in the page definition file. If you take a closer look at the page's source, you will see that the `af:carousel` component is created, with an associated child `af:carouselItem` component inside a `nodeStamp` facet in it. The page source looks similar to the following code:

```
<af:carousel currentItemKey="#{bindings
  .DepartmentEmployees.treeModel.rootCurrencyRowKey}"
  value="#{bindings.DepartmentEmployees.treeModel}" var="item" ...
  <f:facet name="nodeStamp">
    <af:carouselItem ...
      <af:image ...
    </af:carouselItem>
  </f:facet>
</af:carousel>
```

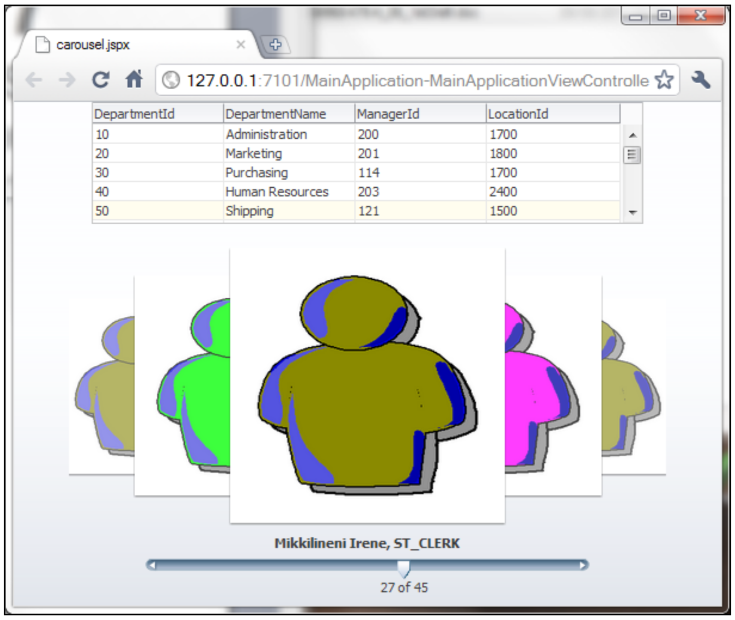
The carousel value is set to the `treeModel` for the `DepartmentEmployees` tree binding. This binding is created when the `DepartmentEmployees` collection is dropped on the page as a carousel. The tree binding is used to iterate over `DepartmentEmployeesIterator`, which is also created when the `DepartmentEmployees` collection is dropped on the page. The iterator result set is wrapped in a `treeModel` object, which allows each item in the result set to be accessed within the carousel using the `var` attribute. The current data in the result set is then accessed by the `af:carouselItem` using the `item` variable indicated by the carousel `var` attribute.

In order to synchronize the department selection in the table with the department employees in the carousel, the necessary partial trigger was added in step 6.

In step 7, we have set the `af:carouselItem Text` attribute to the `#{item.LastName} #{item.FirstName}, #{item.JobId}` expression. This will display the employee's name and job description underneath each carousel item. Remember that the `item` variable indicates the current data object in the result set.

Finally, in steps 8 and 9, we have added an image component (`af:image`) to the carousel item to further enhance the look of the carousel. The image source filename is dynamically determined using the expression `/images/#{item.JobId}.png`. This will use a different image depending on the value of the employee's job identifier. In step 9, we added the images for each employee job identifier.

To see the carousel in action, right-click on `carouselView.jspx` in the **Application Navigator** and select **Run** or **Debug**. Navigate through the Departments table using the carousel component through the department's employees.


carousel.jspx
127.0.0.1:7101/MainApplication-MainApplicationViewControlle

DepartmentId	DepartmentName	ManagerId	LocationId
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500

Mikkilineni Irene, ST_CLERK
27 of 45

There's more...

For this recipe, the employee images were explicitly specified as filenames, each one indicating a specific employee job using the expression `/images/#{item.JobId}.png`. In a more realistic scenario, images for each collection item would be stored in the database in a `BLOB` column associated with the collection item (the employee in this example). To retrieve the image content from the database `BLOB` column, you will need to write a servlet and indicate your choice by passing a parameter to the servlet. For instance, this could be indicated in the `af:image source` attribute as `/yourservlet?imageId=#{item.EmployeeId}`. In this case, the image is identified using the employee identifier. A sample demonstrating the image servlet can be found in the FOD sample.

For more information about the ADF Faces Carousel component, take a look at the section *Using the ADF Faces Carousel Component* in the *Fusion Developer's Guide for Oracle Application Development Framework* which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*

Using an af:poll component to periodically refresh a table

The ADF Faces Poll component (`af:poll`) can be used to deliver poll events to the server as a means to periodically update page components. Poll events are delivered to a poll listener—a managed bean method—by referencing the method using the `pollListener` attribute. These poll events are delivered to the poll listener based on the value specified by the `interval` attribute. The poll interval is indicated in milliseconds; polling can be disabled by setting the interval to a negative value. An `af:poll` can also be referenced from the `partialTriggers` property of a component to partially refresh the component. In this case, a `pollListener` is not needed.

In this recipe, we will implement polling in order to periodically refresh an employees table in the page. By periodically refreshing the table, it will reflect any database changes done to the corresponding `EMPLOYEES` schema table in the database.

Getting ready

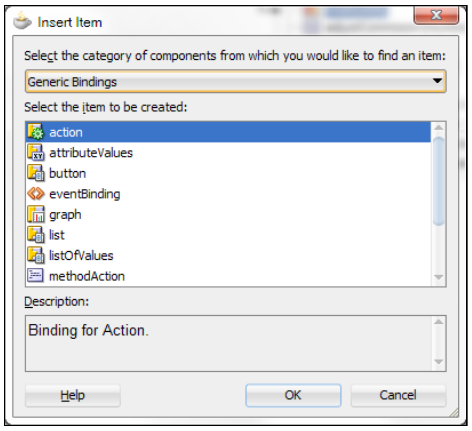
You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Ensure that the `HRComponents` and the `SharedComponents` ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `pollView.jspx`. Use any of the predefined quick start layouts.
3. Expand the **Data Controls** section in the **Application Navigator** and locate the `Employees` collection under the `HrComponentsAppModuleDataControl` data control. Drag-and-drop it on the `pollView.jspx` page. Then, from the **Create** menu, select **Table** | **ADF Read-only Table...**.
4. In the **Edit Table Columns** dialog, select the table columns and choose **Single Row** for the **Row Selection**.
5. Switch to the **Page Data Binding Definition** by clicking on the **Bindings** tab at the bottom of the page editor.
6. Click on the **Create control binding** button (the green plus sign icon) in the **Bindings** section and select **Action** from the **Generic Bindings** category.


Insert Item

Select the category of components from which you would like to find an item:

Generic Bindings

Select the item to be created:

- action
- attributeValues
- button
- eventBinding
- graph
- list
- listOfValues
- methodAction

Description:

Binding for Action.

7. In the **Creation Action Binding** dialog, select the `Employees` collection under the `HrComponentsAppModuleDataControl` and then select **Execute** for the **Operation**.
8. With the **Execute** action selected in the **Structure** window, change the **Id** property from **Execute** to `RefreshEmployees` using the **Property Inspector**.
9. Return to the page **Design** or **Source** editor. Using the **Component Palette**, drag a **Poll** component from the **Operations** section and drop it on the page.
10. With the `af:poll` component selected in the **Structure** window, change the **Interval** property to **3000** and add a poll listener using the **Property Menu** next to the **PollListener** property in the **Property Inspector**. If needed, create a new managed bean.
11. Open the managed bean Java class in the Java editor and add the following code to the poll listener:

    ```
    ADFUtils.findOperation("RefreshEmployees").execute();
    ```

12. Finally, add a partial trigger to the `af:table` component using the **Property Menu** next to the **PartialTriggers** property in the **Property Inspector**. In the **Edit Property: PartialTriggers** dialog, select the poll component in the **Available** list and add it to the **Selected** list.

How it works...

In step 1, we have added the `HRComponents` ADF Library JAR to our application's ViewController project. We have done this since we will be using the business components included in this library. The ADF Library JAR can be added to our project either via the **Resource Palette** or through the ViewController's **Project Properties** | **Libraries and Classpath**. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project.

Then, in step 2, we created a JSF page called `pollView.jspx` that we used to demonstrate the `af:poll` component by periodically refreshing a table of employees. So, in steps 3 and 4, we dropped the `Employees` collection—available through the `HrComponentsAppModuleDataControl` data control—as a read-only table on the page.

In steps 5 through 8, we created an action binding called `RefreshEmployees`. The `RefreshEmployees` action binding will invoke the `Execute` operation on the `Employees` collection, which will query the underlying `Employees` view object. So, by executing the `RefreshEmployees` action binding, we will be able to update the employees table, which is bound to the same `Employees` collection.

To accomplish a periodic update of the employees table, we dropped an `af:poll` component on the page (step 9) and adjusted the time interval in which a poll event will be dispatched (in step 10). This time interval is indicated by the `Interval` poll property in milliseconds, so we set it to 3 seconds (3000 milliseconds).

Then, in steps 10 and 11, we declared a poll listener using the `PollListener` property of the `af:poll` component. This is the method that will receive the poll event each time the poll is fired. In the process, we had to create a new managed bean (step 10). In the poll listener, we use the `ADFUtils` `findOperation()` helper method to retrieve the `RefreshEmployees` action binding from the bindings container. The `ADFUtils` helper class was introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. The `findOperation()` helper method returns an `oracle.binding.OperationBinding` object, on which we call `execute()` to execute it. As stated earlier, this will have the effect of querying the `Employees` collection underlying view object, which in effect refreshes the table.

Finally, in step 12, we had to indicate in the employees table's partial triggers the ID of the poll component. This will cause a partial page rendering for the `af:table` component triggered from the `af:poll` component each time the poll listener is executed.

To test the recipe, right-click on the `pollView.jspx` page in the **Application Navigator** and select **Run** or **Debug** from the context menu. Notice how the employees table is refreshed every 3 seconds, reflecting any modifications done to the `Employees` table.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Using page templates for pop-up reuse

Back in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* in the *Using a generic backing bean actions framework* recipe, we introduced a generic backing bean actions framework, called `CommonActions`, to handle common JSF page actions. In this recipe, we will enhance this generic actions framework by demonstrating how to add pop-up dialogs to a page template definition, that can then be reused by pages based on the template using this framework. The specific use case that we will implement in this recipe is to add a delete confirmation pop-up to the page template. This will provide a uniform delete behavior for all application pages based on this template.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to both the `CommonActions` generic backing bean framework and the `TemplateDef1` page template definition that were created in the *Using a generic backing bean actions framework* and *Using page templates* recipes in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

For testing purposes, you will need to create a skeleton **Fusion Web Application (ADF)** workspace. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace and locate the `TemplateDef1` page template definition using the **Application Navigator**. It can be found under the `WEB-INF/templates` package. Double-click on it so you can open it.
2. Using the **Component Palette**, drop a **Popup** component to the `top` facet. Modify the `af:popup` component's `id` property to `DeleteConfirmation`.
3. Drop a **Dialog** component inside the `af:popup` added in the previous step. Using the **Property Inspector**, update the dialog's **Title** to **Confirm Deletion**. Also change the **Type** property to **cancel**.
4. Drop an **Output Text** component from the **Component Palette** to the dialog. Change its **Value** property to **Continue with deleting this record?**
5. Using the **Component Palette**, drop a **Button** component to the dialog's `buttonBar` facet. Change the `af:commandButton text` property to `Continue`.
6. Using the **Property Inspector**, add the following **ActionListener** to the `af:commandButton: #{CommonActionsBean.onContinueDelete}`. The pop-up source should look similar to the following:

```
<af:popup id="DeleteConfirmation">
  <af:dialog id="pt_d1" title="Confirm Deletion" type="cancel">
  <af:outputText value="Continue with deleting this record?"
    id="pt_ot1"/>
```

```
        <f:facet name="buttonBar">
          <af:commandButton text="Continue"id="continueDeleteButton"
            actionListener="#{CommonActionsBean.onContinueDelete}"/>
        </f:facet>
      </af:dialog>
    </af:popup>
```

7. Locate the `ADFUTils` helper class and open it in the Java editor. Add the following code to the `showPopup()` method:

```
FacesContext facesContext = FacesContext.getCurrentInstance();
ExtendedRenderKitService service =
  Service.getRenderKitService(facesContext,
  ExtendedRenderKitService.class);
service.addScript(facesContext,
  "AdfPage.PAGE.findComponentByAbsoluteId ('generic:"
  + popupId + "').show();");
```

8. Redeploy the `SharedComponents` workspace into an ADF Library JAR.
9. Open the `MainApplication` workspace or create a new **Fusion Web Application (ADF)** workspace. Ensure that you add both the `SharedComponents` and `HRComponents` ADF Library JARs to the ViewController project.
10. Open the `adfc-config` unbounded task flow, go to **Overview** | **Managed Beans** and add a managed bean called `CommonActionsBean`. For the managed bean class, use the `CommonActions` class in the `com.packt.jdeveloper.cookbook.shared.view.actions` package imported from the `SharedComponents` ADF Library JAR.
11. Create a new JSPX page called `templatePopup.jspx` based on the `TemplateDef1` page template definition.
12. With the `af:pageTemplate` selected in the **Structure** window, change the template **Id** in the **Property Inspector** to `generic`.
13. Now, expand the `HrComponentsAppModuleDataControl` data control in the **Data Controls** section of the **Application Navigator** and drop the `Employees` collection on the **mainContent** facet of the page as an **ADF Read-only Form**. In the **Edit Form Fields** dialog, ensure that you select the **Include Navigation Controls** checkbox.
14. Using the **Component Palette**, drop a **Button** component to the form next to the **Last** button. With the button selected in the **Structure** window, change its **Text** property to **Delete**. Also set the ActionListener property to `#{CommonActionsBean.delete}`.

How it works...

In steps 1 through 6, we have expanded the `TemplateDef1` page template definition by adding a pop-up called `DeleteConfirmation`. We can raise this pop-up prior to deleting a record consistently for all of the application pages that are based on the `TemplateDef1` template. Notice that the name of the pop-up should match the name used in the `CommonActions.onConfirmDelete()` method to display the pop-up. This method looks similar to the following:

```
public void onConfirmDelete(final ActionEvent actionEvent) {
  ADFUtils.showPopup("DeleteConfirmation");
}
```

The necessary code to display the pop-up is added in the `ADFUtils.showPopup()` method in step 7. The `ADFUtils` helper class was introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. The following is the `ADFUtils.showPopup()` method:

```
public static void showPopup(String popupId) {
  FacesContext facesContext =FacesContext.getCurrentInstance();
  ExtendedRenderKitService service =
    Service.getRenderKitService(facesContext,
    ExtendedRenderKitService.class);
  service.addScript(facesContext,"AdfPage.PAGE.
    findComponent('generic:"+ popupId + "').show();");
}
```

The code in `ADFUtils.showPopup()` has been explained in the *Using an af:pop-up component to edit a table row* recipe in this chapter. One important thing to notice is how the template ID (`generic`) is prepended to the pop-up ID.

The `onConfirmDelete()` method is called by the generic delete action listener `CommonActions.delete()`. The following is the code for the `CommonActions.delete()` method:

```
public void delete(final ActionEvent actionEvent) {
  onConfirmDelete(actionEvent);
}
```

Notice how in step 5 we have added a **Continue** button to the delete confirmation pop-up and in step 6 we explicitly specify the `CommonActions.onContinueDelete()` method as the continue button's action listener. The code for this method is shown as follows:

```
public void onContinueDelete(final ActionEvent actionEvent) {
  CommonActions actions = getCommonActions();
   actions.onBeforeDelete(actionEvent);
   actions.onDelete(actionEvent);
   actions.onAfterDelete(actionEvent);
}
```

First we call `getCommonActions()` to determine if the `CommonActions` bean has been subclassed and then we call the appropriate action framework methods `onBeforeDelete()`, `onDelete()` and `onAfterDelete()`. The following is the code of the `getCommonActions()` method:

```
private CommonActions getCommonActions() {
  CommonActions actions =
    (CommonActions)JSFUtils.getExpressionObjectReference("#{"
    + getManagedBeanName() + "}");
  if (actions == null) {
    actions = this;
  }
  return actions;
}
```

The subclassed `CommonActions` managed bean name is determined by calling `getManagedBeanName()`. If a subclassed managed bean is not found, then the generic `CommonActions` bean is used; otherwise, the subclassed managed bean class is loaded using the `JSFUtils.getExpressionObjectReference()` helper method, which resolves the expression based on the bean name and instantiates it. The code for the `getManagedBeanName()` method is shown as follows:

```
private String getManagedBeanName() {
  return getPageId().replace("/", "").replace(".jspx", "");
}
```

As you can see, the subclassed managed bean name is determined by calling the helper `getPageId()`, which is shown as follows:

```
public String getPageId() {
  ControllerContext ctx = ControllerContext.getInstance();
  return ctx.getCurrentViewPort().getViewId().substring(
  ctx.getCurrentViewPort().getViewId().lastIndexOf("/")); }
```

The `getPageId()` determines the subclassed `CommonActions` managed bean name from the associated page. The fact that the subclassed managed bean name must match the page name, makes it a requirement for the `CommonActions` framework.

We continue in step 8 by redeploying the `SharedComponents` workspace to an ADF Library JAR.

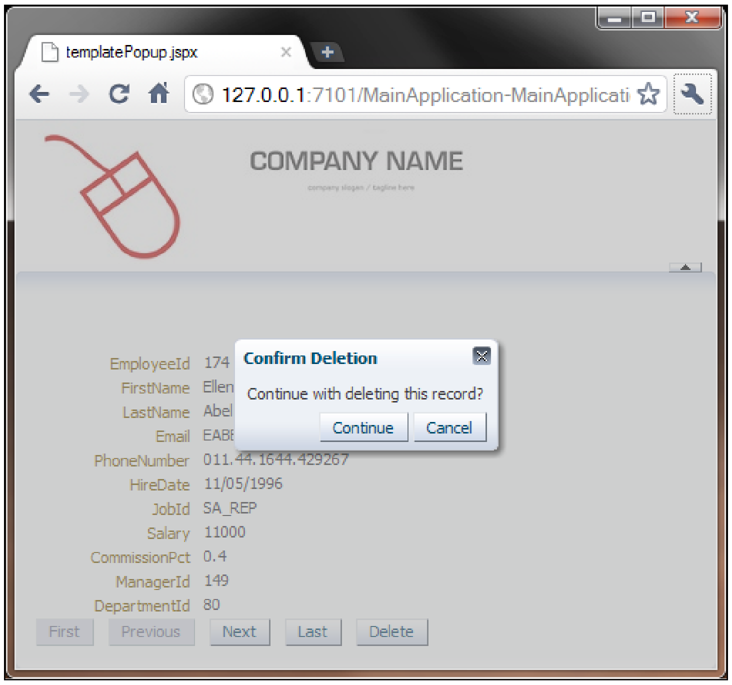
To test the generic template pop-up, in step 9, we created a **Fusion Web Application (ADF)** workspace and added the `SharedComponents` and `HRComponents` ADF Library JARs to its ViewController project.

In step 10, we added a managed bean, called `CommonActionsBean`, to our application based on the `CommonActions` class implemented in the `SharedComponents` workspace.

In steps 11 through 14, we created a page called `templatePopup.jspx` based on the `TemplateDef1` template and drop the `Employees` collection imported from the `HRComponents` workspace, as a read-only form. Notice in step 12 how we ensured that the `af:pageTemplate` component's identifier value is set to the same identifier value as in the template definition, that is, `generic`. This is important for the code in step 7 that loads the pop-up to function properly.

Finally, notice in step 14, how we set the delete button action listener to `#{CommonActionsBean.delete}`. This will allow for generic processing of the delete action.

To test the recipe, right-click on the `templatePopup.jspx` page in the **Application Navigator** and select **Run** or **Debug** from the context menu. When you click on the **Delete** button, the **Confirm Deletion** pop-up defined in the page template will be displayed and the `CommonActions` framework will be used to handle the delete action.



Continue with deleting this record?

EmployeeId	174
FirstName	Ellen
LastName	Abel
Email	EABE
PhoneNumber	011.44.1644.429267
HireDate	11/05/1996
JobId	SA_REP
Salary	11000
CommissionPct	0.4
ManagerId	149
DepartmentId	80

There's more...

For this recipe, we configured the delete button action processing so that it can be provided by the generic `CommonActions` `delete()` method. Assuming that you wanted to provide specialized handling of the delete action, you can do the following:

- Create a new managed bean with the same name as the page, that is, `templatePopup`
- Create the managed bean class and ensure that it extends the `CommonActions` class
- Provide specialized delete action functionality by overriding the following methods: `delete()`, `onBeforeDelete()`, `onDelete()`, `onAfterDelete()` and `onConfirmDelete()`
- Set the delete command button action listener to: `#{templatePopup.delete}`

See also

- *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using page templates*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Exporting data to a client file

You can export data from the server and download it to a file in the client by using the ADF Faces File Download Action Listener component, available in the **Operations** section of the **Component Palette**. Simply specify the default export filename and a managed bean method to handle the download. To actually export the data from the model using business components, you will have to iterate through the relevant view object and generate the exported string buffer.

In this recipe, we will use the File Download Action Listener component (`af:fileDownloadActionListener`) to demonstrate how to export all employees to a client file. The employees will be saved in the file in a comma-separated-values (CSV) format.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

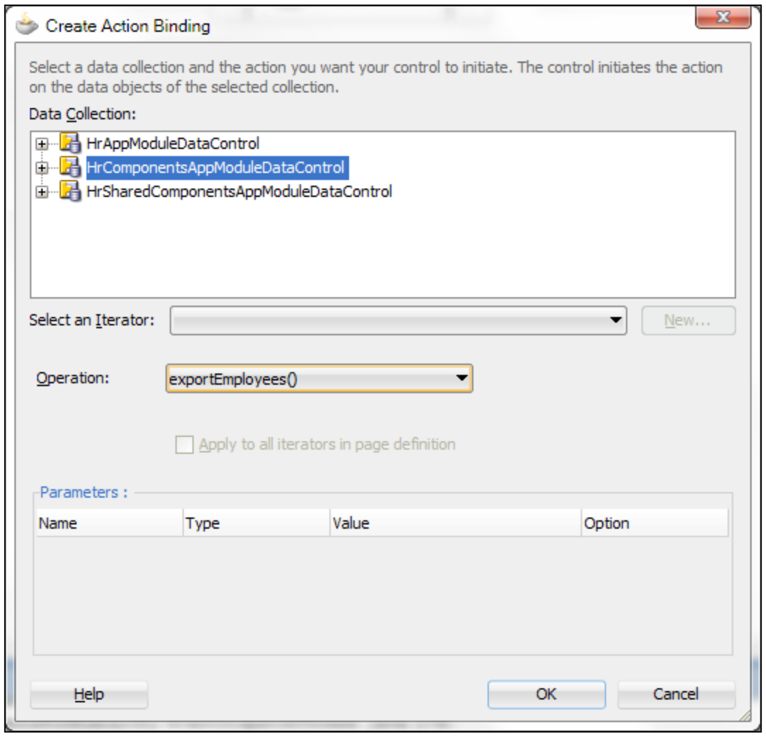
How to do it...

1. Open the `HRComponents` workspace and locate the `HrComponentsAppModule` application module. Open the custom application module Java implementation file `HrComponentsAppModuleImpl.java` in the Java editor.
2. Add the following `exportEmployees()` method to it:

```
public String exportEmployees() {
  EmployeesImpl employees = this.getEmployees();
  employees.executeQuery();
  StringBuilder employeeStringBuilder = new StringBuilder();
  RowSetIterator iterator =
  employees.createRowSetIterator(null);
  iterator.reset();
  while (iterator.hasNext()) {
    EmployeesRowImpl employee = (EmployeesRowImpl)iterator.next();
    employeeStringBuilder.append(employee.getLastName()
      + " " + employee.getFirstName());
    if (iterator.hasNext()) {
      employeeStringBuilder.append(",");
    }
  }
  iterator.closeRowSetIterator();
  return employeeStringBuilder.toString();
}
```

3. Double-click the `HrComponentsAppModule` application module in the **Application Module** and go to the **Java** section. Add the `exportEmployees()` method to the application module's client interface by clicking on the **Edit application module client interface** button (the pen icon).

4. Redeploy the `HRComponents` workspace as an ADF Library JAR.
5. Open the `MainApplication` workspace and add the `HRComponents` and the `SharedComponents` ADF Library JARs to its ViewController project.
6. Create a new JSPX page, called `exportEmployees.jspx`, using one of the quick start layouts.
7. Expand the `HrComponentsAppModuleDataControl` in the **Data Controls** section of the **Application Navigator** and locate the `exportEmployees()` method. Drop the `exportEmployees()` method on the page selecting **ADF Button** from the **Create** menu.
8. Right-click on the `af:commandButton` in the **Structure** window and select **Surround With...** from the context menu. In the **Surround With** dialog, select **Toolbar**.
9. With the `af:commandButton` selected in the **Structure** window, change the **Text** property to **Export Employees** and reset the **ActionListener** and **Disabled** properties to default (remove their expressions).
10. Switch to the **Bindings** tab and add a binding using the **Create control binding** button (the green plus sign icon). In the **Insert Item** dialog, select **methodAction** and click **OK**. In the **Create Action Binding** dialog, select the `HrComponentsAppModuleDataControl` in the **Data Collection** list and **exportEmployees()** for the **Operation**.


Create Action Binding

Select a data collection and the action you want your control to initiate. The control initiates the action on the data objects of the selected collection.

Data Collection:
- HrAppModuleDataControl
- HrComponentsAppModuleDataControl
- HrSharedComponentsAppModuleDataControl

Select an Iterator:

Operation: exportEmployees()

Apply to all iterators in page definition

Parameters :

Name	Type	Value	Option

11. Return to the page **Design** or **Source**. Right-click on the `af:commandButton` in the **Structure** window and select **Insert Inside af:commandButton ADF Faces...** from the context menu. In the **Insert ADF Faces Item** dialog, select **File Download Action Listener** and click **OK**.
12. With the `af:fileDownloadActionListener` selected in the **Structure** window, set the **Filename** property to `employees.csv` in the **Property Inspector**. For the **Method** property, expand on the **Property Menu** and select **Edit...**. In the **Edit Property: Menu** dialog, create a new managed bean called `ExportEmployeesBean` and a new method called `exportEmployees`. Click **OK** to dismiss the **Edit Property: Menu** dialog.
13. Now, open the managed bean and add the following code to the `exportEmployees()` method:

```
String employeesCSV = (String)ADFUtils.findOperation
  ("exportEmployees").execute();
try {
  OutputStreamWriter writer =
    new OutputStreamWriter(outputStream, "UTF-8");
  writer.write(employeesCSV);
  writer.close();
  outputStream.close();
} catch (IOException e) {
  // log the error
}
```

How it works...

In steps 1 through 4 we have updated the `HRComponents` ADF Library JAR by adding a method called `exportEmployees()` to the `HrComponentsAppModule` application module. In this method, we iterate over the `Employees` view object, and for each row we add the employee's last and first name to a string. We separate each employee name with a comma to create a string with all of the employee names in a comma-separated-values (CSV) format. In step 3, we have added the `exportEmployees()` method to the application module's client interface to make it available to the ViewController layer. Then, in step 4, we redeploy the `HRComponents` workspace into an ADF Library JAR.

Steps 5 through 13 cover working on the `MainApplication` workspace. You could instead create your own **Fusion Web Application (ADF)** workspace and apply them to that workspace instead. First, in step 5, we add the `HRComponents` ADF Library JAR to the ViewController project of the `MainApplication` workspace. You can do this either through the **Resource Palette** or through the **Project Properties** | **Libraries and Classpath** settings. The `HRComponents` library has dependencies to the `SharedComponents` workspace, so we make sure that the `SharedComponents` ADF Library JAR is also added to the project.

In step 6, we created a JSF page using one of the predefined quick start layouts. Then, in steps 7 through 10, we added a command button to the page with the underlying bindings. In step 7, note how we initially dropped the `exportEmployees()` method from the **Data Controls** window to the page as a button. We did this so that we can initialize the underlying page bindings. However, note how in step 10, we had to re-bind the `exportEmployees()` method as a `methodAction`. This is because in step 9 we removed the expressions from the `ActionListener` and `Disabled` properties, which as a result removed the `exportEmployees() methodAction` binding. Defining and using this `methodAction` binding will allow us to execute the `exportEmployees()` application module method that returns the employees in a CSV string buffer.

In steps 10 through 13, we added the File Download Action Listener component to the command button. Note in step 12, how we indicated a listener method, called `exportEmployees()`, that will be executed to perform the download action. The actual code for the listener was added in step 13. This code uses the `ADFUtils` helper class to programmatically execute the `exportEmployees methodAction` binding that we added in step 10. Executing the `exportEmployees methodAction` binding will result in returning the employees in a CSV formatted string. Then, using the `OutputStream` passed to the download action listener automatically by the ADF framework, we can write it to the stream. We introduced the `ADFUtils` helper class in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

To test the recipe, right-click on the `exportEmployees.jspx` page in the **Application Navigator** and select **Run** or **Debug** from the context menu. Observe what happens when you click on the **Export Employees** button. A **Save As** dialog is displayed asking you for the name of the file to save the employee CSV data. The default filename in this dialog is the filename indicated in the `Filename` property of the `af:fileDownloadActionListener` component, that is, `employees.csv`.

There's more...

For more information on the `af:fileDownloadActionListener` component, consult the section *How to Use a Command Component to Download Files* in the *Web User Interface Developer's Guide for Oracle Application Development Framework* which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

8 Backing not Baking: Bean Recipes

In this chapter, we will cover:

- Determining whether the current transaction has pending changes
- Using a custom af:table selection listener
- Using a custom af:query listener to allow execution of a custom application module operation
- Using a custom af:query operation listener to clear both the query criteria and results
- Using a session scope bean to preserve session-wide information
- Using an af:popup during long-running tasks
- Using an af:popup to handle pending changes
- Using an af:iterator to add pagination support to a collection

Introduction

Backing (also referred to as managed) beans are Java beans referenced by JSF pages in an ADF Fusion web application through Expression Language (EL). They are usually dedicated to providing specific functionality to the corresponding page. They are part of the ViewController layer in the Model-View-Controller architecture. Depending on their persistence in memory throughout the lifetime of the application, managed beans are categorized based on their scope: from `request` (minimal persistence in memory for the specific user request only) to `application` (maximum persistence in memory for the duration of the application). They can also exist in any of the `session`, `view`, `pageFlow`, and `backingBean` scopes. Managed bean definitions can be added to any of the following ADF Fusion web application configuration files:

- `faces-config.xml`: The JSF configuration file. It is searched first by the ADF framework for managed bean definitions. All scopes can be defined, except for `view`, `backingBean`, and `pageFlow` scopes, which are ADF-specifc.
- `adfc-config.xml`: The unbounded task flow definition file. Managed beans of any scope may be defined in this file. It is searched after the `faces-config.xml` JSF configuration file.
- `Specific task flow definition file`: In this file, the managed bean definitions are accessed only by the specific task flow.

Additionally, if you are using Facelets, you can register a backing bean using annotations.

Determining whether the current transaction has pending changes

This recipe shows you how to determine whether there are unsaved changes to the current transaction. This may come in handy when, for instance, you want to raise a warning pop-up message each time you attempt to leave the current page. This is demonstrated in the recipe *Using an af:popup to handle pending changes* in this chapter. Furthermore, by adding this functionality in a generic way to your application, making it part of the `CommonActions` framework for example, you can provide a standard application-wide approach for dealing with pending uncommitted transaction changes. The `CommonActions` framework was introduced in the *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

Getting ready

The functionality implemented in this recipe will be added to the `ADFUtils` helper class introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. This class is part of the `SharedComponents` workspace.

How to do it...

1. Open the `SharedComponents` workspace and locate the `ADFUtils.java` class in the **Application Navigator**.
2. Double-click on the `ADFUtils.java` to open it in the Java editor and add the following code to it:

```
public static boolean isBCTransactionDirty() {
  // get application module and check for dirty
  // transaction
  ApplicationModule am =
    ADFUtils.getDCBindingContainer().getDataControl()
    .getApplicationModule();
  return am.getTransaction().isDirty();
}
public static boolean isControllerTransactionDirty() {
  // get data control and check for dirty transaction
  BindingContext bc = BindingContext.getCurrent();
  String currentDataControlFrame =
    bc.getCurrentDataControlFrame();
  return bc.findDataControlFrame(
    currentDataControlFrame).isTransactionDirty();
}
```

3. Locate the `hasChanges()` method in the `ADFUtils` helper class. Add the following code to it:

```
// check for dirty transaction in both the model
// and the controller
return isBCTransactionDirty() ||
  isControllerTransactionDirty();
```

How it works...

In steps 1 and 2, we added two helper methods to the `ADFUtils` helper class, namely, `isBCTransactionDirty()` and `isControllerTransactionDirty()`.

The `isBCTransactionDirty()` method determines whether there are uncommitted transaction changes at the ADF-BC layer. This is done by first retrieving the application module from the data control `DCDataControl` class and then calling `getTransaction()` to get its `oracle.jbo.Transaction` transaction object. We call `isDirty()` on the `Transaction` object to determine if any application module data has been modified but not yet committed.

The `isControllerTransactionDirty()` method, on the other hand, checks for uncommitted changes at the controller layer. This is done by first calling `getCurrentDataControlFrame()` on the binding context to return the name of the current data control frame, and then calling `findDataControlFrame()` on the binding context to retrieve the `oracle.adf.model.DataControlFrame` object with the given name. Finally, we call `isTransactionDirty()` on the data control frame to determine whether unsaved data modifications exist within the current task flow context.

When checking for unsaved changes, we need to ensure that both the ADF-BC and the controller layers are checked. This is done by the `hasChanges()` method, which calls both `isBCTransactionDirty()` and `isControllerTransactionDirty()` and returns `true` if unsaved changes exist in any of the two layers.

There's more...

Note that for transient attributes used at the ADF-BC layer, `isDirty()` will return `true` only for entity object modified transient attributes. This is not the case for view object modified transient attributes, and `isDirty()` in this case returns `false`. In contrast, calling `isTransactionDirty()` at the ADFm layer will return `true` if any attributes have been modified.

See also

- *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using an af:popup to handle pending changes*, in this chapter.

Using a custom af:table selection listener

The `selectionListener` attribute of the ADF Table (`af:table`) component synchronizes the currently selected table row with the underlying ADF table binding iterator. By default, upon dropping a collection to a JSF page as an ADF table, JDeveloper sets the value of the `selectionListener` attribute of the corresponding `af:table` component to an expression similar to `#{bindings.SomeCollection.collectionModel.makeCurrent}`. This expression indicates that the `makeCurrent` method of the collection's model is called in order to synchronize the table selection with the table iterator binding.

In this recipe, we will cover how to implement your own custom table selection listener. This will come in handy if your application requires any additional processing before or after a table selection is made.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the MainApplication workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundation*s.

The recipe also uses the HRComponents workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the HRComponents and MainApplication workspaces require database connections to the HR schema.

Moreover, this recipe enhances the JSFUtils helper class introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*, which is part of the SharedComponents workspace.

How to do it...

1. Open the SharedComponents workspace and locate the JSFUtils helper class in the **Application Navigator**. Double-click on it to open it in the Java editor.
2. Add the following method to it, ensuring that you redeploy the SharedComponents workspace to an ADF Library JAR afterwards.

```
public static Object invokeMethodExpression(String expr,
  Class returnType, Class argType, Object argument) {
  FacesContext fc = FacesContext.getCurrentInstance();
  ELContext elctx = fc.getELContext();
  ExpressionFactory elFactory =
    fc.getApplication().getExpressionFactory();
  MethodExpression methodExpr =
    elFactory.createMethodExpression(elctx,
    expr, returnType, new Class[] { argType });
  return methodExpr.invoke(elctx, new Object[] { argument });
}
```

3. Now, open the MainApplication workspace and add the SharedComponents and the HRComponents ADF Library JARs to the ViewController project.
4. Create a JSP XML page based on any of the quick start layouts and drop the Employees collection, under the HrComponentsAppModuleDataControl in the **Data Controls** section of the **Application Navigator**, to the page.

5. With the **af:table** component selected in the **Structure** window, use the **SelectionListener** property menu **Edit...** in the **Property Inspector** and add a new selection listener, called `selectionListener`. Create a new managed bean when asked.

6. Open the managed bean and add the following code to the custom selection listener created previously:

```
// invoke makeCurrent via method expression
  JSFUtils.invokeMethodExpression(
"#{bindings.Employees.collectionModel.makeCurrent}",
  Object.class, SelectionEvent.class, selectionEvent);
// get selected data
RichTable table = (RichTable)selectionEvent.getSource();
JUCtrlHierNodeBinding selectedRowData =
  (JUCtrlHierNodeBinding)table.getSelectedRowData();
// process selected data
String[] attrbNames = selectedRowData.getAttributeNames();
for (String attrbName : attrbNames) {
  Object attrbValue =
    selectedRowData.getAttribute(attrbName);
  System.out.println("attrbName: " + attrbName +
    ", attrbValue: " + attrbValue);
```

How it works...

In steps 1 and 2, we updated the `JSFUtils` helper class by adding a method called `invokeMethodExpression()` used to invoke a JSF method expression. We also ensured that the `SharedComponents` workspace, where the `JSFUtils` helper class is defined, was redeployed into an ADF Library JAR. Then, in step 3, we added the newly deployed `SharedComponents` ADF Library JAR into the ViewController project of our application. We also added the `HRComponents` ADF Library JAR to the ViewController project, as we will be using the `Employees` collection in the steps that follow. You can add the ADF Library JARs either through the **Resource Palette** or through the ViewController **Project Properties | Libraries and Classpath** dialog settings.

In steps 4 and 5, we created a JSF page and dropped the `Employees` collection in it as an ADF Table (`af:table`) component. The `Employees` collection can be found in the `HrComponentsAppModule` application module which resides in the `HRComponents` ADF Library JAR. Then in step 6, we added a custom table `SelectionListener` by defining a method called `selectionListener()` in a managed bean. The code in the custom selection listener first invokes the default selection listener, by invoking the JSF method expression `#{bindings.Employees.collectionModel.makeCurrent}` using the helper method `invokeMethodExpression()` that we added in step 2.

The custom selection listener also demonstrates how to get the selected row data by first retrieving the ADF Table component as an `oracle.adf.view.rich.component.rich.data.RichTable` object. We call `getSource()` on the selection event and then call `getSelectedRowData()` on it. The call to `getSelectedRowData()` returns the ADF table binding as an `oracle.jbo.uicli.binding.JUCtrlHierNodeBinding` object, which can be used to subsequently retrieve the row data. This is done by calling `getAttributeNames()`, for instance, to retrieve the attribute names or by calling `getAttribute()` to retrieve the data value for a specific attribute. Once this information is known for the current table selection, additional business logic can be added to implement the specific application requirements.

There's more...

To do the analogous task with Java code, without invoking the default selection listener `makeCurrent`, involves getting the current row key from the node binding and setting the table `DCIteratorBinding` iterator binding (by calling `setCurrentRowWithKey()` on the iterator binding) to that key. For more information about this approach, take a look at Frank Nimphius' *ADF Corner* article *How-to build a generic Selection Listener for ADF bound ADF Faces Table*. It can be found currently in the following address: `http://www.oracle.com/technetwork/developer-tools/adf/learnmore/23-generic-table-selection-listener-169162.pdf`.

See also

- *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Using a custom af:query listener to allow execution of a custom application module operation

The `queryListener` attribute of the ADF Faces Query (`af:query`) component indicates a method that is invoked to execute the query. By default, the framework executes the `processQuery()` method referenced by the `searchRegion` binding associated with the `af:query` component. This is indicated by the following expression: `#{bindings.SomeQuery.processQuery}`. By creating a custom query listener method, you can provide a custom implementation each time a search is performed by the `af:query` component.

In this recipe, we will demonstrate how to create a custom query listener. Our custom query listener will programmatically execute the query by invoking the default expression as indicated previously. Moreover, after the query execution, it will display a message with the number of rows returned by the specific query.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `SharedComponents` and `HRComponents` workspaces, which were created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects* respectively.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `MainApplication` workspace and ensure that both the `SharedComponents` and the `HRComponents` workspaces are added to the ViewController project.
2. Create a JSP XML page called `queryListener.jspx` using one of the quick start layouts.
3. Locate the `EmployeesCriteria` named **criteria** under the **HrComponentsAppModuleDataControl | Employees** collection in the **Data Controls** section of the **Application Navigator**, and drop it on the page. Select **Query | ADF Query Panel with Table...** from the **Create** menu when asked.
4. With the **af:query** component selected in the **Structure** window, select **Edit...** from the **Property Menu** next to the **QueryListener** and create a new custom query listener method called `queryListener`. Create a new managed bean as well.
5. Open the managed bean that implements the custom query listener and add the following code it:

```
// handle the presence of certain query criterion data
List criteria =
  queryEvent.getDescriptor()
  .getConjunctionCriterion().getCriterionList();
for (int i = 0; i < criteria.size(); i++) {
  AttributeCriterion criterion =
    (AttributeCriterion)criteria.get(i);
```

```
          // do some special processing when a particular
          // criterion was used
          if ("SomeCriterionName".equals(
            criterion.getAttribute().getName()) &&
            criterion.getValues().get(0) != null) {
            // do something, for instance a rollback
            ADFUtils.findOperation("Rollback").execute();
            break;
          }
        }
        // invoke default processQuery query listener
        JSFUtils.invokeMethodExpression(
          "#{bindings.EmployeesCriteriaQuery.processQuery}",
          Object.class, QueryEvent.class, queryEvent);
        // display an information message indicating the
        // number of rows found
        long rowsFound = ADFUtils.findIterator("EmployeesIterator")
          .getEstimatedRowCount();
        FacesContext.getCurrentInstance().addMessage("",
          new FacesMessage(FacesMessage.SEVERITY_INFO,
          "Total Rows Found: " + rowsFound + "", null));
```

How it works...

In step 1, we added both the `SharedComponents` and `HRComponents` ADF Library JARs to the ViewController project of our application. This can be done either through the **Resource Palette** or via the **Project Properties | Libraries and Classpath** dialog settings.

In steps 2 and 3, we created a JSF page and dropped the `EmployeesCriteria` named criteria, defined in the `Employees` view object, as an **ADF Query Panel with Table** to the page. The `Employees` view object is part of the `HrComponentsAppModule`, that in turn is part of the `HRComponents` workspace imported as an ADF Library JAR in step 1. Once this JAR is imported to our project, the `HrComponentsAppModule` application module is available in the **Data Controls** section of the **Application Navigator**. Dropping the `EmployeesCriteria` named **criteria** on the page automatically creates the `af:query` and `af:table` components on the page, along with the underlying binding objects in the page definition file.

In steps 4 and 5, we created a custom query listener to be executed by the `af:query` component when performing the search. We did this declaratively through the **Property Inspector** that also allows us to create and configure a new managed bean, if needed. We simply called our custom query listener `queryListener` and added the necessary code to perform the search in step 4.

The code in the custom query listener `queryListener()` starts by demonstrating how to access the underlying `af:query` component's criteria. In the code, we iterate over the criteria looking for a specific criterion called `SomeCriterionName`. Once we find the specific criterion, we check whether a value is supplied for it and if so, we perform some action specific to our business domain. The criteria are obtained by calling `getCriterionList()` on the `oracle.adf.view.rich.model.ConjunctionCriterion` object, which is obtained by calling `getConjunctionCriterion()` on the `oracle.adf.view.rich.model.QueryDescriptor`. The `QueryDescriptor` is obtained from the event `QueryEvent` passed by the ADF framework to the query listener. The `getCriterionList()` method returns a `java.util.List` of `AttributeCriterion`, which we iterate over to check for the presence of the specific `SomeCriterionName` criterion. The `AttributeCriterion` indicates a query criterion. We can then call its `getValues()` method to retrieve the values supplied for the specific criterion.

To actually perform the search, we invoke the default `processQuery` method supplied by the framework via the expression `#{bindings.EmployeesCriteriaQuery.processQuery}`. This is done using the `JSFUtils` helper class method `invokeMethodExpression()`. The `JSFUtils` helper class was introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. We added the `invokeMethodExpression()` method to the `JSFUtils` class in the *Using a custom af:table selection listener* recipe in this chapter.

Finally, we retrieved the rows obtained after performing the search by calling `getEstimatedRowCount()` on the `Employees` iterator and displayed a message indicating the number of records yielded by the search.

There's more...

The `ConjunctionCriterion` object represents the collection of the search fields for a `QueryDescriptor` object. It contains one or more `oracle.adf.view.rich.model.Criterion` objects, and possibly other `ConjunctionCriterion` objects, combined using a conjunction operator.

For more information regarding the `af:query` UI artifacts and the associated `af:query` model class operations and properties, consult the section *Creating the Query Data Model* in the *Web User Interface Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Using ADFUtils/JSFUtils*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*

Using a custom af:query operation listener to clear both the query criteria and results

In the *Using a custom af:query listener to allow execution of a custom application module operation* recipe in this chapter, we demonstrated how to create your own custom query listener in order to handle the `af:query` component's search functionality yourself. In this recipe, we will show how to provide a custom reset operation functionality for the `af:query` component.

The default reset functionality implemented by the ADF framework resets the `af:query` component by clearing the criteria values, but does not clear the results of the associated `af:table` component that the framework creates when we drop some named criteria on the page. This reset functionality is indicated by the `queryOperationListener` attribute of the `af:query` component, and it is implemented by default by the framework `processQueryOperation()` method referenced by the `searchRegion` binding associated with the `af:query` component. It is indicated by the following expression: `#{bindings.SomeQuery.processQueryOperation}`. The `processQueryOperation()` method is used to handle all of the `af:query` component's operations such as `RESET`, `CREATE`, `UPDATE`, `DELETE`, `MODE_CHANGE`, and so on. These operations are defined by the ADF framework in the inner `Operation` class of the `oracle.adf.view.rich.event.QueryOperationEvent` class.

In this recipe, we will implement a custom `queryOperationListener` that will reset both the `af:query` and the `af:table` components used in conjunction in the same page to provide search functionality.

Getting ready

This recipe relies on having completed the *Using a custom af:query listener to allow execution of a custom application module operation* recipe in this chapter.

The recipe also uses the `SharedComponents` and `HRComponents` workspaces, which were created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects* respectively.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace and locate the `ExtApplicationModuleImpl.java` custom application module extension class. Add the following `resetCriteria()` method to it:

   ```
   public void resetCriteriaValues(ViewCriteria vc) {
     // reset automatic execution
     vc.setProperty(ViewCriteriaHints.CRITERIA_AUTO_EXECUTE,
       false);
     // reset view criteria variables
     VariableValueManager vvm = vc.ensureVariableManager();
     Variable[] variables = vvm.getVariables();
     for (Variable variable : variables) {
       vvm.setVariableValue(variable, null);
     }
     // reset view criteria
     vc.resetCriteria();
     vc.saveState();
   }
   ```

2. Redeploy the `SharedComponents` workspace to an ADF Library JAR.
3. Open the `HRComponents` workspace and locate the `HrComponentsAppModuleImpl.java` application module implementation class. Add the following `resetEmployees()` method to it:

   ```
   public void resetEmployees() {
     EmployeesImpl employees = this.getEmployees();
     ViewCriteria vc = employees.getViewCriteria(
       "EmployeesCriteria");
     // reset view criteria
     super.resetCriteriaValues(vc);
   ```

```
    employees.removeViewCriteria("EmployeesCriteria");
    employees.applyViewCriteria(vc);
    // reset Employees view object
    employees.executeEmptyRowSet();
  }
```

4. Add the `resetEmployees()` method to the application module client interface and redeploy the `HRComponents` workspace to an ADF Library JAR.
5. Open the `MainApplication` workspace. Double-click on the `queryListener.jspx` page in the **Application Navigator** to open the page in the page editor.
6. Click on the **Bindings** tab. Add a `methodAction` binding for the `resetEmployees()` operation under the `HrComponentsAppModuleDataControl` data control.
7. With the **af:query** component selected in the **Structure** window, select **Edit...** from the **Property Menu** next to the **QueryOperationListener** property in the **Property Inspector**.
8. In the **Edit Property: QueryOperationListener** dialog, select the `QueryListenerBean` and create a new method called `queryOperationListener`.
9. Open the `QueryListenerBean.java` in the Java editor and add the following code to the `queryOperationListener()` method:

```
// handle RESET operation only
if (QueryOperationEvent.Operation.RESET.name()
  .equalsIgnoreCase(queryOperationEvent.getOperation()
  .name())) {
  // execute custom reset
  ADFUtils.findOperation("resetEmployees").execute();
} else {
  // default framework handling for all other
  // af:query operations
  JSFUtils.invokeMethodExpression(
    "#{bindings.EmployeesCriteriaQuery.processQueryOperation}",
    Object.class, QueryOperationEvent.class,
    queryOperationEvent);
}
```

10. Finally, ensure that a partial trigger is added to the `af:table` component for the `af:query` component. You can do this using the **Property Menu** next to the **PartialTriggers** property in the `af:table` **Property Inspector**.

How it works...

In step 1, we added the resetCriteriaValues() method to the ExtApplicationModuleImpl custom application module extension class. This method becomes available to all derived application module classes, and is used to reset the specific named criteria values. The method accepts the ViewCriteria to reset, and iterates over the criteria variables obtained from the criteria VariableValueManager by calling getVariables(). For each variable, we call setVariableValue() on the VariableValueManager specifying the variable and a null value. We also call resetCriteria() to restore the criteria to the latest saved state, and saveState() to save the current state. We proceed to step 2 with redeploying the SharedComponents workspace to an ADF Library JAR.

In step 3, we added a method called resetEmployees() to the HrComponentsAppModule application module implementation class, which is used to reset the EmployeesCriteria named criteria defined for the Employees view object. In this method, we obtain the criteria by calling getViewCriteria() on the Employees view object and then call the resetCriteriaValues() method implemented in step 1 to reset the criteria variables. Then, we reapply the criteria to the Employees view object by first calling removeViewCriteria() and subsequently calling applyViewCriteria(). We also call executeEmptyRowSet() to empty the Employees view object result set. This will, in effect reset the af:table component on the page to display no records. In step 4, we added the resetEmployees() to the application module client interface, so that it can be bound to and invoked by the ViewController layer. We also redeployed the HRComponents workspace to an ADF Library JAR.

In steps 5 and 6, we added a method action binding for the resetEmployees() method implemented in step 3. We will call this method to reset the criteria and the Employees view object rowset in step 9 from a custom query operation listener.

In steps 7 and 8, we defined a custom query operation listener, called queryOperationListener() for the af:query component defined in the queryListener.jspx page. This page was created in the *Using a custom af:query listener to allow execution of a custom application module operation* recipe in this chapter.

In step 9, we wrote the necessary Java code to implement the custom query operation listener. First, we checked for the specific operation to ensure that we are dealing with a reset operation. We did this by retrieving the query operation from the QueryOperationEvent by calling getOperation() on it, and comparing it to the QueryOperationEvent.Operation.RESET operation. For a reset operation, we proceeded with executing the resetEmployees operation binding. Calling resetEmployees will reset both the af:query and af:table components. For all other af:query operations, we executed the default framework processQueryOperation() method by invoking the expression #{bindings.EmployeesCriteriaQuery.processQueryOperation}. This is done by calling the JSFUtils helper class invokeMethodExpression() method.

To ensure that the table will be visually updated by the custom reset operation, we added a partial trigger to the `af:table` component by indicating the `af:query` component identifier in its `partialTriggers` property.

There's more...

If you are writing a generic query operation listener and the presence of the `reset` operation binding cannot be guaranteed, use the `QueryModel.reset()` method to reset the `af:query` component only. The `reset()` method in this case is called for all system saved searches as it is shown in the code snippet:

```
try {
    // execute custom reset
    OperationBinding op = ADFUtils.findOperation("reset");
    op.execute();
} catch (RuntimeException e) {
    // just reset the af:query component only
    QueryModel queryModel = ((RichQuery)queryOperationEvent
                            .getSource()).getModel();
    for (int i = 0; i < queryModel.getSystemQueries().size();
                                              i++) {
        queryModel.reset(
            queryModel.getSystemQueries().get(i));
    }
}
```

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*
- *Using a custom af:query listener to allow execution of a custom application module operation* recipe in this chapter

Using a session scope bean to preserve session-wide information

Information stored in the DBMS can be preserved for the duration of the user session by utilizing ADF business components to retrieve it, and a session scope managed bean to preserve it throughout the user session. Using this technique allows us to access session-wide information from any page in our application, without the need to create specific bindings for it in each page.

This recipe demonstrates how to access and preserve session-wide information by implementing the following use case. For each employee authenticated to access the application, its specific information will be maintained by a session-scoped managed bean.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `SharedComponents` and `HRComponents` workspaces, which were created in *Breaking up the application in multiple workspaces Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects* respectively.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

The recipe assumes that ADF security has been enabled for the application and specific users matching the employee's last name have been added to the `jazn-data.xml` file. For information on how to enable ADF security take a look at *Enabling ADF security*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions, and Errors*.

How to do it...

1. Open the `HRComponents` workspace. Create a view object called `UserInfo` based on the `Employees` entity object.
2. Update the `UserInfo` view object query by adding the following `WHERE` clause to its query: `Employee.LAST_NAME = :inEmployeeName`.
3. Add a bind variable called `inEmployeeName`. Ensure that the **Value Type** is set to **Expression** and use the following Groovy expression in the **Value** field to initialize it: `adf.context.securityContext.userName`.

4. Ensure that you create both view object and view row Java classes.
5. Create an application module called `UserInfoAppModule` and add the `UserInfo` view object to its data model.
6. Generate an application module implementation class, and add the following methods to it. Also add these methods to the application module client interface.

```
public String getFirstName() {
  String firstName = null;
  UserInfoImpl usersInfo = (UserInfoImpl)getUserInfo();
  try {
    usersInfo.executeQuery();
    UserInfoRowImpl userInfo =
      (UserInfoRowImpl)usersInfo.first();
    if (userInfo != null) {
      firstName = userInfo.getFirstName ();
    }
  } catch (SQLStmtException sqlStmtException) {
    // handle exception
  }
  return firstName;
}
public String getLastName() {
  String lastName = null;
  UserInfoImpl usersInfo = (UserInfoImpl)getUserInfo();
  try {
    usersInfo.executeQuery();
    UserInfoRowImpl userInfo =
      (UserInfoRowImpl)usersInfo.first();
    if (userInfo != null) {
      lastName = userInfo.getLastName();
    }
  } catch (SQLStmtException sqlStmtException) {
    // handle exception
  }
  return lastName;
}
```

7. Redeploy the `HRComponents` workspace to an ADF Library JAR.
8. Open the `MainApplication` workspace and add both the `HRComponents` and the `SharedComponents` ADF Library JARs to the ViewController project.
9. Create a managed bean called `SessionInfoBean`. Make sure that the managed bean's scope is set to `session`. Also generate the managed bean class.

10. Open the `SessionInfoBean.java` class in the Java editor, and add the following code to it:

```
private String firstName;
private String lastName;
public SessionInfoBean() {
}
public String getFirstName() {
  if (firstName == null) {
    UserInfoAppModule userInfoAppModule =
      (UserInfoAppModule)ADFUtils
      .getApplicationModuleForDataControl(
      "UserInfoAppModuleDataControl");
    firstName = userInfoAppModule.getFirstName();
  }
  return firstName;
}
public String getLastName() {
  if (lastName == null) {
    UserInfoAppModule userInfoAppModule =
      (UserInfoAppModule)ADFUtils
      .getApplicationModuleForDataControl
      ("UserInfoAppModuleDataControl");
    lastName = userInfoAppModule.getLastName();
  }
  return lastName;
}
```

How it works...

In steps 1 through 4, we created a new view object called `UserInfo` based on the `Employees` entity object. Assuming that each employee will be authenticated to access our application using the employee's last name, we will use the information available in the `EMPLOYEES` database table to provide information specific to the employee currently authenticated. In order to retrieve information specific to the authenticated employee, we updated the `UserInfo` view object query by adding a `WHERE` clause to retrieve the specific employee based on a bind variable (in step 2). In step 3, we created the bind variable and used the Groovy expression `adf.context.securityContext.userName` to initialize it. This expression retrieves the authenticated user's name from the `SecurityContext` and uses it to query the specific employee.

In steps 5 and 6, we created an application module called `UserInfoAppModule`, and added the `UserInfo` view object to its data model and methods to retrieve the authenticated user's information. For this recipe, we added the methods `getFirstName()` and `getLastName()` to retrieve the user's first and last name respectively. These methods execute the `UserInfo` view object and retrieve the first row from the result set. In each case, the specific information is received by calling the corresponding `UserInfo` view row implementation class getter, that is, `getFirstName()` and `getLastName()`. Other methods can be added to retrieve additional user information based on your specific business requirements. In step 5, we also exposed these methods to the application module client interface, so that the methods can be bound and invoked from the ViewController layer.

In step 7, we redeployed the `HRComponents` workspace to an ADF Library JAR. Then, in step 8, we added the `HRComponents` along with the dependent `SharedComponents` ADF Library JARs to the `MainApplication`'s ViewController project.

Finally, in steps 9 and 10, we added a session-scoped managed bean, called `SessionInfoBean`, to the `MainApplication` ViewController project and implemented methods `getFirstName()` and `getLastName()` to retrieve the authenticated user's information. These methods call the corresponding `getFirstName()` and `getLastName()` implemented by the `UserInfoAppModule` application module in step 6. We got a reference to the `UserInfoAppModule` application module in the `SessionInfoBean` constructor by calling the `ADFUtils` helper class `getApplicationModuleForDataControl()` method. The `ADFUtils` helper class was introduced in *Using ADFUtils/JSFUtils*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

Now, we can use the following expressions on any page of our application to display the authenticated user's information:

Authenticated user's information	**Expression**
First Name	`#{SessionInfoBean.firstName}`
Last Name	`#{SessionInfoBean.lastName}`

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Handling Security, Session Timeouts, Exceptions and Errors*

Using an af:popup during long running tasks

For long-running tasks in your application, a pop-up message window can be raised to alert the users that the specific task may take a while. This can be accomplished using a combination of ADF Faces components (`af:popup` and `af:dialog`) and some JavaScript code.

In this recipe, we will initiate a long-running task in a managed bean, and raise a pop-up for the duration of the task to alert us to the fact that this operation may take awhile. We will hide the pop-up once the task completes.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `MainApplication` workspace and create a new JSPX page called `longRunningTask.jspx` based on any of the quick start layouts.
2. Drop a **Button** (`af:commandButton`) component from the **Component Palette** to the page. You may need to surround the button with an `af:toolbar` component. Using the **Property Inspector**, change the button's **Text** property to **Long Running Task** and set its **PartialSubmit** property to **true**.
3. Create an action listener for the button by selecting **Edit...** from the **Property Menu** next to the **ActionListener** property in the **Property Inspector**. Create a new managed bean called `LongRunningTaskBean` and a new method called `longRunningTask`.
4. Edit the `LongRunningTaskBean` Java class and add the following code to the `longRunningTask()` method:

```
try {
    // wait for 5 seconds
    Thread.currentThread().sleep(5000);
    } catch (InterruptedException e) {
}
```

5. Return to the `longRunningTask.jspx` page editor. Right-click on the **af:commandButton** in the **Structure** window and select **Insert Inside af:commandButton | ADF Faces...**. From the **Insert ADF Faces Item** dialog, select **Client Listener**. In the **Insert Client Listener** dialog, enter `longRunningTask` for the **Method** field and select **action** for the **Type** field.

6. Add an `af:resource` to the `af:document` tag. Make sure that the `af:resource type` attribute is set to `javascript` and add the following JavaScript code inside it:

```
function longRunningTask(evt) {
  var popup = AdfPage.PAGE.findComponentByAbsoluteId(
    'longRunningPopup');
  if (popup != null) {
    AdfPage.PAGE.addBusyStateListener(popup,
      busyStateListener);
    evt.preventUserInput();
  }
}
function busyStateListener(evt) {
  var popup = AdfPage.PAGE.findComponentByAbsoluteId(
    'longRunningPopup');
  if (popup != null) {
    if (evt.isBusy()) {
      popup.show();
    }
    else if (popup.isPopupVisible()) {
      popup.hide();
      AdfPage.PAGE.removeBusyStateListener(popup,
        busyStateListener);
    }
  }
}
```

7. Finally, add a Popup (`af:popup`) ADF Faces component to the page with an embedded Dialog (`af:dialog`) component in it. Ensure that the pop-up identifier is set to `longRunningPopup` and that its `ContentDelivery` attribute is set to `immediate`. Also add an `af:outputText` component to the dialog with some text indicating a long running process. Your pop-up should look similar to the following:

```
<af:popup childCreation="deferred" autoCancel="disabled"
  id="longRunningPopup" contentDelivery="immediate">
<af:dialog id="d2" closeIconVisible="false" type="none"
  title="Information">
<af:outputText value="Long operation in progress... Please
  wait..." id="ot1"/>
</af:dialog>
</af:popup>
```

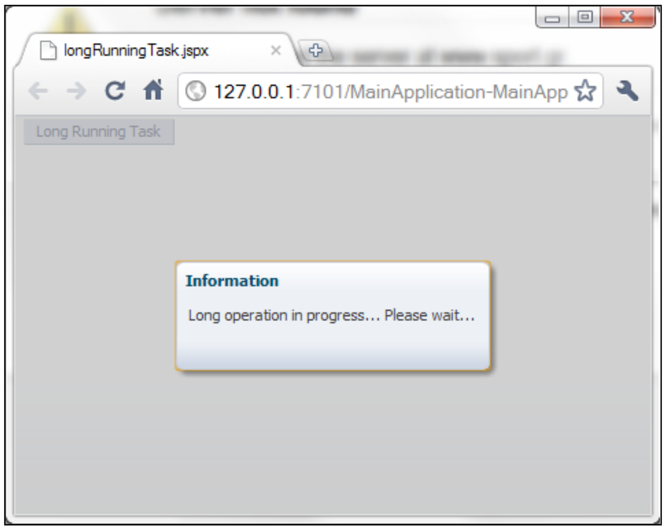
How it works...

In steps 1 and 2, we created a JSF page called `longRunningTask.jspx` and added a button component to it. When pressed, the button will initiate a long-running task through an action listener. The action listener is added to the button in steps 3 and 4. It is defined to a method called `longRunningTask()` in a managed bean. The implementation of `longRunningTask()` simply waits for 5 seconds (step 4). We have also ensured (in step 2) that the button component's `partialSubmit` property is set to `true`. This will enable us to call the `clientListener` method that is added in steps 5 and 6.

In steps 5 and 6, we defined a `clientListener` for the button component. The client listener is implemented by the `longRunningTask()` JavaScript method, added to the page in step 6. The `longRunningTask()` JavaScript method adds a busy state listener for the pop-up component (the pop-up itself is added to the page in step 7) by calling `addBusyStateListener()` and prevents any user input by calling `preventUserInput()` on the JavaScript event. The busy state listener is implemented by the JavaScript method `busyStateListener()`. In it, we hide the pop-up and remove the busy state listener once the event completes.

Finally, in step 7, we added the `longRunningPopup` pop-up to the page. The pop-up is raised by the `busyStateListener()` as long as the event is busy (for 5 seconds). We made sure that the pop-up's `contentDelivery` attribute was set to `immediate` to deliver the pop-up content immediately once the page is loaded.

To test the recipe, right-click on the `longRunningTask.jspx` page in the **Application Navigator** and select **Run** or **Debug** from the context menu. When you click on the button, the pop-up is raised for the duration of the long-running task (the action listener in the managed bean). The pop-up is hidden once the long-running task completes.


longRunningTask.jspx
127.0.0.1:7101/MainApplication-MainApp
Long Running Task
Information
Long operation in progress... Please wait...

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

Using an af:popup to handle pending changes

In the recipe *Determining whether the current transaction has pending changes* in this chapter, we showed how to establish whether there are uncommitted pending changes to the current transaction. In this recipe, we will use the functionality implemented in that recipe to provide a generic way to handle any pending uncommitted transaction changes. Specifically, we will update the `CommonActions` framework introduced in *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* to raise a pop-up message window asking you whether you want to commit the changes.
We will add the pop-up window to the `TemplateDef1` page template definition that we created in *Using page templates*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

Getting ready

We will modify the `TemplateDef1` page template definition and the `CommonActions` actions framework. Both reside in the `Sharedcomponents` workspace, which is deployed as an ADF Library JAR and it was introduced in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

Furthermore, we will utilize the `HRComponents` workspace, also deployed as an ADF Library JAR. This workspace was introduced in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Finally, you will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with the recipe. For this, you can use the `MainApplication` workspace introduced in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace, locate the `TemplateDef1` page template definition and open it in the page editor.
2. Add a Popup (`af:popup`) component to the page. Set the pop-up identifier to `CreatePendingChanges`. Add an embedded Dialog (`af:dialog`) component to the pop-up and set its **Title** attribute to **Confirm Pending Changes**.
3. Add an Output Text (`af:outputText`) component to the dialog and set its **Value** attribute to **Pending changes exist. Do you want to save changes?** Also add a Button (`af:commandButton`) component to the dialog and set its `ActionListener` property to `#{CommonActionsBean.onContinueCreate}`. The `CreatePendingChanges` dialog definition should look similar to the following:

```
<af:popup id="CreatePendingChanges">
<af:dialog id="pt_d2" title="Confirm Pending Changes"
  type="cancel">
<af:outputText value=
  "Pending changes exist. Do you want to save changes?"
  id="pt_ot2"/>
<f:facet name="buttonBar">
<af:commandButton id=
  "continuePendingChangesButton" text="Continue"
  binding=
  "#{CommonActionsBean.onContinueCreate}"/>
</f:facet>
</af:dialog>
</af:popup>
```

4. Open the `CommonActions` Java class in the Java editor and add the following methods to it:

```
public void create(final ActionEvent actionEvent) {
  if (ADFUtils.hasChanges()) {
    onCreatePendingChanges(actionEvent);
  } else {
    onContinueCreate(actionEvent);
  }
}
public void onCreatePendingChanges(
  final ActionEvent actionEvent) {
  ADFUtils.showPopup("CreatePendingChanges");
}
public void onContinueCreate(final ActionEvent actionEvent) {
  CommonActions actions = getCommonActions();
  actions.onBeforeCreate(actionEvent);
  actions.onCreate(actionEvent);
```

```
    actions.onAfterCreate(actionEvent);
  }
  protected void onBeforeCreate(final ActionEvent actionEvent) {
    // commit before creating a new record
    ADFUtils.execOperation(Operations.COMMIT);
  }
  public void onCreate(final ActionEvent actionEvent) {
    ADFUtils.execOperation(Operations.INSERT);
  }
  protected void onAfterCreate(final ActionEvent actionEvent) {
  }
```

5. Redeploy the `SharedComponents` workspace into an ADF Library JAR.
6. Open the main workspace application and ensure that both the `SharedComponents` and the `HRComponents` ADF Library JARs are added to the ViewController project.
7. Create a JSPX page called `pendingChanges.jspx` based on the `TemplatedDef1` template. Ensure that the `af:pageTemplate` component identifier in the page is set to `generic`.
8. Expand the **Data Controls** section of the **Application Navigator** and drop the `Employees` collection under the `HrComponentsAppModuleDataControl` to the page as an **ADF Form**.
9. Expand the **Operations** node under the `Employees` collection and drop a **CreateInsert** operation as an **ADF Button** to the page. Change the **CreateInsert** button's `ActionListener` property to the `CommonActions` framework `create()` method. The `ActionListener` expression should be `#{CommonActionsBean.create}`.
10. Switch to the page bindings and add an action binding for the `HrComponentsAppModuleDataControl` **Commit** operation.

How it works...

In steps 1 through 3, we added an `af:popup` component called `CreatePendingChanges` to the `TemplateDef1` page template definition. This is the popup that will be raised by the `CommonActions` framework if there are any unsaved transaction changes when we attempt to create a new record. This is done by the `CommonActions` `onCreatePendingChanges()` method (see step 4). Note that in step 3, we added a **Continue** button, which when pressed, saves the uncommitted changes. This is done through the button's action listener implemented by the `onContinueCreate()` method in the `CommonActions` framework (see step 4). If we press **Cancel**, the uncommitted changes are not saved (are still pending) and the creation of the new row is never initiated.

In step 4, we updated the `CommonActions` framework by adding the methods to handle the creation of a new row. Specifically, the following methods were added:

- `create()`: This method calls the `ADFUtils` helper class method `hasChanges()` to determine whether there are uncommitted transaction changes. If it finds any, it calls `onCreatePendingChanges()` to handle them. Otherwise, it calls `onContinueCreate()` to continue with the row creation action.
- `onCreatePendingChanges()`: The default implementation displays the `CreatePendingChanges` pop-up.
- `onContinueCreate()`: Called either directly from `create()`—if there are no pending changes—or from the `CreatePendingChanges` pop-up upon pressing the **Continue** button. Implements the actual row creation by calling the methods `onBeforeCreate()`, `onCreate()`, and `onAfterCreate()`.
- `onBeforeCreate()`: Called to handle any actions prior to the creation of the new row. The default implementation invokes the `Commit` action binding.
- `onCreate()`: Called to handle the creation of the new row. The default implementation invokes the `CreateInsert` action binding.
- `onAfterCreate()`: Called to handle any post creation actions. The default implementation does nothing.

In step 5, we redeploy the `SharedComponents` workspace to an ADF Library JAR. Then, in step 6, we add it along with the `HRComponents` ADF Library JAR to the `MainApplication`'s ViewController workspace.

In step 6, we created a JSPX page called `pendingChanges.jspx` based on the `TemplatedDef1` template. We made sure that the template identifier was set to `generic`, the same as the identifier of the `af:pageTemplateDef` component in the `TemplatedDef1` template definition. This is necessary because the code in the `ADFUtils.showPopup()` helper method, which is used to raise a pop-up, prepends the pop-up identifier with the template identifier.

In step 8, we created an ADF Form by dropping the `Employees` collection to the page. The `Employees` collection is part of the `HrComponentsAppModuleDataControl` data control, which is available once the `HRComponents` ADF Library JAR is added to the project.

Then, in step 9, we dropped the `CreateInsert` operation, available under the `Employees` collection, as an ADF Button to the page. Furthermore, we changed its `actionListener` property to the `CommonActions create()` method. This will handle the creation of the new row in a generic way and it will raise the pending changes pop-up, if there are any unsaved transaction changes.

Finally, in step 10, we added an action binding for the `Commit` operation. This is invoked by the `CommonActions onBeforeCreate()` method to commit any transaction pending changes.

The functionality to raise a pop-up message window indicating that there are pending unsaved transaction changes and committing the changes—as it is implemented in this recipe—applies specifically to the new row creation action. Similar functionality will need to be added for the other actions in your application, for instance, navigating to the next, previous, first, and last row in a collection.

See also

- *Determining whether the current transaction has pending changes*, in this chapter.
- *Using page templates*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

Using an af:iterator to add pagination support to a collection

A collection in an ADF Fusion web application, when dropped from the **Data Controls** window to a JSF page as an ADF Table, may be iterated through using the `af:table` ADF Faces component. Alternatively, when dropped as an ADF Form, it may be iterated a row at a time using the accompanying form buttons which can optionally be created by JDeveloper.

In this recipe, we will show how to add pagination support to a collection by utilizing the iterator (`af:iterator`) ADF Faces component along with the necessary scrolling support provided by a managed bean.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the main workspace application. Ensure that the `HRComponents` ADF Library JAR is added to its ViewController project.
2. Create a new JSP XML page called `collectionPagination.jspx` based on a quick start layout.
3. Expand the **Data Controls** window, locate the `Employees` collection under the `HrComponentsAppModuleDataControl` and drop it on the page as an ADF Read-only Table.
4. Switch to the page bindings editor, and with the `EmployeesIterator` iterator selected in the **Executables** list, change its **RangeSize** property to the desired page size. We will use 3 for this recipe.
5. Using the **Component Palette**, locate an Iterator component and drop it to the page. Using the **Property Inspector**, update the `af:iterator` component `Value`, `Var`, and `Rows` properties as shown in the following code fragment:

```
<af:iterator id="i1"
  value="#{bindings.Employees.collectionModel}" var="row"
  rows="#{bindings.Employees.rangeSize}"/>
```

6. Using the **Property Inspector**, bind the `af:iterator` component to a newly created managed bean, called `CollectionPaginationBean`. Now the `af:iterator` definition should look similar to the following:

```
<af:iterator id="i1"
  value="#{bindings.Employees.collectionModel}" var="row"
  rows="#{bindings.Employees.rangeSize}"
  binding="#{CollectionPaginationBean.employeesIterator}"/>
```

7. Move the `af:table` column contents (the `af:outputText` components) inside the `af:iterator` component. Remove the `af:table` component when done.
8. Surround the `af:iterator` with a Panel Box (`af:panelBox`) component. Drop a Toolbar component inside the panel box's toolbar facet. Add four buttons to the toolbar called First, Previous, Next, and Last.
9. For each of the buttons, add the action listeners and the disabled conditions shown in the following code fragment:

```
<af:panelBox
  text="Page # #{CollectionPaginationBean.pageNumber}"
  id="pb2">
<f:facet name="toolbar">
<af:toolbar id="t1">
<af:commandButton text="First" id="cb1"
  actionListener="#{CollectionPaginationBean.onFirst}"
  disabled="#{CollectionPaginationBean.previousRowAvailable
  eq false}"/>
```

```
<af:commandButton text="Previous" id="cb2"
  actionListener="#{CollectionPaginationBean.onPrevious}"
  disabled="#{CollectionPaginationBean.previousRowAvailable
  eq false}"/>
<af:commandButton text="Next" id="cb3"
  actionListener="#{CollectionPaginationBean.onNext}"
  disabled="#{CollectionPaginationBean.nextRowAvailable
  eq false}"/>
<af:commandButton text="Last" id="cb4"
  actionListener="#{CollectionPaginationBean.onLast}"
  disabled="#{CollectionPaginationBean.nextRowAvailable
   eq false}"/>
</af:toolbar>
</f:facet>
<af:iterator id="i1"
  value="#{bindings.Employees.collectionModel}" var="row"
  rows="#{bindings.Employees.rangeSize}"
  binding="#{CollectionPaginationBean.employeesIterator}">
```

10. Open the `CollectionPaginationBean` managed bean in the Java editor and add the following code to it:

```
public void onFirst(ActionEvent actionEvent) {
  this.employeesIterator.setFirst(0);
}
public void onPrevious(ActionEvent actionEvent) {
  this.employeesIterator.setFirst(
  this.employeesIterator.getFirst() - PAGE_SIZE);
}
public void onNext(ActionEvent actionEvent) {
  this.employeesIterator.setFirst(
  this.employeesIterator.getFirst() + PAGE_SIZE);
}
public void onLast(ActionEvent actionEvent) {
  this.employeesIterator.setFirst(
  employeesIterator.getRowCount() -
  employeesIterator.getRowCount() % PAGE_SIZE);
}
public boolean isPreviousRowAvailable() {
  return this.employeesIterator.getFirst() != 0;
}
public boolean isNextRowAvailable() {
  return (employeesIterator.getRowCount() >=
  employeesIterator.getFirst() + PAGE_SIZE);
}
public int getPageNumber() {
  return (this.employeesIterator.getFirst()/PAGE_SIZE) + 1;
}
```

How it works...

In step 1, we ensure that the `HRComponents` ADF Library JAR is added to the ViewController project of the `MainApplication` workspace. We will be using this library in order to access the `Employees` collection available through the `HrComponentsAppModule`. The library can be added to the project either through the **Resource Palette** or via the **Project Properties | Libraries and Classpath** options.

We created a new JSF page called `collectionPagination.jspx` in step 2, and in step 3, dropped the `Employees` collection on the page as an ADF Read-only Table component (`af:table`). When we did this JDeveloper created the underlying iterator and tree bindings. Then, in step 4, we switched to the page bindings and change the `EmployeesIterator` range size to our desired value. Note that this page size is indicated in the managed bean created in step 6 by the constant definition `PAGE_SIZE` and set to `3` for this recipe.

In steps 5 through 7, we setup an iterator (`af:iterator`) component. First, we dropped the iterator component on the page from the **Component Palette** and then we updated its `value` property (in step 5) to indicate the `CollectionModel` of the `Employees` tree binding, created earlier when we dropped the `Employees` collection to the page as a table. In addition, in step 5, we updated its `rows` and `var` attributes so that we will be able to copy over the table column contents to the `af:iterator` component. We did this in step 7. In step 6, we also bound the `af:iterator` component to a newly created managed bean called `CollectionPaginationBean` as a `UIXIterator` variable called `employeesIterator`.

In steps 8 and 9, we added a navigation toolbar to the page along with buttons for scrolling through the `Employees` collection namely buttons First, Previous, Next, and Last. For each button, we added the appropriate action listener and disabled the condition methods implemented by the `CollectionPaginationBean` managed bean (implemented in step 10). For the complete page source code, refer to the book's relevant source code.

Finally, in step 10, we implemented the action listener and disabled the condition methods for the navigation buttons. These methods are explained as follows:

- `onFirst()`: Action listener for the First button. Uses the bound iterator's `setFirst()` method with an argument of `0` (the index of the first row) to set the iterator to the beginning of the collection.
- `onPrevious()`: Action listener for the Previous button. Sets the first row to the current value decreased by the page size. This will scroll the collection to the previous page.
- `onNext()`: Action listener for the Next button. Sets the first row to the current value increased by the page size. This will scroll the collection to the next page.
- `onLast()`: Action listener for the Last button. Sets the first row to the first row of the last page. We call the `getRowCount()` iterator method to determine the iterator's row count and subtract the last page's rows from it. This will scroll the collection to the first row of the last page.

- `isPreviousRowAvailable()`: Disable condition for the First and Previous buttons. Returns `true` if the iterator's row index is not the first one.
- `isNextRowAvailable()`: Disable condition for the Last and Next buttons. Returns `true` if there are available rows beyond the current page.
- `getPageNumber()`: It is used in the page to display the current page number.

To test the recipe, right-click on the `collectionPagination.jspx` in the **Application Navigator** and select **Run** or **Debug** from the context menu.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*

9 Handling Security, Session Timeouts, Exceptions, and Errors

In this chapter, we will cover:

- Enabling ADF security
- Using a custom login page
- Accessing the application's security information
- Using OPSS to retrieve the authenticated user's profile from the identity store
- Detecting and handling session timeouts
- Using a custom error handler to customize how exceptions are reported to the ViewController
- Customizing the error message details
- Overriding attribute validation exceptions

Introduction

The ADF security framework provides authentication and authorization services for the ADF Fusion web application. To a certain degree, this security framework is supported in JDeveloper through a number of wizards and overview editors (available via the **Application | Secure** menu options) that allow interactive declarative configuration of certain parts of the application's security configuration. The security overview editors simplify the work needed to secure the application by authorizing ADF resources in a declarative manner. This resource authorization is achieved at the task flow, page definition and business components (entity objects and their attributes) levels. Authorization defined for task flows protects not only the task flow's entry point but all the pages included in the task flow as view activities.

Configuring the application's session timeout can be done through a number of options in the application's deployment descriptor file `web.xml`.

Customization of error and exception handling for an ADF Fusion web application can be achieved by overriding certain framework classes and by creating your own exception classes.

Enabling ADF security

Enabling security for an ADF Fusion web application involves enabling both user authentication and authorization. Authentication refers to enabling users to access your application using a credentials validation login facility. On the other hand, authorization refers to controlling access to the application resources by defining and configuring security policies on ADF application resources, such as task flows, page definitions, and business components (entity objects and their attributes). ADF security is enabled for the Fusion web application through the use of the **Configure ADF Security** wizard available under the **Application | Secure** menu option. Moreover, JDeveloper provides additional declarative support through the `jazn-data.xml` security configuration overview editor, and through declarative security support at the business components level using the entity object overview editor (**General** and **Attributes** tabs).

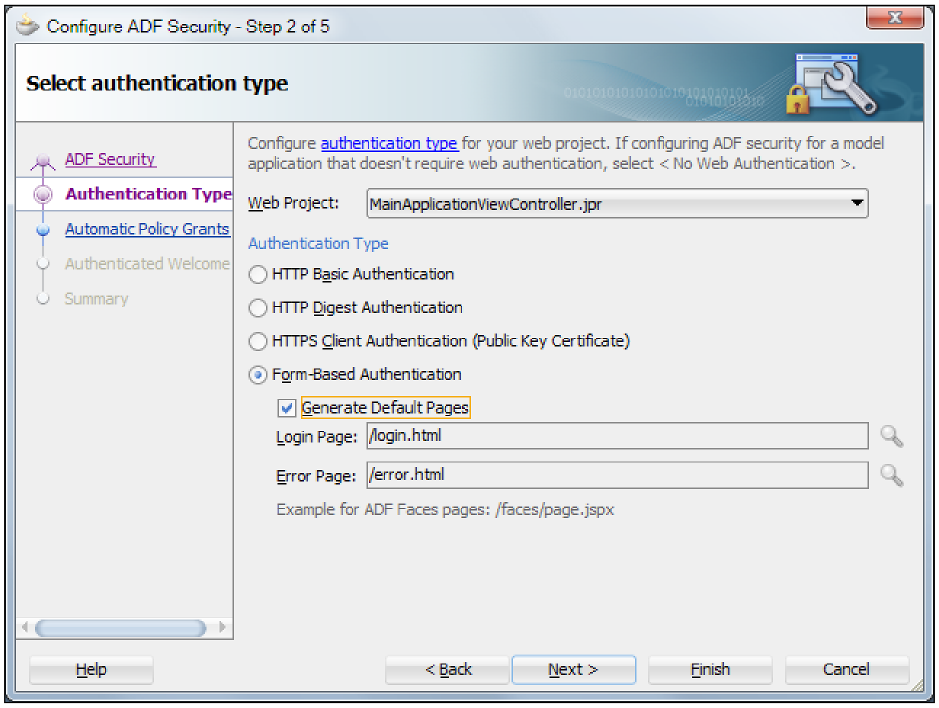
In this recipe, we will go over the process of enabling security for an ADF Fusion web application by creating and configuring the necessary artifacts, such as login, error and welcome pages, redirection, user and role creation, and configuration using the **Configure ADF Security** wizard.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `MainApplication` workspace. From the **Application** menu select **Secure | Configure ADF Security...** to start the **Configure ADF Security** wizard.
2. In the **Enable ADF** Security page, select **ADF Authentication and Authorization** and click **Next**.
3. In the **Select authentication type** page, select the appropriate ViewController project from the **Web Project** combo box. Select **Form-Based Authentication** and click on the **Generate Default Pages** checkbox. Click **Next** to proceed.


Configure ADF Security - Step 2 of 5

Select authentication type

Configure authentication type for your web project. If configuring ADF security for a model application that doesn't require web authentication, select < No Web Authentication >.

Web Project: MainApplicationViewController.jpr

Authentication Type
- ( ) HTTP Basic Authentication
- ( ) HTTP Digest Authentication
- ( ) HTTPS Client Authentication (Public Key Certificate)
- (●) Form-Based Authentication

✓ Generate Default Pages

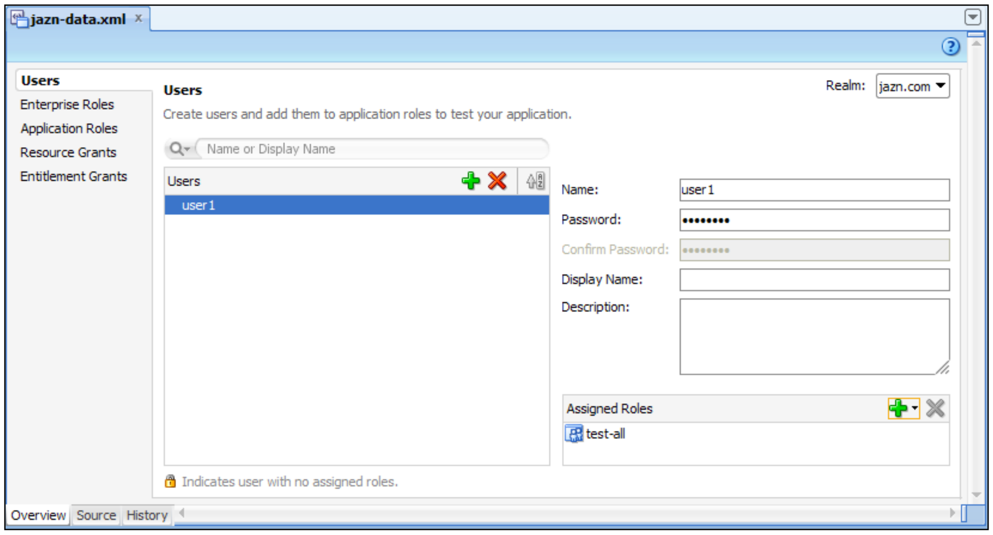
Login Page: /login.html

Error Page: /error.html

Example for ADF Faces pages: /faces/page.jspx

4. In the **Enable automatic policy grants** page, select **Grant to All Objects**. Click **Next** to proceed.
5. In the **Specify authenticated welcome** page, click on the **Redirect Upon Successful Authentication** checkbox and specify your main application page. You can click on the **Generate Default** checkbox to generate a default `welcome.jspx` page. Click **Next** to proceed. In the **Summary** page, review your selections and click **Finish** to complete the security configuration wizard.

6. Select **Users** from the **Application** | **Secure** menu to open the `jazn-data.xml` security configuration overview editor. With the **Users** tab selected, create a user called `user1` and assign to it the `test-all` role. Use `user1234` for the password.



7. Locate the `web.xml` deployment descriptor in the **Web Content** | **WEB-INF** folder in the **Application Navigator** and double-click on it to open it. Click on the **Source** tab. Change the `success_url` parameter value for the `adfAuthentication` servlet to `/faces/welcome.jspx`.

8. Add the following welcome file list to the `web.xml` deployment descriptor:

```
<welcome-file-list>
  <welcome-file>/faces/welcome.jspx</welcome-file>
</welcome-file-list>
```

9. Open the `welcome.jspx` page in the page editor and add a Button (`af:commandButton`) component. Change the button's `text` property to **Logout**. Using the property menu add an **Action** to a method called `logout` defined in a newly created managed bean called `AuthenticationBean`.

10. Open the `AuthenticationBean` managed bean in the Java editor and add the following code to the `logout()` method:

```
// create a dispatcher and forward to the login.html page
final String LOGOUT_URL =
  "/adfAuthentication?logout=true&end_url=login.html";
FacesContext ctx = FacesContext.getCurrentInstance();
HttpServletRequest request =
  (HttpServletRequest)ctx.getExternalContext().getRequest();
```

```
        HttpServletResponse response =
          (HttpServletResponse)ctx.getExternalContext()
        getResponse();
        RequestDispatcher dispatcher =
          request.getRequestDispatcher(LOGOUT_URL);
        try {
          dispatcher.forward(request, response);
        } catch (Exception e) {
        // log exception
        }
        ctx.responseComplete();
        return null;
```

How it works...

To enable ADF security for our ADF Fusion web application, we have used the **Configure ADF Security** wizard, available in JDeveloper through the **Application** | **Secure** | **Configure ADF Security...** menu selection. Using the wizard will allow us to enable security in a declarative manner as it will create all related security artifacts, including a login page, an error page, redirection upon a successful authentication to a specific page (`welcome.jspx` in our case), a `test-all` application role assigned to all application task flows and securable pages, and configuration of the `adfAuthentication` servlet in `web.xml`. We started the ADF security configuration wizard in step 1.

In step 2, we choose to enable both ADF authentication and authorization. This option enables the ADF authentication servlet `adfAuthentication` to enforce access to the application through configured login and logout pages. The `adfAuthentication` servlet is added to `web.xml` deployment descriptor.

This also adds a security constraint for the `adfAuthentication` resource, a security role called `valid-users` to allow all users to access the `adfAuthentication` resource, and a filter mapping to `web.xml`.

The `valid-users` role is mapped in the `weblogic.xml` configuration file to an implicit group called `users` defined in WebLogic. WebLogic configures all authenticated users to be members of the `users` group. The following code snippet from `weblogic.xml` shows this role mapping:

```
<security-role-assignment>
  <role-name>valid-users</role-name>
  <principal-name>users</principal-name>
</security-role-assignment>
```

This step also configures authorization for the application, which is enforced through authorization checks on application resources based on configured application roles assigned to them and to authenticated users.

In step 3, we select the authentication type. In this case, we choose form-based authentication and let the wizard create default login and error pages. You could create your own login page using ADF Faces components and handle the authorization process yourself, as it is demonstrated in *Using a custom login page* in this chapter. The generated login page defines a form with the standard `j_security_check` action, which accepts the username and password as input and passes them to the `j_SecurityCheck` method within the container's security model. The wizard updates the `web.xml` file to indicate form-based authentication and identify the login and error pages as shown in the following code snippet:

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

In step 4, by selecting **Enable automatic policy grants**, we allow the wizard to create a `test-all` application role and assign it to all application resources. This will allow us to create users with full access to the application resources once we assign the `test-all` role to them. At a later phase of the application development process, you should remove this role. Also, note that the `test-all` role is granted to anonymous users as well.

In step 5, we create a default welcome page that we will be redirected to upon a successful authentication. This option added the `success_url` initialization parameter to the `adfAuthentication` servlet. We have prepended the `welcome.jspx` page with `/faces/` (see step 7) since we will be adding ADF Faces components to it in step 9. Step 5 completes the security wizard. For a complete list of the files that are updated by the security wizard and their changes, consult the table *Files Updated for ADF Authentication and Authorization* in section *What Happens When You Enable ADF Security* of the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

In step 6, we create a user called `user1`. We will use this user to test the recipe. We map the `test-all` application role to the user to allow access to all of the application resources.

We add a `welcome-file-list` configuration to the `web.xml` file indicating our `/faces/welcome.jspx` welcome page in step 8, so that we will be successfully redirected to the welcome page upon successful authentication. This will allow us to test the recipe by running the `login.html` page.

In step 9, we added a Button component to the welcome page to perform the application log out. Log out is done by defining an action called `logout` implemented by an `AuthenticationBean` managed bean. The `logout` action was implemented by the `logout()` method in step 10. The method creates a `RequestDispatcher` for the logout URL and calls its `forward()` method to redirect the request. The logout URL passes a `logout` parameter to the `adfAuthentication` servlet with the value `true` to indicate a logout action. It also specifies an `end_url` parameter to the `adfAuthentication` servlet with the `login.html` URL. This in effect logs us out of the application and redirects us back to the `login.html` page.

To test the recipe, right-click on the `login.html` page and go through the authorization process. You can log in using the `user1/user1234` credentials. Upon successful authorization, you will be forwarded to the `welcome.jspx` page. Click on the **Logout** button to log out from the application.

There's more...

Note that the Configure ADF Security wizard does not enable authorization for pages that are not associated with databound components, that is, they neither have associated page definition bindings and they are not associated with a specific task flow. In such cases, these pages appear in the **Resource Grants** section of the `jazn-data.xml` overview editor as unsecurable pages. The welcome page, `welcome.jspx` page in this recipe, is one such case. You can still enforce authorization checking in these cases by creating an empty page definition file for the page. This is done by right-clicking on the page and selecting **Go to Page Definition** from the context menu. In the **Confirm Create New Page Definition** dialog click on the **Yes** button to proceed with the creation of the page definition file.

For more information about enabling and configuring ADF security, consult chapter *Enabling ADF Security in a Fusion Web Application* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using a custom login page*, Chapter 9, *Handling Security, Session Timeouts, Exceptions and Errors*

Using a custom login page

In the recipe *Enabling ADF security* in this chapter, we've seen how to enable ADF security for an ADF Fusion web application using the **Configure ADF Security** wizard (available in JDeveloper through the **Application** | **Secure** menu). In one of the steps, the wizard allows for the creation of a default login page that handles the user authorization process. For the specific step in that recipe, we have chosen to create a default login page.

In this recipe, we will create a custom login page utilizing ADF Faces components. Moreover, we will handle the user authentication ourselves using custom login authentication code implemented by the `AuthenticationBean` managed bean. This managed bean was introduced in the *Enabling ADF security* recipe in this chapter.

Getting ready

You need to complete the *Enabling ADF security* recipe in this chapter before you start working on this recipe. The *Enabling ADF security* recipe requires a skeleton **Fusion Web Application (ADF)** workspace. For this purpose, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `MainApplication` workspace in JDeveloper. Locate and open the `AuthenticationBean` managed bean in the Java editor. Add the following code to it:

```
private String username;
private String password;
public void setUsername(String username) {
  this.username = username.toLowerCase();
}
public String getUsername() {
  return this.username;
}
public void setPassword(String password) {
  this.password = password;
}
public String getPassword() {
  return this.password;
}
public String login() {
  final String WELCOME_URL =
    "/adfAuthentication?success_url=/faces/welcome.jspx";
```

```
      FacesContext ctx = FacesContext.getCurrentInstance();
      HttpServletRequest request =
        (HttpServletRequest)ctx.getExternalContext().getRequest();
      if (authenticate(request)) {
        HttpServletResponse response =
          (HttpServletResponse)ctx.getExternalContext().getResponse();
        RequestDispatcher dispatcher =
          request.getRequestDispatcher(WELCOME_URL);
        try {
          dispatcher.forward(request, response);
        } catch (Exception e) {
          reportLoginError(e.getMessage());
        }
        ctx.responseComplete();
      }
      return null;
    }
    private boolean authenticate(HttpServletRequest request) {
      String password = getPassword() == null ? "" : getPassword();
      CallbackHandler handler = new URLCallbackHandler(
        getUsername(), password.getBytes());
      boolean authenticated = false;
      try {
        Subject subject = Authentication.login(handler);
        ServletAuthentication.runAs(subject, request);
        ServletAuthentication.generateNewSessionID(request);
        authenticated = true;
      } catch (FailedLoginException failedLoginException) {
        reportLoginError("Wrong credentials specified.");
      } catch (LoginException loginException) {
        reportLoginError(loginException.getMessage());
      }
      return authenticated;
    }
    private void reportLoginError(String errorMessage) {
      FacesMessage fm = new FacesMessage(
      FacesMessage.SEVERITY_ERROR, null, errorMessage);
      FacesContext ctx = FacesContext.getCurrentInstance();
      ctx.addMessage(null, fm);
    }
```

2. Create a page called `login.jspx` based on a quick start layout.

3. Using the **Component Palette**, drop two Input Text (`af:inputText`) components on the page, one for the username and another for the password. Set the **Secret** property of the password input text to `true`. In addition, set the `value` attribute of the username and password input text components (you can use the **Expression Builder** dialog to do this) to the expressions `#{AuthenticationBean.username}` and `#{AuthenticationBean.username}` respectively.
4. Drop a Button (`af:commandButton`) component on the login page and change its `text` property to **Login**. Moreover, set the button's `action` property to the expression `#{AuthenticationBean.login}`.
5. Open the `web.xml` deployment descriptor located in the **Web Content** | **WEB-INF** folder in the **Application Navigator** and switch to the **Security** tab. Change the **Login Page** to `/faces/login.jspx`.
6. Finally, change the `LOGOUT_URL` constant definition in the `logout()` method of the `AuthenticationBean` managed bean to `/adfAuthentication?logout=true&end_url=/faces/login.jspx`.

How it works...

In step 1, we added a `login()` method to the `AuthenticationBean` managed bean (introduced in the *Enabling ADF security* recipe in this chapter to handle the logout functionality), to handle the user authentication process. The `login()` method is set to the `action` property of the **Login** button added to the login page in step 4. To authenticate the user, we call the `authenticate()` helper method from the `login()` method. The code in `authenticate()` retrieves the username and password values supplied by the user and calls the static `Authentication.login()` to create a `javax.security.auth.Subject`. It subsequently uses the `Subject` when calling `ServletAuthentication.runAs()` to authenticate the request. The authentication process completes by calling `ServletAuthentication.generateNewSessionID()` to generate a new session identifier. Once the user is authenticated, the request is forwarded to the welcome page. This is done by calling `forward()` on a `RequestDispatcher` object and specifying the welcome page URL. The welcome page URL is specified using the parameter `success_url` to the `adfAuthentication` servlet. It is identified by the constant definition `WELCOME_URL`, which is defined as: `/adfAuthentication?success_url=/faces/welcome.jspx`.

Furthermore, we have added setters and getters for the username and password. These are specified as `value` attributes to the corresponding username and password input text components that are added to the login page in step 3.

In steps 2 through 4, we created the custom login page. We added two input text fields that we will use to specify the login credentials (username and password), and a **Login** button, which when pressed will initiate the authentication process. The username and password input text `value` attributes are bound to the `username` and `password` attributes of the `AuthenticationBean` managed bean respectively. Moreover, the **Login** button `action` property is set to the `login()` method of the `AuthenticationBean` managed bean.

In step 5, we updated the login page configuration in `web.xml` to point to the custom login page. Note how we prepended the login page URL with `/faces/` to allow processing of the page by the faces servlet, since it contains ADF Faces components.

Finally, we updated the `LOGOUT_URL` constant used by the `logout()` method in the `AuthenticationBean` managed bean, so that we are redirected to our custom login page instead.

To test the recipe, right-click on the `login.jspx` page and go through the authorization process. You can use the `user1/user1234` credentials. Upon successful authorization, you should be forwarded to the `welcome.jspx` page.

There's more...

Note that the way we have explained programmatic authentication in this recipe is proprietary to the WebLogic Server. The recipe will have to be adapted using similar APIs offered by other application servers.

For more information about creating a custom login page, consult section *Creating a Login Page* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Enabling ADF security*, Chapter 9, *Handling Security, Session Timeouts, Exceptions and Errors*

Accessing the application's security information

You can access the application's security information at the ViewController layer either through Java code in a managed bean or through Expression Language (EL) in your JSF pages by utilizing the methods available via the `oracle.adf.share.security.SecurityContext` bean. These methods will allow you to determine whether authorization and/or authentication are enabled in your application, the roles assigned to the authenticated user, whether the user is assigned a specific role, and so on. At the ADF-BC level, security information can be accessed through the methods available in the `oracle.jbo.Session`.

In this recipe, we will see how to access the application's security information from a managed bean, a JSF page and at the ADF-BC level.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this purpose, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. The recipe assumes that you have enabled ADF security by completing recipes *Enabling ADF security* and *Using a custom login page* in this chapter.

The recipe also uses the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*. *Chapter 2*, *Dealing with Basics: Entity Objects*.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Open the `HRComponents` workspace and locate the `Employees` custom row implementation Java class `EmployeesRowImpl.java` in the **Application Navigator**. Double-click on it to open it in the Java editor.
2. Override the `isAttributeUpdateable()` method and add the following code to it:

```
// allow employee changes only if the user has the
// 'AllowEmployeeChanges' role
return ADFContext.getCurrent().getSecurityContext().isUserInRole(
  "AllowEmployeeChanges")
   ? super.isAttributeUpdateable(i) : false;
```

3. Redeploy the `HRComponents` workspace to an ADF Library JAR.
4. Open the `MainApplication` workspace. Add the `HRComponents` ADF Library JAR to the ViewController project.
5. Create a JSPX page called `applicationSecurity.jspx` and drop the `Employees` collection (available under the `HrComponentsAppModuleDataControl` data control in the **Data Controls** window) as an **ADF Form** on it.
6. Add a Button (`af:commandButton`) component to the page. Using the **Property Inspector**, change the button's **Disabled** property to `#{securityContext.userInRole['AllowEmployeeChanges'] eq false}`.
7. Add an action listener to the button component by defining a new managed bean called `ApplicationSecurityBean` and a method called `onApplicationSecurity`.

8. Open the `ApplicationSecurityBean` managed bean and add the following code to the `onApplicationSecurity()` method:

   ```
   // check for user having the 'AllowEmployeeChanges' role
   if (ADFContext.getCurrent().getSecurityContext()
     .isUserInRole("AllowEmployeeChanges")) {
     FacesContext context =
       FacesContext.getCurrentInstance();
     context.addMessage(null,new FacesMessage
       (FacesMessage.SEVERITY_INFO, "User is allowed to
       edit the employee data.", null));
   }
   ```

9. Select **Application Roles** from the **Application** | **Secure** menu. Create a new application role called `AllowEmployeeChanges`. Click on the **Add User or Role** button (the green plus sign icon) in the **Mappings** section, then **Add User** to map the `user1` user to the `AllowEmployeeChanges` role.
10. Select **Resource Grants** from the **Application** | **Secure** menu. Select **Web Page** for the **Resource Type** and locate the **applicationSecurity** page. Click on the **Add Grantee** button (the green plus sign icon) in the **Granted To** section, then select **Add Application Role** from the menu. Add the `AllowEmployeeChanges` role ensuring that the **view** action in the **Actions** section is selected.

How it works...

In steps 1 through 3, we have updated the `HRComponents` ADF Library JAR in order to demonstrate how to access the application security information at the business components level. Specifically, we have overridden the `ViewRowImpl isAttributeUpdateable()` method for the custom `EmployeesRowImpl` row implementation class, in order to control the `Employees` view object attributes that can be updated based on a specific role assigned to the currently authorized user. We did this by calling `isUserInRole()` on the `oracle.jbo.Session` and specifying the specific role, `AllowEmployeeChanges` in this case. We obtained the `Session` object from the application module by calling `getSession()`. The effect of adding this piece of code is that if the authorized user does not have the `AllowEmployeeChanges` role, none of the `Employees` attributes will be updatable. The `HRComponents` workspace is deployed to an ADF Library JAR in step 3.

In steps 4 through 6, we created a page called `applicationSecurity.jspx` for the main application and dropped in it the `Employees` collection as an ADF Form (step 5). The `Employees` collection is available under the `HrComponentsAppModuleDataControl` data control in the **Data Controls** window once we add the `HRComponents` ADF Library JAR to the ViewController project of the application (which we did in step 4). In step 6, we added an `af:commandButton` component to the page and set its `Disabled` property to the EL expression `#{securityContext.userInRole['AllowEmployeeChanges'] eq false}`. This expression uses the `SecurityContext` bean to check whether the currently authorized user has the `AllowEmployeeChanges` role assigned to it. If the user does not have the role assigned, the button will appear on the page disabled.

In steps 7 and 8, we added an action listener to the button, specifying the `onApplicationSecurity` method on a newly created bean. We have added code to the `onApplicationSecurity()` action listener to call the `SecurityContext` bean `isUserInRole()` method to determine whether the current user has been assigned the `AllowEmployeeChanges` role. For the purpose of this recipe, we display a message if the user is authorized to edit the employee data.

Finally, in steps 9 and 10, we added an application role called `AllowEmployeeChanges` and mapped it to the `user1` user. We also enabled view access to the `applicationSecurity.jspx` page by adding the `AllowEmployeeChanges` role to it.

Observe what happens when you run the main application: if you run the `applicationSecurity.jspx` page by right-clicking on it in the **Application Navigator**, the employee information and the button in the page are disabled. This is because we have not gone through the user authorization process and the current anonymous user does not have the `AllowEmployeeChanges` role. This is not the case if you access the `applicationSecurity.jspx` page after a successful authorization. For this purpose, we have updated the `welcome.jspx` page, the page that we are redirected to upon a successful log in, and added a link to the `applicationSecurity.jspx` page. Observe in this case that both the employee fields and the button in the page are enabled.

There's more...

Some of the other commonly used `SecurityContext` methods and/or expressions are listed in the following table:

Method/Expression	Description
`#{secrityContext.taskflowViewable['SomeTaskFlow']}`	Returns `true` if the user has access to the specific `SomeTaskFlow` task flow.
`#{secrityContext.regionViewable['SomePageDef']}`	Returns `true` if the user has access to the specific `SomePageDef` page definition file associated with a page.
`#{secrityContext.userName}`	Returns the authenticated user's username.

Method/Expression	Description
`#{secrityContext.authenticated}`	Returns `true` if the user has been authenticated.
`#{securityContext.userInAllRoles['roleList']}`	Returns true if the user has all roles in the comma-separated `roleList` assigned.

For a comprehensive list of the `SecurityContext` EL expressions take a look at the section *Using Expression Language (EL) with ADF Security* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

Note that you can access the `SecurityContext` bean at the business components layer using the `adf.context.securityContext` Groovy expression. For instance, to get the username of the currently authorized user, use the expression `adf.context.securityContext.userName`.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*
- *Enabling ADF security*, Chapter 9, *Handling Security, Session Timeouts, Exceptions and Errors*
- *Using a custom login page*, Chapter 9, *Handling Security, Session Timeouts, Exceptions and Errors*

Using OPSS to retrieve the authenticated user's profile from the identity store

Oracle Platform Security Services (OPSS) is a comprehensive standards-based security framework and the underlying security-providing platform for Oracle Fusion Middleware. It provides an abstract layer through the use of an Application Programming Interface (API) for accessing security provider and identity management details. It is through the use of the OPSS API that generic access is achieved to vendor-specific security providers.

In this recipe, we will introduce the OPSS framework by implementing the following use case: using the `HR` schema, for an authenticated employee-user, we will update the employee information in the `EMPLOYEES` table with information from the user's profile obtained from the identity store. For an authenticated employee-user who is not already in the `EMPLOYEES` table, we will create a new row in it.

Getting ready

This recipe adds a security utility helper class to the `SharedComponents` workspace. This workspace was introduced in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. It also updates the `UserInfo` application module introduced in *Using a session scope bean to preserve session-wide information*, *Chapter 8*, *Backing not Baking: Bean Recipes*. The `UserInfo` application module resides in the `HRComponents` workspace, which was also created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this purpose, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also requires that ADF security is enabled for the main application (see additional recipes *Enabling ADF security* and *Using a custom login page* in this chapter) and the presence of a main application page (we will use the `welcome.jspx` page developed for the *Enabling ADF security* recipe).

Finally, you need a connection to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace and add the following `SecurityUtils` helper class to it:

```
public class SecurityUtils {
  private static ADFLogger LOGGER =
    ADFLogger.createADFLogger(SecurityUtils.class);
  public static UserProfile getUserIdentityStoreProfile(
    String username) {
    UserProfile userProfile = null;
    try {
      // get the identity store
      IdentityStore idStore = getIdentityStore();
      // create a search filter based on the
      // specific username
      SimpleSearchFilter filter =
        idStore.getSimpleSearchFilter(UserProfile.NAME,
        SimpleSearchFilter.TYPE_EQUAL, username);
      SearchParameters sp = new SearchParameters(filter,
        SearchParameters.SEARCH_USERS_ONLY);
      // search identity store
      SearchResponse response = idStore.search(sp);
```

```
        // check for search results
        if (response.hasNext()) {
          User user = (User)response.next();
          if (user != null) {
            // retrieve the user profile
            userProfile = user.getUserProfile();
          }
        }
      } catch (Exception e) {
        LOGGER.severe(e);
      }
      // return the user profile
      return userProfile;
    }
    private static IdentityStore getIdentityStore()
      throws JpsException {
      // get the JPS context
      JpsContext jpsCtx = JpsContextFactory.getContextFactory()
        .getContext();
      // return the identity store
      IdentityStoreService service =
        jpsCtx.getServiceInstance(IdentityStoreService.class);
      return service.getIdmStore();
    }
  }
```

2. Redeploy the `SharedComponents` workspace to an ADF Library JAR.
3. Open the `HRComponents` workspace and add the `SharedComponents` ADF Library JAR to the `HRComponentsBC` business components project.
4. Locate the `UserInfoAppModuleImpl.java` custom application module implementation class in the **Application Navigator**. Double-click on it to open it in the Java editor. Add the following methods to it:

```
public void synchronizeEmployee() {
  try {
    // get information for currently logged-in user
    // from identity store
    UserProfile userProfile = SecurityUtils
      .getUserIdentityStoreProfile(getUserPrincipalName());
    if (userProfile != null) {
      // get EMPLOYEES row from currently logged-in user
      UserInfoImpl employees = (UserInfoImpl)getUserInfo();
      employees.executeQuery();
      UserInfoRowImpl employee =
        (UserInfoRowImpl)employees.first();
      // if user is not in EMPLOYEES table, add it
      if (employee == null) {
        addEmployee(employees, userProfile);
      } else { // user in EMPLOYEES table
        String email = userProfile.getBusinessEmail();
        if (email != null &&
          !email.equals(employee.getEmail())) {
          employee.setEmail(email);
        }
      }
      // commit transaction
      this.getDBTransaction().commit();
      // requery users to fetch any calculated attributes
      employees.executeQuery();
    }
  } catch (Exception e) {
    // log exception
  }
}
private void addEmployee(UserInfoImpl employees,
  UserProfile userProfile) throws IMException {
  // create employee row
  UserInfoRowImpl employee =
    (UserInfoRowImpl)employees.createRow();
  // set required employee row data from
  // identity store profile
  employee.setLastName(getUserPrincipalName());
  employee.setEmail(userProfile.getBusinessEmail()== null ?
     "n/a" : userProfile.getBusinessEmail());
  employee.setHireDate(new Date(
    new Timestamp(System.currentTimeMillis())));
  employee.setJobId("IT_PROG");
  employee.setDepartmentId(new Number(60));
  // add employee row
  employees.insertRow(employee);
}
```

5. Add the `synchronizeEmployee()` method to the `UserInfoAppModule` application module client interface and redeploy the `HRComponents` workspace to an ADF Library JAR.
6. Open the `MainApplication` workspace and add the `HRComponents` ADF Library JAR to the ViewController project.
7. Create a bounded task flow called `syncEmployeesTaskFlow`. Using the **Property Inspector** change the **URL Invoke** property to **url-invoke-allowed**.

8. Expand the `UserInfoAppModuleDataControl` in the **Data Controls** window and drop the `synchronizeEmployee()` method to the `syncEmployeesTaskFlow` task flow.

9. Create a managed bean called `SyncEmployeesBean`, and add the following methods to it:

```
public String getProgrammaticallyInvokeTaskFlow() {
  // setup task flow parameters
  Map<String, Object> parameters =
    new java.util.HashMap<String, Object>();
  // construct and return the task flow's URL
  return getTaskFlowURL("/WEB-INF/taskflows/chapter9/
syncEmployeesTaskFlow.xml#syncEmployeesTaskFlow", parameters);
}
private String getTaskFlowURL(String taskFlowSpecs,
  Map<String, Object> parameters) {
  // create a TaskFlowId from the task flow specification
  TaskFlowId tfid = TaskFlowId.parse(taskFlowSpecs);
  // construct the task flow URL
  String taskFlowURL =
    ControllerContext.getInstance().getTaskFlowURL(
    false, tfid, parameters);
  // remove the application context path from the URL
  FacesContext fc = FacesContext.getCurrentInstance();
  String taskFlowContextPath =
    fc.getExternalContext().getRequestContextPath();
  return taskFlowURL.replaceFirst(taskFlowContextPath, "");
}
```

10. Open the `welcome.jspx` page. Using the **Component Palette**, drop a **Link (Go)** (`af:goLink`) component to the page and set the link's **Destination** property to `#{SyncEmployeesBean.programmaticallyInvokeTaskFlow}`.

How it works...

In steps 1 and 2 we have added a helper class called `SecurityUtils` to the `SharedComponents` workspace. We will use this class to retrieve the user's profile from the identity store. For this purpose, we have implemented the method `getUserIdentityStoreProfile()`. To search for the specific username in the identity store, it calls the `IdentityStore` `search()` method. The username specified for the search is passed as an argument to the `getUserIdentityStoreProfile()`. The search yields an `oracle.security.idm.SearchResponse`, which is then iterated to retrieve an `oracle.security.idm.User` identity. We retrieve the user's identity store profile by calling `getUserProfile()` on the `User` object.

In step 2, we have redeployed the `SharedComponents` workspace to an ADF Library JAR.

In steps 3 through 5, we have made the necessary changes to the `UserInfoAppModule` application module to allow for the synchronization of employee-users. We have assumed that each employee in the `EMPLOYEES HR` schema table is also a user of the application. In step 3, we have added the `SharedComponents` ADF Library JAR to the `HRComponents` business components project so that we can make use of the `SecurityUtils` helper class. Then, in step 4, we implemented a method called `synchronizeEmployee()`, to allow for the synchronization of the `EMPLOYEES` table. This method is also exposed to the application module's client interface (in step 5), so that it can be invoked from the ViewController layer as an operation binding.

The synchronization of the `EMPLOYEES` table is based on the following logic: if the currently authorized user is not in the `EMPLOYEES` table, it is added. Information from the user's identity store profile is used to populate the `EMPLOYEES` table fields. If the user is already in the `EMPLOYEES` table, the user's information in the `EMPLOYEES` table is updated with the information from the user's identity store profile. The currently authorized user is searched in the database using the `UserInfo` view object. If the user is not found in the database, we call `addEmployee()` to add it. Otherwise, the user's information in the database is updated. Note that the query used by the `UserInfo` view object uses a `WHERE` clause that is based on the currently authorized user. This is done in a declarative manner by specifying the Groovy expression `adf.context.securityContext.userName` for the binding variable `inEmployeeName` used by the `UserInfo` view object query.

In steps 6 through 8, we have created a task flow called `syncEmployeesTaskFlow` and dropped in it the `synchronizeEmployee()` method as a method call activity. The `synchronizeEmployee()` method is available under the `UserInfoAppModuleDataControl` in the **Data Controls** window once the `HRComponents` ADF Library JAR is added to the project (this is done in step 6). Observe how in step 7 we set the task flow **URL Invoke** property to `url-invoke-allowed`. This will allow us to invoke the `syncEmployeesTaskFlow` task flow using its URL.

Steps 9 and 10 are added only so that we can test the recipe. For more information about the code in the `getProgrammaticallyInvokeTaskFlow()` and `getTaskFlowURL()` methods, take a look at *Calling a task flow as a URL programmatically*, *Chapter 6*, *Go with the flow: Task Flows*.

To test the recipe, right-click on the `login.jspx` page and go through the authorization process. You can use the `user1/user1234` credentials. Upon a successful authorization, you will be forwarded to the `welcome.jspx` page. Click on the `syncEmployeesTaskFlow.xml` link. Observe that for new employee-users, the user information is added to the `EMPLOYEES HR` table. For existing employee-users, the user information is updated in the table.

There's more...

One of the hurdles in getting the OPSS framework to work for your specific application security environment involves the proper configuration of the Identity Store Service. Configuration is done through the `jps-config.xml` file located in the `config/fmwconfig` folder under the domain directory in WebLogic. For example, in order to configure OPSS when multiple LDAP authenticators are used in WebLogic, you will need to set up the `virtualize` property in WebLogic. For a comprehensive reference on OPSS configuration, consult section *Configuring the Identity Store Service* in the *Fusion Middleware Application Security Guide*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Calling a task flow as a URL programmatically*, *Chapter 6*, *Go with the flow: Task Flows*
- *Using a session scope bean to preserve session-wide information*, *Chapter 8*, *Backing not Baking: Bean Recipes*
- *Enabling ADF security*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions and Errors*
- *Using a custom login page*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions and Errors*

Detecting and handling session timeouts

Each time a client request is sent to the server a predefined, application-wide, configurable session timeout value is written to the page to determine when a session timeout should occur. A page is considered eligible to timeout if there is no keyboard, mouse or any other programmatic activity on the page. Moreover, an additional application configuration option exists to warn the user sometime prior to the session expiration that a timeout is imminent.

In this recipe, we will see how to gracefully handle a session timeout by redirecting the application to a specific page, the login page in this case, once a session timeout is detected.

Getting ready

You will need to create a skeleton **Fusion Web Application (ADF)** workspace before you proceed with this recipe. For this purpose, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `MainApplication` workspace in JDeveloper. Add the following `SessionTimeoutFilter` filter to the ViewController project:

```
public class SessionTimeoutFilter implements Filter {
  private FilterConfig filterConfig = null;
  public SessionTimeoutFilter() {
    super();
  }
  @Override
  public void init(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
  }
  @Override
  public void destroy() {
    filterConfig = null;
  }
  @Override
  public void doFilter(ServletRequest servletRequest,
    ServletResponse servletResponse, FilterChain filterChain)
    throws IOException, ServletException {
    // get requested session
    String requestedSession =
      ((HttpServletRequest)servletRequest)
        .getRequestedSessionId();
    // get current session
    String currentSession =
      ((HttpServletRequest)servletRequest).getSession().getId();

    // check for invalid session
    if (currentSession.equalsIgnoreCase(requestedSession)
                 == false && requestedSession != null) {
      // the session has expired or renewed
      // redirect request to the page defined by the
      // SessionTimeoutRedirect parameter
      ((HttpServletResponse)servletResponse)
        .sendRedirect(((HttpServletRequest)
         servletRequest).getContextPath()
        + ((HttpServletRequest)servletRequest)
                .getServletPath()
              + "/" + filterConfig.getInitParameter(
                  "SessionTimeoutRedirect"));
    } else {
      // current session is still valid
```

```
            filterChain.doFilter(servletRequest, servletResponse);
          }
        }
      }
```

2. Open the `web.xml` deployment descriptor and add the following `filter` and `filter-mapping` definitions to it. Make sure that you add these definitions at the end of any other `filter` and `filter-mapping` definitions.

```
<filter>
  <filter-name>SessionTimeoutFilter</filter-name>
  <filter-class>com.packt.jdeveloper.cookbook.
    hr.main.view.filters.SessionTimeoutFilter</filter-class>
  <init-param>
    <param-name>SessionTimeoutRedirect</param-name>
    <param-value>/faces/login.jspx</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>SessionTimeoutFilter</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```

3. For testing purposes add the following session-timeout configuration to `web.xml`:

```
<session-config>
  <session-timeout>4</session-timeout>
</session-config>
```

How it works...

In step 1, we have added a filter called `SessionTimeoutFilter`. In it, we obtain from the request both the session identifier of the current request and the identifier of the current session. We compare the session identifiers, and if they differ we redirect the user to the page identified by the `SessionTimeoutRedirect` filter initialization parameter. A difference in the session identifiers indicates that a session timeout has occurred. We have set the `SessionTimeoutRedirect` filter parameter to the `login.jspx` page in step 2. Also in step 2, we have added the `SessionTimeoutFilter` filter definition to the `web.xml` deployment descriptor.

Finally, for testing purposes only, we have set the application-wide session timeout to 4 minutes in step 3.

There's more...

In addition to the `session-timeout` configuration setting in `web.xml`, you can configure a session timeout warning interval by defining the context parameter `oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_TIMEOUT`. This parameter is set to a number of seconds before the actual session timeout would occur and raises a warning dialog indicating that the session is about to expire. You then have the opportunity to extend the session by performing some activity on the page. Note that if its value is set to less than 120 seconds, this feature might be disabled under certain conditions.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

Using a custom error handler to customize how exceptions are reported to the ViewController

You can alter the way error messages are reported to the ADF Controller by implementing a custom error handler class that extends the `oracle.adf.model.binding.DCErrorHandlerImpl` class. The custom error handler class can then provide custom implementations for the following methods:

- `reportException()`: This method is called by the ADF framework to report an exception. You can override this method to handle how each exception type is reported.
- `getDisplayMessage()`: Returns the exception error message. You can override this method in order to change the error message.
- `getDetailedDisplayMessage()`: Returns the exception error message details. You can override this method in order to change the error message details.

This recipe shows you how to extend the `DCErrorHandlerImpl` error handling class so that you can provide custom handling and reporting of the application exceptions to the ViewController layer.

Getting ready

We will add the custom error handler to the `SharedComponents` workspace. This workspace was created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

For testing purposes, you will need to create a skeleton **Fusion Web Application (ADF)** workspace. For this purpose, we will use the `MainApplication` workspace that was developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `SharedComponents` workspace and add the following `CustomDCErrorHandlerImpl` class to its ViewController project:

```
public class CustomDCErrorHandlerImpl
  extends DCErrorHandlerImpl {
  public CustomDCErrorHandlerImpl() {
    super(true);
  }
  public void reportException(DCBindingContainer
    dCBindingContainer,Exception exception) {
    // report JboExceptions as errors
    if (exception instanceof ExtJboException
      || exception instanceof JboException) {
      FacesContext.getCurrentInstance().addMessage(
        null,new FacesMessage(FacesMessage.SEVERITY_ERROR,
        exception.getMessage(), null));
    } else { // report all others as information
      FacesContext.getCurrentInstance().addMessage(
        null,new FacesMessage(
        FacesMessage.SEVERITY_INFO,
        exception.getMessage(), null));
    }
  }
}
```

2. Redeploy the `SharedComponents` workspace to an ADF Library JAR.
3. Open the `MainApplication` workspace and add the `SharedComponents` ADF Library JAR to its ViewController project.
4. Open the `DataBindings.cpx` Data Binding Registry file and select the root **Databindings** node in the **Structure** window. Using the **Property Menu** next to the **ErrorHandlerClass** in the **Property Inspector**, specify the `CustomDCErrorHandlerImpl` class implemented previously.

How it works...

We have created a custom error handler called `CustomDCErrorHandlerImpl` in steps 1 and 2 as part of the `SharedComponents` workspace. The class extends the default error handling implementation provided by the `oracle.adf.model.binding.DCErrorHandlerImpl` class. We only need to override the `reportException()` method at this time to provide custom handling for the application-generated exceptions. For the purposes of this recipe, we are looking for `ExtJboException` and `JboException` types of exceptions, that is, exceptions generated by the business components layer, and we are displaying them as error Faces messages at the ViewController layer. `ExtJboException` is a custom application exception that was implemented in *Using a custom exception class*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. All other exceptions are shown as informational messages to the user. We make sure that the `SharedComponents` workspace is redeployed as an ADF Library JAR and that is added to the main workspace ViewController project in step 3.

One last thing that we need to do is set the `ErrorHandlerClass` property of the `Databindings` node in the `DataBindings.cpx` bindings registry file to our custom `CustomDCErrorHandlerImpl` class. We do this in step 4.

There's more...

In this recipe, we customized the way the application exceptions are handled and reported to the ViewController layer by providing a custom implementation of the `reportException()` method. To customize the way the actual error message is formatted take a look at the *Customizing the error message details* recipe in this chapter.

For more information about custom error handling in your application, consult the *Customizing Error Handling* section of the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Customizing the error message details*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions and Errors*

Customizing the error message details

In the recipe *Using a custom error handler to customize how exceptions are reported to the ViewController* in this chapter, we've seen how to create a custom `DCErrorHandlerImpl` class and override its `reportException()` method in order to provide custom handling of the application's exceptions. In this recipe, we will go over the process of overriding the `DCErrorHandlerImpl` class `getDisplayMessage()` method, so that we can provide custom handling of specific application error messages. In particular, we will see how to reformat error messages generated by exceptions thrown from the database business logic code, using functionality provided by the ADF resource bundles. More specifically, we will assume that our application's database business logic source code throws exceptions using a user-defined database error number, with the actual resource error number and parameters bundled within the exception message. An example of the error message thrown by the database layer is: `ORA-20200: APPL-00007: Some Error $1parameter1$1 $2parameter2$2`. In this case, the business exception generated by the database is identified by the user-defined error number `-20200`. The actual error message is bundled within the message details and it is identified by the error number `00007`. The error message parameters are delimited by the parameter placeholders `$1` and `$2`.

Getting ready

You will need to complete *Using a custom error handler to customize how exceptions are reported to the ViewController* recipe in this chapter before delving into this recipe.

How to do it...

1. Open the `SharedComponents` workspace and locate the `CustomDCErrorHandlerImpl` class. Open the class in the Java editor and override the `getDisplayMessage()` method.
2. Add the following code to the `getDisplayMessage()` method:

```
// get the error message from the framework
String errorMessageRaw =
  super.getDisplayMessage(bindingContext,exception);
// handle messages generated by the database business logic
return handleDatabaseApplicationError(errorMessageRaw);
```

3. Add the following helper methods to the `CustomDCErrorHandlerImpl` class:

```
private String handleDatabaseApplicationError(
  String errorMessageRaw) {
  // the error code for application-specific messages
  // generated by the database application-specific
  // business code
  final String APPLICATION_ERROR_CODE = "20200";
```

```
// the application error messages bundle
final ResourceBundle errorMessagesBundle =
  ResourceBundle.getBundle("com.packt.jdeveloper.cookbook.
  shared.bc.exceptions.messages.ErrorMessages");
// check for null/empty error message
if (errorMessageRaw == null||"".equals(errorMessageRaw)) {
  return errorMessageRaw;
}
// check for database error message
if (errorMessageRaw.indexOf("ORA-") == -1) {
  return errorMessageRaw;
}
// check for end of database error code indicator
  int endIndex = errorMessageRaw.indexOf(":");
  if (endIndex == -1) {
    return errorMessageRaw;
  }
  // get the database error code
  String dbmsErrorMessageCode =
    errorMessageRaw.substring(4, endIndex);
  String errorMessageCode = "";
  if (APPLICATION_ERROR_CODE.equals(dbmsErrorMessageCode)) {
    int start = errorMessageRaw.indexOf("-", endIndex)+1;
    int end = errorMessageRaw.indexOf(":", start);
    errorMessageCode = errorMessageRaw.substring(
    start, end);
  } else {
    // not application-related error message
    return errorMessageRaw;
  }
  // get the application error message from the
  // application resource bundle using the specific
  // application error code
  String errorMessage = null;
  try {
    errorMessage = errorMessagesBundle.getString(
    "message." + errorMessageCode);
  } catch (MissingResourceException mre) {
    // application error code not found in the bundle,
    // use original message
    return errorMessageRaw;
  }
  // get the error message parameters
  ArrayList parameters =
    getErrorMessageParameters(errorMessageRaw);
```

```
        if (parameters != null && parameters.size() > 0) {
          // replace the message parameter placeholders with the
          // actual parameter values
        int counter = 1;
        for (Object parameter : parameters) {
          // parameter placeholders appear in the message
          // as {1}, {2}, and so on
          errorMessage = errorMessage.replace("{" +
            counter + "}", parameter.toString());
          counter++;
        }
      }
      // return the formated application error message
      return errorMessage;
    }
    private ArrayList getErrorMessageParameters(
      String errorMessageRaw) {
      // the parameter indicator in the database
      // application-specific error
      final String PARAMETER_INDICATOR = "$";
      ArrayList parameters = new ArrayList();
      // get parameters from the error message
      for (int i = 1; i <= 10; i++) {
        int start = errorMessageRaw.indexOf(PARAMETER_INDICATOR + i)
          + 2;
        int end = errorMessageRaw.indexOf(PARAMETER_INDICATOR
          + i, start);
        if (end == -1) {
          parameters.add(i - 1, "");
        } else {
          parameters.add(i - 1,errorMessageRaw.substring(start, end));
        }
      }
      // return the parameters
      return parameters;
    }
```

4. Redeploy the `SharedComponents` workspace into an ADF Library JAR.

How it works...

In steps 1 and 2, we have updated the `CustomDCErrorHandlerImpl` custom error handler class by overriding the `getDisplayMessage()` method. The `CustomDCErrorHandlerImpl` custom error handler class was added to the `SharedComponents` workspace in the recipe *Using a custom error handler to customize how exceptions are reported to the ViewController* in this chapter. By overriding the `getDisplayMessage()` method, we will get a chance to reformat the error message displayed by the application before it is displayed. In our case, we will reformat any messages related to exceptions thrown from the database business logic code. This is done by the helper method `handleDatabaseApplicationError()` added in step 3. This method checks for errors originating from database exceptions by looking for the "`ORA-`" substring in the error message. If this is found, the database business error message number is extracted. This is a user-defined application-specific error message number used in the database business logic code to throw application business logic exceptions. PL/SQL error numbers in the range of `-20000` to `-20999` are reserved for user-defined errors. For this recipe, it is defined by the constant `APPLICATION_ERROR_CODE` and it is equal to `20200` (we parse the error number after the `-`).

If this is indeed a business logic error message, the actual resource error number is bundled within it and it is extracted; the actual error number is saved in the `errorMessageCode` variable. We use this error number to look up the actual error message string in the application resource bundle, which is initialized by the call `ResourceBundle.getBundle()`. We have used the `ErrorMessages.properties` bundle, introduced in *Using a custom exception class*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*, to store the application error messages. Resources in this bundle are identified by error numbers prepended with the "`message.`" string, for instance, `message.00007`. So, we called `getString()` on the resource bundle to locate the actual error message after we prepended the error code with "`message.`". This functionality is also implemented by the `BundleUtils` helper class introduced in *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

We retrieve the parameter values bundled in the database error message by calling the `getErrorMessageParameters()` helper method. This method identifies any parameters bundled in the raw database error message by looking for the parameter placeholder identifiers `$1`, `$2`, and so on. The parameters are added to an `ArrayList`, which is iterated when replacing the parameter placeholder identifiers `{1}`, `{2}`, and so on, in the actual message string.

This is an example error message thrown by the database business logic code: `ORA-20200: APPL-00007: Some Error $1parameter1$1 $2parameter2$2`. The error message defined in the resource bundle for error number 00007 is `message.00007=Message generated by the database business code. Parameters: {1}, {2}`. When we go through our custom `getDisplayMessage()` method, the actual message displayed by the application would be: **Message generated by the database business code. Parameters: parameter1, parameter2**.

The final step redeploys the `SharedComponents` workspace to an ADF Library JAR, so that it can be reused by other application workspaces.

See also

- *Using a custom exception class*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using a custom error handler to customize how exceptions are reported to the ViewController*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions and Errors*

Overriding attribute validation exceptions

At the ADF-BC layer, built-in validators are stored in the XML metadata definition file with no ability to customize the exception message and/or centralize the application error messages in a single application-wide message bundle file. To overcome this you can extend the `oracle.jbo.ValidationException` and `oracle.jbo.AttrValException` classes. Then in your custom entity object implementation class you can override the `validateEntity()` and `setAttributeInternal()` methods to throw these custom exceptions instead. Even better, if you have gone through the process of creating framework extension classes (see *Setting up BC base classes*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*), this functionality can be added to the base entity object framework extension class and thereby used in a generic way throughout the application.

In this recipe, we will extend the `oracle.jbo.AttrValException` class in order to provide a custom attribute validation exception. We will then override the `setAttributeInternal()` method in the entity object framework extension class to throw the custom attribute validation exception.

Getting ready

We will add the custom attribute validation exception to the `SharedComponents` workspace. The `SharedComponents` workspace was created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. Moreover, we will update the entity object framework extension class, which resides in the `SharedComponents` workspace. The entity object framework extension class was created in *Setting up BC base classes*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `SharedComponents` workspace in JDeveloper and add the following `ExtAttrValException` class to the `SharedBC` project:

```
public class ExtAttrValException extends AttrValException {
  public ExtAttrValException(String errorCode,
    Object[] errorParameters) {
  super(ResourceBundle.class, errorCode,
    errorParameters);
  }
  public ExtAttrValException(final String errorCode) {
    super(ResourceBundle.class, errorCode, null);
  }
  public String getMessage() {
    return BundleUtils.loadMessage(this.getErrorCode(),
    this.getErrorParameters());
  }
}
```

2. Open the `ExtEntityImpl` entity object framework extension class in the Java editor and override the `setAttributeInternal()` method.
3. Add the following code to the `setAttributeInternal()` method:

```
try {
  super.setAttributeInternal(attrib, value);
} catch (AttrValException e) {
  // throw custom attribute validation exception
  throw new ExtAttrValException(e.getErrorCode(),
    e.getErrorParameters());
}
```

4. Redeploy the `SharedComponents` workspace to an ADF Library JAR.

How it works...

In step 1, we have extended the `AttrValException` framework attribute validation exception by providing our custom implementation class called `ExtAttrValException`. This overrides the `getMessage()` method, which uses the helper class `BundleUtils` to load the error message from the application-wide message bundle file. Using the specific exception error code, the `BundleUtils` helper class was created in *Using a generic backing bean actions framework*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

In order to utilize the custom attribute validation exception in our application, we have overridden the `setAttributeInternal()` method of the `ExtEntityImpl` entity object framework extension class to throw `ExtAttrValException` instead of `AttrValException`. This was done in steps 2 and 3. The `setAttributeInternal()` method validates and sets the attribute value for the attribute identified by the `attrib` index.

Finally, in step 4, we redeploy the `SharedComponents` workspace to an ADF Library JAR.

There's more...

You can follow similar steps to customize the validation exceptions of your application's entity objects. In this case, you will need to extend the `oracle.jbo.ValidationException` class. Then you will need to override the `validateEntity()` method of the entity object framework extension class to throw your custom validation exception.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Setting up BC base classes*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

10 Deploying ADF Applications

In this chapter, we will cover:

- Configuring and using the Standalone WebLogic Server
- Deploying on the Standalone WebLogic Server
- Using ojdeploy to automate the build process
- Using Hudson as a continuous integration framework

Introduction

The development and testing of ADF Fusion web applications in JDeveloper does not require any special deployment work by you, the developer. JDeveloper does a pretty good job setting up and configuring a WebLogic domain, namely the Integrated WebLogic Server, that is subsequently used to transparently deploy applications. This transparent deployment process takes place each time you choose to run or debug an application in JDeveloper.

To further test ADF Fusion web applications, using an environment that more closely resembles the actual production environment, you ought to consider configuring the Standalone WebLogic Server. This involves the creation of a WebLogic domain configured as closely as possible to the actual production environment (server instances, clusters, security realm configuration, services configuration, and so on) and deploying your ADF Fusion web application to it periodically.

There are a number of techniques used to deploy applications to the Standalone WebLogic Server. For instance, to deploy in a continuous integration production or testing environment, a script technique needs to be considered. On the other hand, for local development and testing purposes, deploying from JDeveloper will suffice.

Configuring and using the Standalone WebLogic Server

JDeveloper Studio Edition ships along with the WebLogic application server included. WebLogic Server is an essential part of the ADF Fusion web application development process, as it allows for the deployment, running, debugging, and testing of your application. It is installed on the development machine during the installation of JDeveloper.

When you choose to run or debug a Fusion web application from within the JDeveloper IDE, WebLogic is started and the application is deployed and run automatically on it. This configuration is called "Integrated WebLogic Server" as it is tightly integrated with the JDeveloper IDE. The very first time an ADF Fusion web application is run (or debugged) in JDeveloper, the necessary integrated WebLogic Server configuration takes place automatically. The configuration process creates the WebLogic domain and a server instance to deploy the application onto.

In addition to the Integrated WebLogic Server, the WebLogic configuration software allows for the creation and configuration of a "standalone" WebLogic domain. This domain that you configure separately according to your specific configuration requirements is known as the Standalone WebLogic Server. This is started independently of JDeveloper, and you deploy your applications on it using a separate deployment process. The Standalone WebLogic Server offers, among others, the following advantages: control over the specific configuration of the WebLogic domain; control over the deployment process; freeing up resources in JDeveloper when debugging and testing; freeing up resources on the development machine (when the WebLogic Server runs on another machine); and the ability to remotely debug the application.

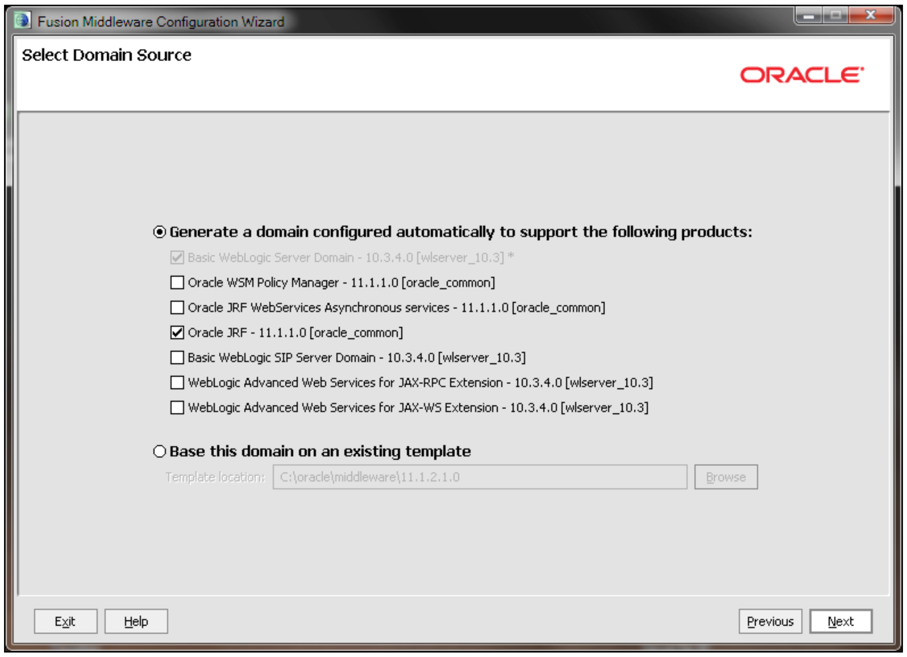
In this recipe, we will go over the steps involved in configuring the Standalone WebLogic Server that we can use subsequently to deploy our ADF Fusion web application.

Getting ready

You will need WebLogic installed on your development environment. WebLogic is installed during the installation of JDeveloper Studio Edition based on your installation choices. For information on installing JDeveloper on a Linux distribution, take a look at the *Installation of JDeveloper on Linux* recipe in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Start the **Fusion Middleware Configuration Wizard**. You do this by running the `config` script located in the `common/bin` directory under the WebLogic installation directory. In the **Welcome** page, select **Create a new WebLogic domain** and click **Next**.
2. In the **Select Domain Source** page, select **Oracle JRF – 11.1.1.0 [oracle_common]** and click **Next**.



(●) Generate a domain configured automatically to support the following products:

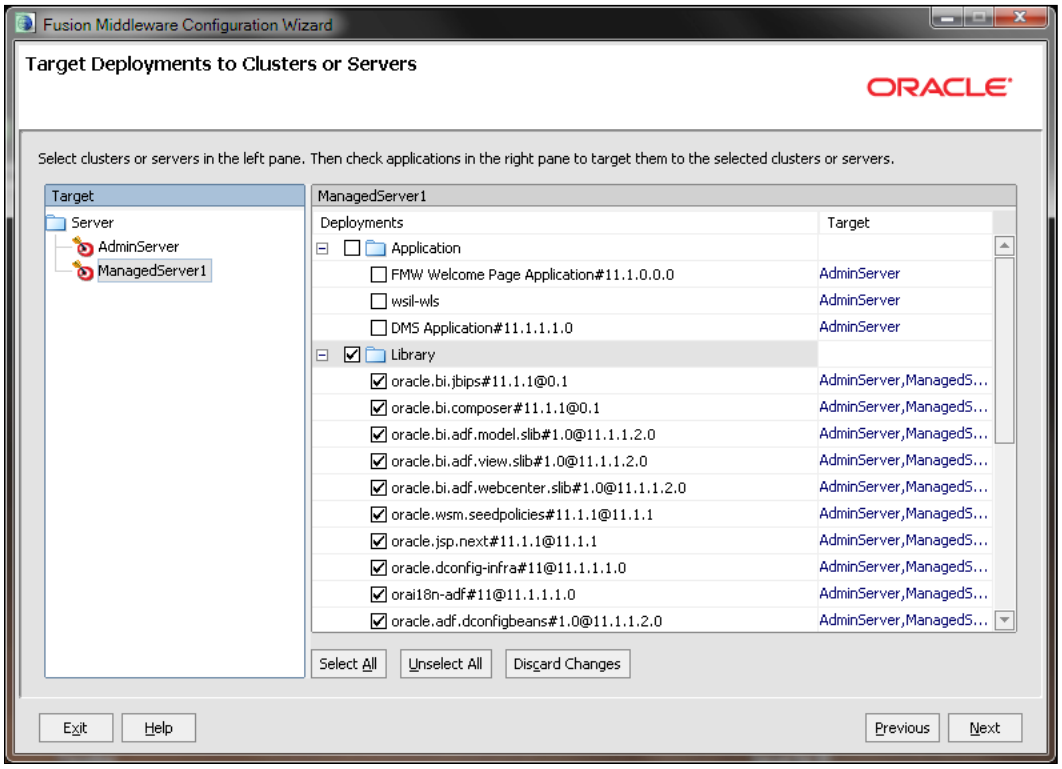
- [✓] Basic WebLogic Server Domain - 10.3.4.0 [wlserver_10.3] *
- [ ] Oracle WSM Policy Manager - 11.1.1.0 [oracle_common]
- [ ] Oracle JRF WebServices Asynchronous services - 11.1.1.0 [oracle_common]
- [✓] Oracle JRF - 11.1.1.0 [oracle_common]
- [ ] Basic WebLogic SIP Server Domain - 10.3.4.0 [wlserver_10.3]
- [ ] WebLogic Advanced Web Services for JAX-RPC Extension - 10.3.4.0 [wlserver_10.3]
- [ ] WebLogic Advanced Web Services for JAX-WS Extension - 10.3.4.0 [wlserver_10.3]

( ) Base this domain on an existing template

Template location: C:\oracle\middleware\11.1.2.1.0 Browse

3. In the **Specify Domain Name and Location** page, enter the **Domain name** and the **Domain location** and click **Next**. You may keep the default values.
4. In the **Configure Administrator User Name and Password** page, enter the domain administrator **Name** and **User password**. Confirm the password and click **Next**.
5. In the **Configure Server Start Mode and JDK** page, select **Development Mode** for the **WebLogic Domain Startup Mode** and the **Sun SDK** from the list of **Available JDKs**. Click **Next** to continue.
6. In the **Select Optional Configuration** page, click on the **Administration Server**, **Managed Servers**, **Clusters and Machines**, and **Deployments and Services** checkboxes. Click **Next** to continue.
7. In the **Configure the Administration Server** page, enter the **Name**, **Listen address**, and **Listen port** of the administration server. You may keep the default values. Then click on the **Next** button.

8. In the **Configure Managed Servers** page, click on the **Add** button and specify the **Name**, **Listen address**, and **Listen port** of the managed server. You may choose the default values. Click **Next** to continue.
9. In the **Configure Clusters** page, click **Next**.
10. In the **Configure Machines** page, click on the **Add** button and specify the machine **Name**, **Node manager listen address**, and **Node manager listen port**. You may keep the default values. Click **Next** to continue.
11. In the **Assign Servers to Machines** page, shuttle the managed server from the **Server** list to the specific machine in the **Machine** list and click **Next**.
12. In the **Target Deployments to Clusters or Servers** page, make sure that all libraries are targeted to both administration and managed servers. You can do this by selecting the managed server in the **Target** list and selecting the **Library** node in the **Deployments** list. Click **Next** to continue.


Fusion Middleware Configuration Wizard

Target Deployments to Clusters or Servers

Select clusters or servers in the left pane. Then check applications in the right pane to target them to the selected clusters or servers.

Target
- Server
  - AdminServer
  - ManagedServer1

ManagedServer1

Deployments	Target
☐ Application	
☐ FMW Welcome Page Application#11.1.0.0.0	AdminServer
☐ wsil-wls	AdminServer
☐ DMS Application#11.1.1.1.0	AdminServer
✓ Library	
✓ oracle.bi.jbips#11.1.1@0.1	AdminServer,ManagedS...
✓ oracle.bi.composer#11.1.1@0.1	AdminServer,ManagedS...
✓ oracle.bi.adf.model.slib#1.0@11.1.1.2.0	AdminServer,ManagedS...
✓ oracle.bi.adf.view.slib#1.0@11.1.1.2.0	AdminServer,ManagedS...
✓ oracle.bi.adf.webcenter.slib#1.0@11.1.1.2.0	AdminServer,ManagedS...
✓ oracle.wsm.seedpolicies#11.1.1@11.1.1	AdminServer,ManagedS...
✓ oracle.jsp.next#11.1.1@11.1.1	AdminServer,ManagedS...
✓ oracle.dconfig-infra#11@11.1.1.1.0	AdminServer,ManagedS...
✓ orai18n-adf#11@11.1.1.1.0	AdminServer,ManagedS...
✓ oracle.adf.dconfigbeans#1.0@11.1.1.2.0	AdminServer,ManagedS...

13. In the **Target Services to Clusters or Servers** page, make sure that all services are targeted to both the administration and managed server. You can do this by selecting the managed server in the **Target** list and clicking on the high level service nodes in the **Service** list. Click **Next** to continue.
14. In the **Configuration Summary** page, verify the domain configuration in the **Domain Summary** and **Details** sections. Click on the **Create** button to proceed with the creation of the domain.
15. Once the domain is created successfully, click on the **Done** button in the **Creating Domain** page to dismiss the configuration wizard.

How it works...

In Steps 1 through 15, we go through the process of creating and configuring a new WebLogic domain. A domain in WebLogic is the basic administrative unit, and it consists of associated resources such as one or more WebLogic server instances, machines, clusters, services, applications, libraries, and others. Creation and configuration of a new WebLogic domain is achieved using the **Fusion Middleware Configuration Wizard** utility. The configuration utility can be started by running the `config` script located in the `common/bin` directory under the WebLogic installation directory. WebLogic is installed in the `wlserver_xx.x` directory under the Middleware home directory, where `xx.x` is the WebLogic Server version. Note that for a Windows installation of JDeveloper, a shortcut is created for the configuration utility, called **Configuration Wizard**, under the **Oracle Fusion Middleware 11.1.2.x.x** | **WebLogic Server 11gR1** | **Tools** group in the **Start** menu.

In step 1 of the domain configuration wizard, we have chosen to create a new domain. You can also choose to extend an existing domain by adding additional extension sources to the domain and/or reconfiguring the domain structure (servers, clusters, machines, and so on).

In step 2, we have selected the **Oracle JRF – 11.1.1.0** domain source. This will install the necessary libraries to the domain in order to support the deployment and execution of ADF Fusion web applications.

We proceed in step 3 to specify the domain name and location. By default, domains are created in the `user_projects/domains` directory under the Middleware home directory. Choosing this default location is acceptable for development purposes. For production installations, you should choose a top level directory independent of the specific WebLogic Server installation.

In step 4, we have specified the domain's administrator username and password. These credentials are necessary to access the domain for administrative purposes either using the `console` application or through any other administration utilities (`WLST`, `weblogic.Deployer`, and so on).

In step 5, we configured the server startup mode to be in development mode. This mode enables the WebLogic auto-deployment feature, which allows for the automatic deployments of applications that reside in the `autodeploy` domain directory. This will be fine for this recipe. In a production environment configuration, production mode should be selected along with the JRockit JDK. For a comprehensive list of differences between the development and production startup modes, take a look at the *Differences Between Development and Production Mode* table in the *Creating Domains Using the Configuration Wizard* documentation. This document is available through the Oracle WebLogic Server Documentation Library currently at `http://docs.oracle.com/cd/E14571_01/wls.htm`.

In step 6, we indicated which components we will be providing additional configuration for. In this case, we configured the administration server, a managed server and its machine, and the deployments and services. For each component, an additional wizard page will be presented to further configure the specific component.

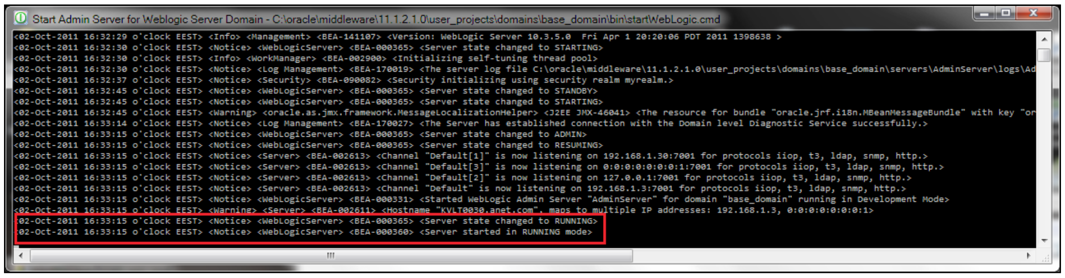
In steps 7 through 11, we created and configured the domain's server instances. The administration server is used to manage the domain, and its creation is required. The managed server will be used to deploy ADF applications. In both cases, we specified the server's name, listen address, and listen port. Managed servers are assigned to WebLogic machines (this is done in step 11). This identifies a physical unit of hardware that is associated with a WebLogic Server instance. They are used in conjunction with the WebLogic node manager to start and shutdown remote servers. Furthermore, WebLogic uses a configured machine in order to delegate tasks, such as HTTP session replication, in a clustered configuration. A machine was created and configured in step 10.

In steps 12 and 13, we have made available all installed product libraries and services, to the managed server instance. This will allow us to deploy and run ADF Fusion web applications on the managed server.

After reviewing the configuration in step 14, we create the domain in step 15.

There's more...

Once the domain creation completes successfully, it can be started by separately starting the administration and managed server instances. To start the administration server, run the `startWebLogic` script located in the domain `bin` directory. When you do so, observe in the console window that the server is started successfully, as shown in the following screenshot:



Start Admin Server for Weblogic Server Domain - C:\oracle\middleware\11.1.2.1.0\user_projects\domains\base_domain\bin\startWebLogic.cmd

```
<02-Oct-2011 16:32:29 o'clock EEST> <Info> <Management> <BEA-141107> <Version: WebLogic Server 10.3.5.0  Fri Apr 1 20:20:06 PDT 2011 1398638 >
<02-Oct-2011 16:32:30 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to STARTING>
<02-Oct-2011 16:32:30 o'clock EEST> <Info> <WorkManager> <BEA-002900> <Initializing self-tuning thread pool>
<02-Oct-2011 16:32:30 o'clock EEST> <Notice> <Log Management> <BEA-170019> <The server log file C:\oracle\middleware\11.1.2.1.0\user_projects\domains\base_domain\servers\AdminServer\logs\Ad
<02-Oct-2011 16:32:37 o'clock EEST> <Notice> <Security> <BEA-090082> <Security initializing using security realm myrealm.>
<02-Oct-2011 16:32:45 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to STANDBY>
<02-Oct-2011 16:32:45 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to STARTING>
<02-Oct-2011 16:32:45 o'clock EEST> <Warning> <oracle.as.jmx.framework.MessageLocalizationHelper> <J2EE JMX-46041> <The resource for bundle "oracle.jrf.i18n.MBeanMessageBundle" with key "or
<02-Oct-2011 16:33:14 o'clock EEST> <Notice> <Log Management> <BEA-170027> <The Server has established connection with the Domain level Diagnostic Service successfully.>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to ADMIN>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RESUMING>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <Server> <BEA-002613> <Channel "Default[1]" is now listening on 192.168.1.30:7001 for protocols iiop, t3, ldap, snmp, http.>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <Server> <BEA-002613> <Channel "Default[3]" is now listening on 0:0:0:0:0:0:0:1:7001 for protocols iiop, t3, ldap, snmp, http.>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <Server> <BEA-002613> <Channel "Default[2]" is now listening on 127.0.0.1:7001 for protocols iiop, t3, ldap, snmp, http.>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <Server> <BEA-002613> <Channel "Default" is now listening on 192.168.1.3:7001 for protocols iiop, t3, ldap, snmp, http.>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <WebLogicServer> <BEA-000331> <Started WebLogic Admin Server "AdminServer" for domain "base_domain" running in Development Mode>
<02-Oct-2011 16:33:15 o'clock EEST> <Warning> <Server> <BEA-002611> <Hostname "KVLT0030.anet.com", maps to multiple IP addresses: 192.168.1.3, 0:0:0:0:0:0:0:1>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RUNNING>
<02-Oct-2011 16:33:15 o'clock EEST> <Notice> <WebLogicServer> <BEA-000360> <Server started in RUNNING mode>
```

The managed server can be started by running the `startManagedWebLogic` script, also located in the domain `bin` directory. Run the `startManagedWebLogic` script by specifying the name of the managed server instance and the URL of the administration server, for instance, `startManagedWebLogic.cmd ManagedServer1 http://localhost:7001`. In this case, it has been assumed that the managed server name is `ManagedServer1` and that the administration server runs locally and listens to port 7001.

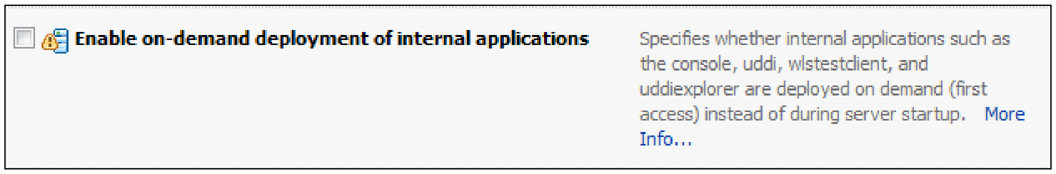
Note that each time you start the managed server instance, you will be asked to enter the domain administrator username and password. To avoid having to specify these credentials each time, create a file called `boot.properties` and add the following information to it:

```
username=<adminusername>
password=<adminpassword>
```

Replace `<adminusername>` and `<adminpassword>` with the administrator username and password respectively, and place the `boot.properties` file in the `servers/ManagedServer1/security` directory under the domain directory. Note that the `ManagedServer1` directory under the domain `servers` directory will not exist until you start the managed server at least once. You may also need to create the `security` directory yourself. Observe that the administrator username and password specified in the `boot.properties` file will be encrypted after starting the WebLogic Server.

To start the WebLogic administration console, browse the following address: `http://localhost:7001/console`. Use the administrator credentials specified during domain creation to log in.

To avoid having to redeploy the console application each time the domain is restarted, uncheck the **Enable on-demand deployment of internal applications** checkbox in the **Configuration** | **General** tab and click **Save**.



☐ Enable on-demand deployment of internal applications

Specifies whether internal applications such as the console, uddi, wlstestclient, and uddiexplorer are deployed on demand (first access) instead of during server startup. More Info...

For more information on the Fusion Middleware Configuration Wizard, consult the *Creating Domains Using the Configuration Wizard* documentation.

See also

- *Installation of JDeveloper on Linux*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

Deploying on the Standalone WebLogic Server

Once you have created and configured a WebLogic domain, ADF Fusion web applications can be deployed onto it. During development, deployment can take place from the JDeveloper IDE. The process involves the creation of an Application Server connection in the **Resource Palette** and the creation of deployment profiles for the ViewController project and the application workspace. The application can then be deployed onto the standalone WebLogic domain using the **Application** | **Deploy** menu.

In this recipe, we will go through the process of manually deploying a Fusion web application to a WebLogic domain using the JDeveloper IDE.

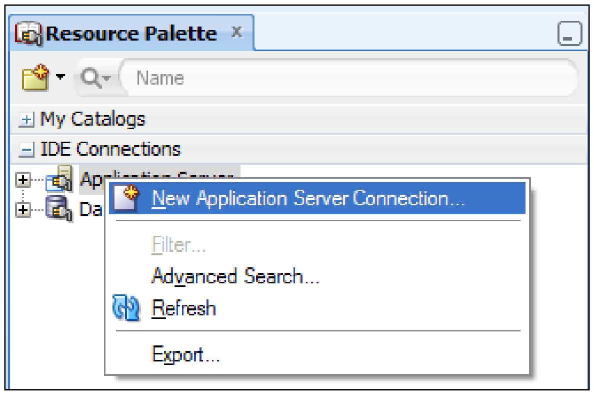
Getting ready

You need to complete the *Configuring and using the Standalone WebLogic Server* recipe in this chapter before delving in this recipe. Furthermore, a skeleton **Fusion Web Application (ADF)** workspace is required for this recipe. For this purpose, we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. The `MainApplication` workspace requires a connection to the `HR` schema.

How to do it...

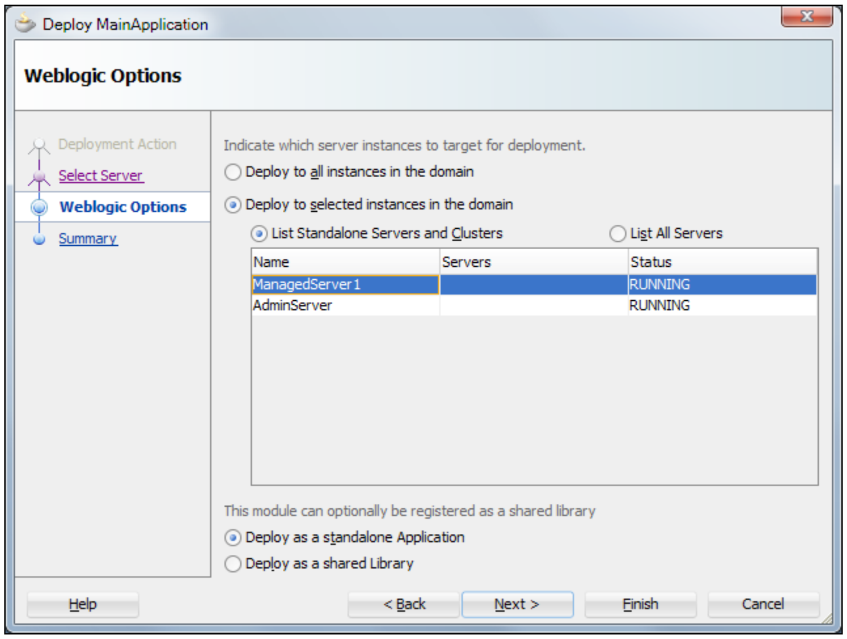
1. Open the `MainApplication` workspace. Double-click on the ViewController project in the **Application Navigator** to open the **Project Properties** dialog.
2. Click on **Deployment** and then click on the **New...** button to create a new deployment profile.
3. On the **Create Deployment Profile** dialog, select **WAR File** for the **Profile Type** and enter the name of the **Deployment Profile Name**.
4. On the **Edit WAR Deployment Profile Properties** dialog in the **General** section, enter the name and location of the **WAR File** and specify the **Java EE Web Context Root**. When done, click **OK** to dismiss all dialogs.

5. Select **Application Properties...** from the **Application** menu. On the **Application Properties** dialog, select **Deployment** and click on the **New...** button to create a new application deployment profile.
6. On the **Create Deployment Profile** dialog, select **EAR File** for the **Profile Type** and enter the **Deployment Profile Name**.
7. On the **Edit EAR Deployment Profile Properties** dialog, select the **General** section. Enter the name and location of the **EAR file** and specify the **Application name**.
8. Select **Application Assembly** and ensure that the **Java EE Modules** to be included in the EAR are selected. In this case, include both the Model and ViewController projects. When done, dismiss all dialogs by clicking **OK**.
9. Select **View | Resource Palette** to display the **Resource Palette** window. In the **Resource Palette**, expand the **IDE Connections** node. Right-click on the **Application Server** node and select **New Application Server Connection...**.



10. In the **Name and Type** page of the **Create Application Server Connection** wizard, enter the **Connection Name**. Select **WebLogic 10.3** for the **Connection Type** and click **Next**.
11. In the **Authentication** page, enter the WebLogic administrator credentials and click **Next**.
12. In the **Configuration** page, enter the WebLogic domain configuration and click **Next**.
13. In the **Test** page, click on the **Test Connection** button and ensure that **Status** is successful for all tests. Make sure that the WebLogic administration server instance is started before commencing with the tests. Click on the **Finish** button to complete the definition of the connection.

14. From the **Application** menu, select **Deploy** and then the deployment profile name.
15. In the **Deployment Action** page of the **Deploy** wizard, select **Deploy to Application Server** and click **Next**.
16. In the **Select Server** page, select the application server connection created earlier from the list of **Application Servers**. You can leave the **Overwrite modules of the same name** checkbox selected. Click **Next** to continue.
17. In the **Weblogic Options** page, click on the **Deploy to selected instances** radio button, and select the managed server instance to deploy onto, from the list of WebLogic Server instances. Click **Next** to continue.



Indicate which server instances to target for deployment.

Deploy to all instances in the domain

[✓ Deploy to selected instances in the domain]

[✓ List Standalone Servers and Clusters] List All Servers

Name	Servers	Status
ManagedServer1		RUNNING
AdminServer		RUNNING

This module can optionally be registered as a shared library

[✓ Deploy as a standalone Application]

Deploy as a shared Library

18. In the **Summary** page, verify the **Deployment Summary** and click **Finish** to proceed with the deployment. Observe in the **Deployment Log** window that the application is deployed successfully.

How it works...

In steps 1 through 4, we have defined a deployment profile for the ViewController project of the `Mainapplication` workspace. This project will be deployed as a Web Archive (WAR) file. In step 4, we have set the location and name of the WAR file that will be generated during deployment and specified the application context root. The context root is combined with the servlet mapping defined in `web.xml` to form the complete application URL. It is the base address for the application and all its associated resources.

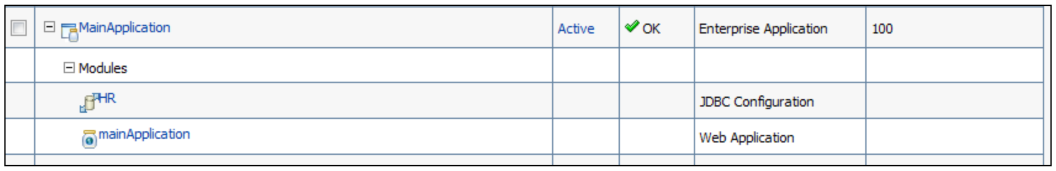
In steps 5 through 8, we have defined the application's Enterprise Archive (EAR) deployment profile. Observe how in step 7, we specify the name and location of the EAR file along with the application name. This is the name of the Java EE application as it will appear in the **Deployments** table in WebLogic. Moreover, note that in step 8, we specify the EE modules to be included in the EAR. In this case, we have included both the Model and ViewController projects. Failure to include both of these projects will result in a failed deployment.

In steps 9 through 13, we have created a new application server connection. We use the **Resource Palette** facility and the **Create Application Server Connection** wizard to go through the steps required to define a connection for a standalone WebLogic domain. Ensure that the WebLogic domain has been started before going through this process.

With the deployment profiles in place, and with the connection to the standalone WebLogic Server properly configured and successfully tested, we use the **Application** | **Deploy** menu to deploy the application. This is done in steps 14 through 18. The available application server connections were presented in step 16 based on the application server connections defined in JDeveloper. In step 17, we choose to deploy the application to the managed server instance.

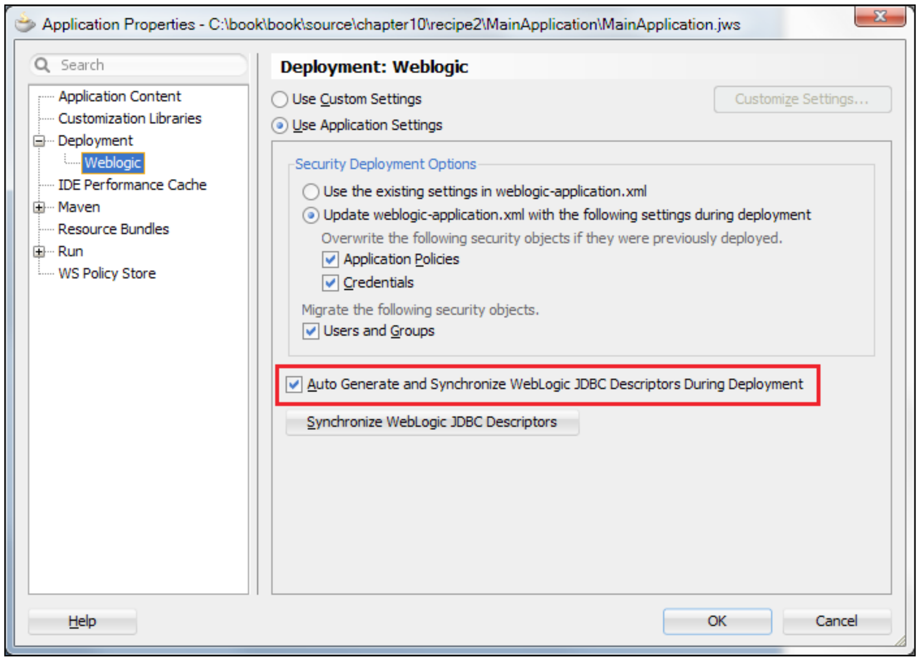
There's more...

You can check the application deployment status using the WebLogic administrator console. To do this, go to the **Summary of Deployments** available by selecting **Deployments** from the **Domain Structure** tree. The following screenshot shows our test application's deployment status:



☐	⊟ MainApplication	Active	✓ OK	Enterprise Application	100
	⊟ Modules				
	HR			JDBC Configuration	
	mainApplication			Web Application	

Observe that the **Health** status of the application is **OK**. Also, note that the `HR` data source is bundled in the deployed application. Whether the data source is bundled in the enterprise archive (EAR) produced by the deployment process, or not, is configured in the **Application Properties** dialog in JDeveloper. In the **Deployment** | **WebLogic** page, check or uncheck the **Auto Generate and Synchronize WebLogic JDBC Descriptors During Deployment** option to include or exclude the data source in the EAR file.


Application Properties - C:\book\book\source\chapter10\recipe2\MainApplication\MainApplication.jws

Deployment: Weblogic

( ) Use Custom Settings
(•) Use Application Settings

Security Deployment Options
( ) Use the existing settings in weblogic-application.xml
(•) Update weblogic-application.xml with the following settings during deployment
Overwrite the following security objects if they were previously deployed.
[✓] Application Policies
[✓] Credentials
Migrate the following security objects.
[✓] Users and Groups

[✓] Auto Generate and Synchronize WebLogic JDBC Descriptors During Deployment

Note that this recipe presents a method to deploy applications directly to a WebLogic domain using JDeveloper. This technique is typically used to deploy the application to a test environment during the development process, as it allows testing of features such as OPSS security configuration, LDAP configuration, and so on, that are not otherwise available when running the application directly in JDeveloper. An alternative technique involves deploying the application to an EAR file, which can be deployed in turn by a separate process using a variety of other tools. The EAR file can be produced using JDeveloper or with tools such as ojdeploy (see recipe *Using ojdeploy to automate the build process* in this chapter). In production environments, continuous integration tools such as Hudson (see recipe *Using Hudson as a continuous integration framework* in this chapter), can be combined with ojdeploy, ant, and WLST scripts to automatically deploy the application to its application server. For more information on deploying ADF applications, take a look at the section *Deploying the Application* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Configuring and using the Standalone WebLogic Server*, in this chapter

Using ojdeploy to automate the build process

ojdeploy is a command-line utility that can be used to automate the build and deployment process of ADF Fusion web applications. It is part of the JDeveloper installation package, and is installed alongside JDeveloper in the `jdeveloper/jdev/bin` directory (under the Middleware home directory). The utility can be run directly from the command line or it can be called from an ant script.

In this recipe, we demonstrate how to use ojdeploy to build an ADF Fusion web application comprised of three different workspaces. The final output of the build process is the application's Enterprise Archive file (EAR) file, which can be deployed to the Application Server using one of several possible techniques outlined in the *Deploying on the Standalone WebLogic Server* recipe in this chapter.

Getting ready

You need to have access to the `SharedComponents`, `HRComponents` and `MainApplication` workspaces. These workspaces were created in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. You also need to complete the recipe *Deploying on the Standalone WebLogic Server* in this chapter, to ensure that you have created the necessary deployment profiles for the `MainApplication` workspace. Finally, ensure that the `jdeveloper/jdev/bin` directory (under the Middleware home directory) is added to the `PATH` environment variable before running ojdeploy.

How to do it...

1. Using a text editor create the following ojdeploy build file `ojbuild.xml` as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ojdeploy-build basedir=".">
<!-- shared components workspace -->
<!-- This will build the SharedComponents.jar ADF Library JAR in
the ReUsableJARs directory -->
<deploy>
```

```
<parameter name="workspace"
  value="${application.root}
  \SharedComponents\SharedComponents.jws"/>
<parameter name="project" value="SharedViewController"/>
<parameter name="profile" value="SharedComponents"/>
</deploy>
<!-- HRComponents workspace -->
<!-- This will build the HRComponents.jar ADF Library JAR in the
  ReUsableJARs directory -->
<deploy>
<parameter name="workspace"
  value="${application.root}\HRComponents\HRComponents.jws"/>
<parameter name="project" value="HRComponentsViewController"/>
<parameter name="profile" value="HRComponents"/>
</deploy>
<!-- main application workspace -->
<!-- This will build both of the MainApplication.war and
  MainApplication.ear archives in the
  MainApplication\MainApplicationViewController\deploy and
  MainApplication\deploydirectories respectively -->
<deploy>
<parameter name="workspace"
  value="${application.root}
  \MainApplication\MainApplication.jws"/>
<parameter name="profile" value="MainApplication"/>
</deploy>
</ojdeploy-build>
```

2. Open a command shell and start the ojdeploy process by running the following command. Change `<application_root_directory>` to the appropriate directory under which your workspaces are located, as follows:

```
ojdeploy -buildfile ojbuild.xml -define
  application.root=<application_root_directory>
```

How it works...

In step 1, we have created an ojdeploy build file called `ojbuild.xml`. This is an XML file that comprises `ojdeploy-build` nodes along with embedded `deploy` nodes. Each `deploy` node defines a deployment process for the specific workspace, the project within the workspace, and the named deployment profile defined for the project. This information is specified by the `workspace`, `project`, and `profile` parameters respectively. If you do not specify a project name, then the workspace deployment profile is used, as in the case of the `MainApplication` workspace `deploy` configuration.

In step 2, we have initiated the deployment process by running ojdeploy with the `–buildfile` command-line argument. This parameter is used to specify the ojdeploy build file defined in step 1. Moreover, observe the usage of the `–define` argument to define a value for the macro `application.root`. This macro is used in the `ojbuild.xml` build file to reference the root application directory under which all application workspaces are located.

The result of running the ojdeploy deployment process for this recipe is the creation of the `SharedComponents.jar` and `HRComponents.jar` ADF Library JARs in the `ReUsableJARs` directory, the `MainApplication.war` archive in the `MainApplication\MainApplicationViewController\deploy` directory and the `MainApplication.ear` archive in the `MainApplication\deploy` directory.

There's more...

Note that the ojdeploy process performs a full business components validation. This involves the validation of all referenced business components throughout the ADF-BC projects involved in the build process. The validation process cross-references the component metadata XML files with the corresponding custom Java implementation classes.

For additional help on ojdeploy command-line arguments, built-in macros, and usage examples, run `ojdeploy –help` in the command line. A sample output is as listed:

```
Oracle JDeveloper Deploy 11.1.2.1.0.6081
Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights
reserved.

Usage:

ojdeploy -profile <name> -workspace <jws> [ -project <name> ] [
  <options> ]
ojdeploy -buildfile <ojbuild.xml> [ <options> ]
ojdeploy -buildfileschema
```

Arguments:

Name	Description
`profile`	The name of the profile to be deployed
`workspace`	Full path to the JDeveloper Workspace file(`.jws`)
`project`	Name of the JDeveloper Project within the .jws where the Profile can be found. If omitted, the Profile is assumed to be in the workspace
`buildfile`	Full path to a build file for batch deploy
`buildfileschema`	Print XML Schema for the build file

Options:

Name	Description
`basedir`	Interpret path for workspace relative to a base directory
`outputfile`	Substitute for the output file specified in the profile
`nocompile`	Skip compilation of Project or Workspace
`nodependents`	Do not deploy dependent profiles
`clean`	Clean output directories before compiling
`nodatasources`	Not include datasources from IDE
`forcerewrite`	Rewrite output file even if it is identical to existing file
`updatewebxmlejbrefs`	Update EJB references in `web.xml`
`define`	Define variables as comma separated name-value pairs
`statuslogfile`	Full path to an output file for status summary - no macros allowed
`failonwarning`	Stop deployment on warnings
`timeout`	Time in seconds allowed for each deployment task
`stdout`	Redirect `stdout` to file
`stderr`	Redirect `stderr` to file
`ojserver`	Run deployment using `ojserver`
`address`	Listen address for `ojserver`

Built-in macros:

Name	Description
`workspace.name`	name of the workspace (without the `.jws` extension)
`workspace.dir`	directory of the `workspace.jws` file
`project.name`	name of the project (without the `.jpr` extension)
`project.dir`	directory of the `project.jpr` file
`profile.name`	name of the `profile` being deployed
`deploy.dir`	default deploy directory for the `profile`
`base.dir`	current `ojdeploy` directory unless overridden by the -basedir parameter or by the "basedir" attribute in the build script

Note: `project.name` and `project.dir` are only available when project-level profile is being deployed.

Examples:

`Deploy a Project-level profile`

```
ojdeploy -profile webapp1 -workspace
  /usr/jdoe/Application1/Application1.jws -project Project1
ojdeploy -profile webapp1 -workspace Application1/Application1.jws -
  basedir /usr/jdoe -project Project1
```

`Deploy a Workspace-level profile`

```
ojdeploy -profile earprofile1 -workspace
  /usr/jdoe/Application1/Application1.jws
```

`Deploy all Profiles from all Projects of a Workspace`

```
ojdeploy -workspace /usr/jdoe/Application1/Application1.jws -project
  \* -profile \*
```

`Build in batch mode from a ojbuild file`

```
ojdeploy -buildfile /usr/jdoe/ojbuild.xml
```

`Build using ojbuild file, pass into, or override default variables in, the build file.`

```
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -define
  myhome=/usr/jdoe,mytmp=/tmp
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -basedir /usr/jdoe
```

`Build using ojbuild file, set or override parameters in the default section`

```
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -nocompile
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -outputfile
  '${workspace.dir}/${profile.name}.jar'
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -define mydir=/tmp -
  outputfile  '${mydir}/${workspace.name}-${profile.name}'
```

More examples:

```
ojdeploy -workspace
  Application1/Application1.jws,Application2/Application2.jws -
  basedir /home/jdoe -profile app*
ojdeploy -buildfile /usr/jdoe/ojbuild.xml -define
  outdir=/tmp,rel=11.1.1
-outputfile
  '${outdir}/built/${workspace.name}/${rel}/${profile.name}.jar'
ojdeploy -workspace Application1/Application1.jws -basedir /home/jdoe
  -nocompile
-outputfile '${base.dir}/${workspace.name}-${profile.name}'
ojdeploy -workspace /usr/jdoe/Application1.jws -project \* -profile
  \* -stdout /home/jdoe/stdout/${project.name}.log
```

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Deploying on the Standalone WebLogic Server*, in this chapter

Using Hudson as a continuous integration framework

Hudson is an open source continuous integration server that can be used to execute and monitor the execution of repeated jobs, such as building a software project. In the context of developing ADF Fusion web applications, Hudson can be used to build an ADF application directly from the version control sources and to deploy the built enterprise archive onto the application server. This is done automatically and continuously based on how Hudson is configured for each job.

In this recipe, we will go through the steps of defining a Hudson job that will build and deploy a sample ADF Fusion web application. We will check out the latest version of the application from the version control (Subversion) repository, build the application using ojdeploy, and finally deploy the application on the Standalone WebLogic Server using the weblogic.Deployer deployment tool.

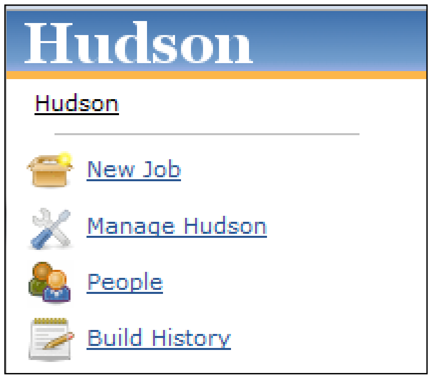
Getting ready

For the sample ADF Fusion web application, we will use the `SharedComponents`, `HRComponents`, and `MainApplication` workspaces that were created in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. It is assumed that the application components reside in a Subversion repository. In addition, we will utilize the ojdeploy build file developed in the *Using ojdeploy to automate the build process* recipe in this chapter.

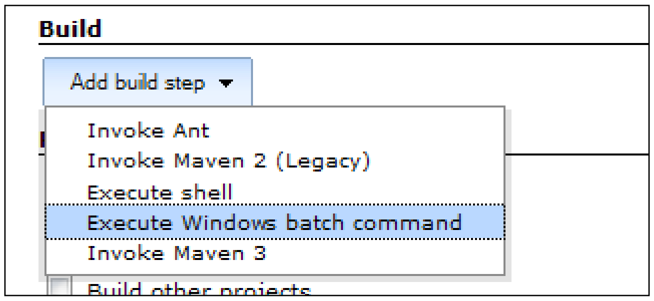
How to do it...

1. Download the latest version of Hudson. At the time of this writing, Hudson can be downloaded from `http://hudson-ci.org/`.
2. Install Hudson according to the documentation instructions. For the purpose of this recipe, we will run Hudson directly by executing `java -jar hudson-x.x.x.war` from the command line. `hudson-x.x.x.war` is the specific version of Hudson that was downloaded.
3. Access the Hudson dashboard using your browser. If you are running Hudson locally as stated previously, the URL would be `http://localhost:8080`.

4. Create a new job by clicking on the **New Job** link in the main page.

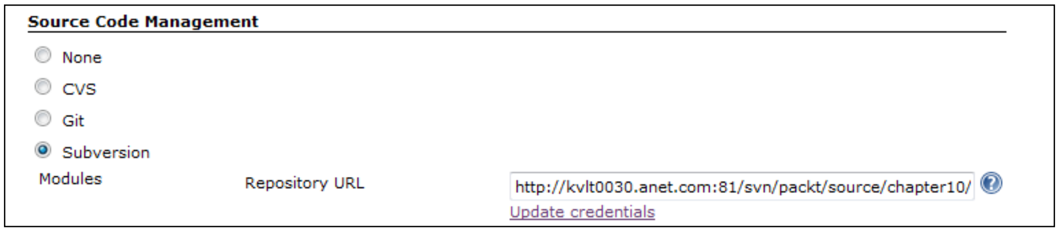


5. Provide the **Job Name** and select **Build a free-style software project**. Click **OK** to proceed.
6. In the job configuration page, select **Discard Old Builds**.
7. In the **Source Code Management** section, select **Subversion** and provide the **Repository URL**.
8. In the **Build Triggers** section, select **Build periodically** and enter `10 minutes` for the **Schedule** value. Use the following `cron` syntax **10 * * * ***.
9. In the **Build** section, click on the **Add build step** button. Select **Execute Windows batch command** (**Execute shell** if Hudson is running on Linux) and enter `build.cmd` in the **Command** field.



10. Click on the **Save** button to save the job definition.

11. With the job selected, click on the **Configure** link. In the **Source Code Management** section, click on the **Update credentials** link under the **Subversion Repository URL**.



Source Code Management

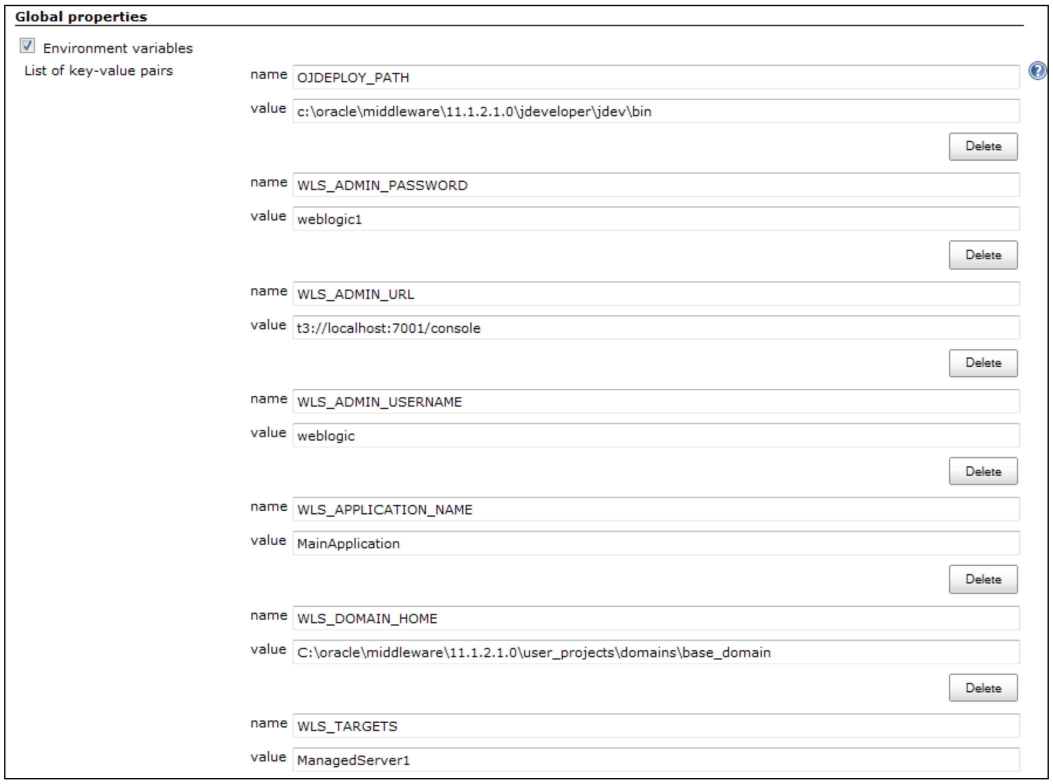
- None
- CVS
- Git
- [✓ Subversion]

Modules

Repository URL: http://kvlt0030.anet.com:81/svn/packt/source/chapter10/

Update credentials

12. In the **Subversion Authorization** screen, select **User name/password authentication** and enter the Subversion credentials.
13. From the main dashboard page, click on **Manage Hudson** | **Configure System**. In the **Global properties** section, click on **Environment variables** and then the **Add** button. Create new environment variables `OJDEPLOY_PATH`, `WLS_DOMAIN_HOME`, `WLS_ADMIN_URL`, `WLS_ADMIN_USERNAME`, `WLS_ADMIN_PASSWORD`, `WLS_APPLICATION_NAME`, and `WLS_TARGETS` and set their values appropriately.



Global properties

✓ Environment variables

List of key-value pairs

name	value
OJDEPLOY_PATH	c:\oracle\middleware\11.1.2.1.0\jdeveloper\jdev\bin
WLS_ADMIN_PASSWORD	weblogic1
WLS_ADMIN_URL	t3://localhost:7001/console
WLS_ADMIN_USERNAME	weblogic
WLS_APPLICATION_NAME	MainApplication
WLS_DOMAIN_HOME	C:\oracle\middleware\11.1.2.1.0\user_projects\domains\base_domain
WLS_TARGETS	ManagedServer1

14. Create the `build.cmd` script file at the application root folder with the the following code as its contents. Ensure that the `build.cmd` file is added to Subversion.

    ```
    REM Build application using ojdeploy
    "%OJDEPLOY_PATH%\ojdeploy" -buildfile ojbuild.xml -define
      application.root="%WORKSPACE%"  REM Deploy EAR
    call "%WLS_DOMAIN_HOME%\bin\setDomainEnv.cmd" %*
    java weblogic.Deployer -adminurl %WLS_ADMIN_URL% -username
      %WLS_ADMIN_USERNAME% -password %WLS_ADMIN_PASSWORD% -name
      %WLS_APPLICATION_NAME% -undeploy
    java weblogic.Deployer -adminurl %WLS_ADMIN_URL% -username
      %WLS_ADMIN_USERNAME% -password %WLS_ADMIN_PASSWORD% -name
      %WLS_APPLICATION_NAME% -deploy -upload
      "%WORKSPACE%\MainApplication\deploy\
      %WLS_APPLICATION_NAME%.ear" -
      targets "%WLS_TARGETS%"
    ```

How it works...

In steps 1 through 3, we have downloaded Hudson from the Hudson website and started it using the `java –jar hudson-x.x.x.war` command. This is not the recommended way to run Hudson in a production environment, but it will do for this recipe. It is recommended that the Hudson Web Archive (WAR) file is deployed onto one of the supported Web containers, as outlined in the Hudson installation documentation currently available in the Hudson wiki page `http://wiki.hudson-ci.org/display/HUDSON/Installing+Hudson`. Once started, Hudson can be accessed through a Web browser using the IP address or hostname of the server it is running on. We have executed it locally using the default startup configuration, so in this case, it is accessible through `http://localhost:8080`. The main Hudson page is called the Hudson Dashboard.

Steps 4 through 10 detail the definition of a Hudson job that will be used to build an ADF Fusion web application. The job uses ojdeploy to build the application Enterprise Archive (EAR) file and weblogic.Deployer to deploy the EAR file to the Standalone WebLogic Server. Both ojdeploy and weblogic.Deployer are accessed via an operating system script file. As we will be running Hudson on a Windows operating system, a cmd script file is used.

A Hudson job is defined by clicking on the **New Job** link in the Hudson Dashboard. This eventually takes you to the job definition page, a page with a rather long list of configuration parameters. However, the basic configuration parameters needed to get a simple job up and running are outlined in steps 6 through 10. First you need to specify the name of the Hudson job and select its type. Note that the job name also becomes part of the workspace directory, the directory used by Hudson to check out and stage the build, so be careful if you specify a job name with spaces in it. In this case, ensure that you access the workspace directory (when referenced) within double quotes, as in `"%WORKSPACE%"`. The Hudson workspace is accessible via the system-defined environment variable `WORKSPACE`.

In step 5, we have also chosen a `free-style software project` job type, which is a general job type.

In step 6, we have indicated what to do with previous builds. The option **Discard Old Builds** will allow you to define how many days to keep your builds and the maximum number of builds to keep.

In step 7, we specified the source control management system that we are using and entered the source control repository information. For this recipe, we are using Subversion as our source control management system. The credentials for accessing Subversion are specified at a later stage (see steps 11 and 12).

In step 8, we specified the job triggers. These are the possible ways that you can trigger the execution of the job. You can define multiple triggers. We have indicated that this job will run every 10 minutes. Observe that we have specified the time value using cron syntax. The cron syntax time value consists of 5 fields separated with white space: `MINUTE HOUR DOM MONTH DOW`, where `DOM` is the day of the month and `DOW` is the day of the week. For further details and examples on the cron time value syntax, see the Hudson online help.

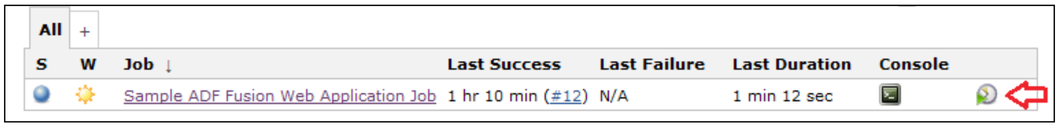
We have concluded the definition of the job by indicating in step 9 the execution of a Windows batch command. As indicated earlier, this is fine for the purposes of this recipe since we are running Hudson on a Windows operating system. You will adapt this step depending on your specific configuration. The Windows batch file that we will execute is called `build.cmd` and is implemented in step 14. We saved the job definition in step 10.

In steps 11 through 13, we provide additional configuration information. Note the definition of the environment variables in step 13. We will be using these environment variables in the `build.cmd` script.

The `build.cmd` script file is implemented in step 14. We have used ojdeploy and the `ojbuild.xml` build file that we created in recipe *Using ojdeploy to automate the build process* in this chapter. Note that the `application.root` parameter has been set to the job workspace directory. Hudson will check out the application from Subversion into this directory. The script file shows how to deploy the resulted EAR file to a Standalone WebLogic Server target. This is done using the weblogic.Deployer tool, a Java-based command-line tool that allows for the deployment (and undeployment) of applications to and from WebLogic. To ensure the proper configuration of the WebLogic domain environment, we have run the `setDomainEnv.cmd` script prior to the deployment process. Also, note that we have chosen to undeploy the application before its deployment. Finally, observe the usage of the environment variables defined in step 13.

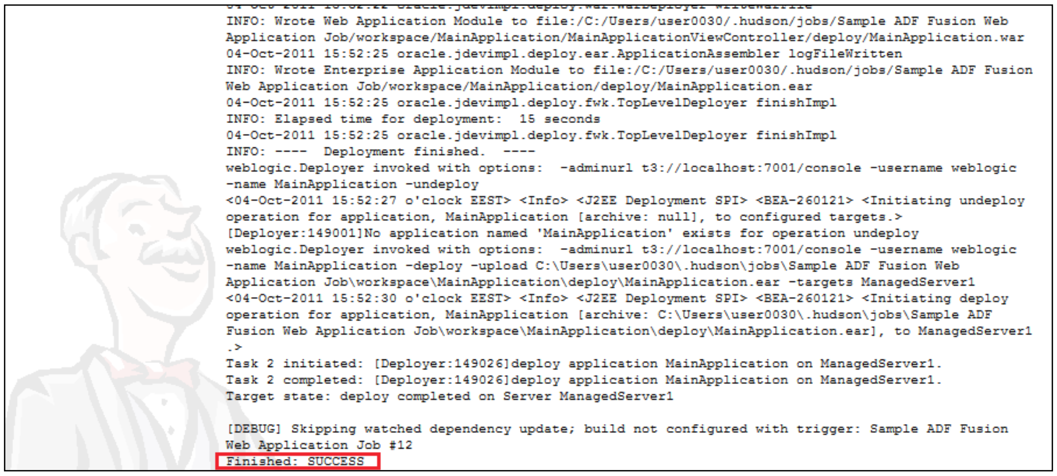
There's more...

To manually start the job, return to the Hudson Dashboard and click on the **Schedule a build** icon (the icon with the green arrow to the right).



S	W	Job ↓	Last Success	Last Failure	Last Duration	Console
●	☀	Sample ADF Fusion Web Application Job	1 hr 10 min (#12)	N/A	1 min 12 sec	▣

You can monitor the job status using the **Console Output** page. The status of the job is indicated at the bottom of the **Console Output**.



```
INFO: Wrote Web Application Module to file:/C:/Users/user0030/.hudson/jobs/Sample ADF Fusion Web
Application Job/workspace/MainApplication/MainApplicationViewController/deploy/MainApplication.war
04-Oct-2011 15:52:25 oracle.jdevimpl.deploy.ear.ApplicationAssembler logFileWritten
INFO: Wrote Enterprise Application Module to file:/C:/Users/user0030/.hudson/jobs/Sample ADF Fusion
Web Application Job/workspace/MainApplication/deploy/MainApplication.ear
04-Oct-2011 15:52:25 oracle.jdevimpl.deploy.fwk.TopLevelDeployer finishImpl
INFO: Elapsed time for deployment:  15 seconds
04-Oct-2011 15:52:25 oracle.jdevimpl.deploy.fwk.TopLevelDeployer finishImpl
INFO: ----  Deployment finished.  ----
weblogic.Deployer invoked with options:  -adminurl t3://localhost:7001/console -username weblogic
-name MainApplication -undeploy
<04-Oct-2011 15:52:27 o'clock EEST> <Info> <J2EE Deployment SPI> <BEA-260121> <Initiating undeploy
operation for application, MainApplication [archive: null], to configured targets.>
[Deployer:149001]No application named 'MainApplication' exists for operation undeploy
weblogic.Deployer invoked with options:  -adminurl t3://localhost:7001/console -username weblogic
-name MainApplication -deploy -upload C:\Users\user0030\.hudson\jobs\Sample ADF Fusion Web
Application Job\workspace\MainApplication\deploy\MainApplication.ear -targets ManagedServer1
<04-Oct-2011 15:52:30 o'clock EEST> <Info> <J2EE Deployment SPI> <BEA-260121> <Initiating deploy
operation for application, MainApplication [archive: C:\Users\user0030\.hudson\jobs\Sample ADF
Fusion Web Application Job\workspace\MainApplication\deploy\MainApplication.ear], to ManagedServer1
.>
Task 2 initiated: [Deployer:149026]deploy application MainApplication on ManagedServer1.
Task 2 completed: [Deployer:149026]deploy application MainApplication on ManagedServer1.
Target state: deploy completed on Server ManagedServer1

[DEBUG] Skipping watched dependency update; build not configured with trigger: Sample ADF Fusion
Web Application Job #12
Finished: SUCCESS
```

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Using ojdeploy to automate the build process*, in this chapter

11
Refactoring, Debugging, Profiling, and Testing

In this chapter, we will cover:

- Synchronizing business components with database changes
- Refactoring ADF components
- Configuring and using remote debugging
- Logging Groovy expressions
- Dynamically configuring logging in WebLogic Server
- Performing log analysis
- Using CPU profiler for an application running on a Standalone WebLogic Server
- Configuring and using JUnit for unit testing

Introduction

Refactoring support in JDeveloper allows you to modify the structure of an ADF Fusion web application without altering the overall behavior of the application. Each time you refactor an application component, JDeveloper transparently transforms the application structure by taking care of any references to the component. Refactoring at the ADF Fusion web application level allows renaming, modifying, and deleting application components. More options exist when refactoring Java code.

JDeveloper includes a comprehensive list of debugging features to allow you to debug ADF Fusion web applications deployed and running both locally on the Integrated WebLogic Server and remotely on the Standalone WebLogic Server. Similarly, profiling support in JDeveloper allows you to gather CPU and memory profiling statistics for applications deployed and running both locally and remotely.

You test your ADF Fusion web application by debugging it and profiling it in the JDeveloper IDE. When it comes to unit testing, JUnit can be integrated in JDeveloper through the installation of separate JDeveloper JUnit extensions. Once installed, these extensions make available a number of wizards in JDeveloper that make adding JUnit unit tests to ADF Fusion web applications quite easy.

Synchronizing business components with database changes

During the development process of an ADF Fusion web application, as the database schema evolves, there will be a need to synchronize the corresponding business components used in order to reflect these changes in the database schema. The process of synchronizing the business components is inherently supported in JDeveloper via the **Synchronize with Database** feature. Other capabilities also exist, such as making an attribute transient for a database table column that has been removed, and adding new entity attributes to view objects via the **Add Attribute from Entity** feature.

In this recipe, we will demonstrate a business components synchronization scenario that involves the addition, deletion, and modification of database table columns.

Getting ready

Before engaging in this recipe, you need to create a sample table in your database schema called `SYNCHRONIZATION`. We will use this table to demonstrate the business objects synchronization features. Use the following SQL code to accomplish this:

```
CREATE TABLE SYNCHRONIZATION (DELETED_COLUMN   VARCHAR2(30),
  MODIFIED_COLUMN  VARCHAR2(30));
```

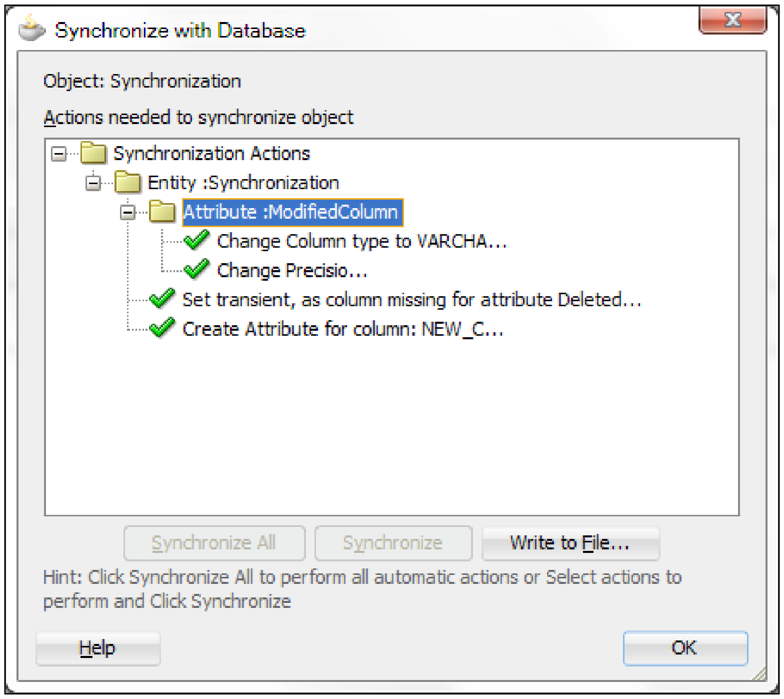
How to do it...

1. Create a **Fusion Web Application (ADF)** workspace. Using the **New Entity Object...** wizard, create an entity object for the `SYNCHRONIZATION` table. Also, generate a default view object called `SynchronizationView`.

2. Use the following SQL to modify the `SYNCHRONIZATION` table in the database:

```
ALTER TABLE SYNCHRONIZATION MODIFY(MODIFIED_COLUMN VARCHAR2(20
  BYTE));
ALTER TABLE SYNCHRONIZATION ADD (NEW_COLUMN  VARCHAR2(30));
ALTER TABLE SYNCHRONIZATION DROP
 COLUMN DELETED_COLUMN;
```

3. Right-click on the **Synchronization** entity object in the **Application Navigator** and select **Synchronize with Database...**.
4. In the **Synchronize with Database** dialog, click on the **Synchronize All** button and click **OK** on the verification dialog.



Synchronize with Database

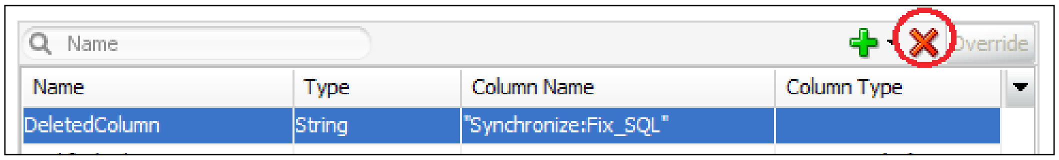
Object: Synchronization

Actions needed to synchronize object

```
Synchronization Actions
  Entity :Synchronization
    Attribute :ModifiedColumn
      Change Column type to VARCHA...
      Change Precisio...
    Set transient, as column missing for attribute Deleted...
    Create Attribute for column: NEW_C...
```

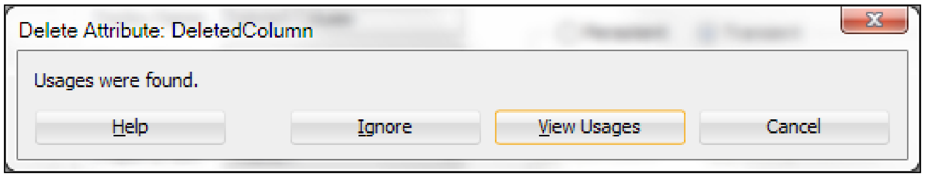
Hint: Click Synchronize All to perform all automatic actions or Select actions to perform and Click Synchronize

5. Open the `Synchronize` entity object in the **Overview** editor and click on the **Attributes** tab. Select the `DeletedColumn` attribute and click on the **Delete selected attribute(s)** button (the red X icon).



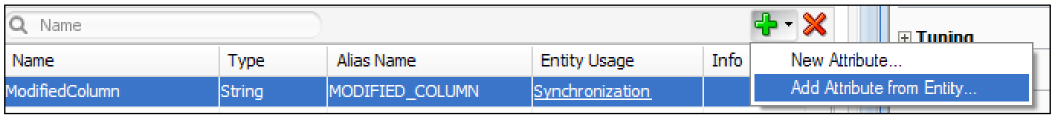
Name	Type	Column Name	Column Type
DeletedColumn	String	"Synchronize:Fix_SQL"	

6. In the **Delete Attribute** dialog, click on the **View Usages** button. Repeat step 5, this time clicking on the **Ignore** button.



Usages were found.

7. Double-click on the `SynchronizationView` view object in the **Application Navigator** and click on the **Attributes** tab in the **Overview** editor. Select the `DeletedColumn` attribute and click on the **Delete selected attribute(s)** button (the red X icon).
8. Select **Add Attribute from Entity...** by clicking on the green plus sign on top of the attributes list.



Name	Type	Alias Name	Entity Usage	Info
ModifiedColumn	String	MODIFIED_COLUMN	Synchronization	

9. In the **Attributes** dialog, select the `NewColumn` attribute in the **Available** tree and shuttle it to the **Selected** list.

How it works...

To demonstrate the business components database synchronization feature in JDeveloper, we have created an entity object based on the `SYNCHRONIZATION` table. Then we altered the table by adding, removing, and modifying table columns. The synchronization feature is accessible by right-clicking on the entity object in the **Application Navigator** and selecting **Synchronize with Database...**. Only entity objects are synchronized automatically. You will have to manually synchronize all other related business component objects, including any bindings that were made for the affected attributes and any references to these bindings and attributes in pages and in Java code (managed beans, business components custom implementation classes).

Observe in step 5 that the removal of a table column does not automatically remove the corresponding entity object attribute, but makes the attribute transient instead. As the attribute referring to a deleted column may be referenced by entity-based view objects, you will have to delete the corresponding view object attribute manually. We did this in step 7. Furthermore, observe that any new entity object attributes that were generated for the newly added table columns are not automatically added to the view object. You will have to do this manually. We do did this in steps 8 and 9.

There's more...

Note that adding new columns to a table does not affect the behaviour of the application, if the corresponding entity object is not synchronized. However, to use the new columns in your application, synchronization is required.

Refactoring ADF components

JDeveloper offers extensive support for refactoring ADF Fusion web application components, available through the **Refactor** main menu selections or via context menus for selected ADF components. The refactoring of ADF application components in most cases includes renaming, moving, and deleting these components. Refactoring of ADF components is supported throughout the Model-View-Controller architecture of the application including business components and their attributes, task flows, bindings, JSF files, and managed beans. Refactoring transparently takes care of updating any references to the refactored object, without affecting the overall functionality of the application.

In this recipe, we will demonstrate the refactoring facilities in JDeveloper by refactoring business components, business components attributes, task flows, JSF pages, associated page definition files and their bindings, and managed beans.

Getting ready

This recipe requires that you already have a **Fusion Web Application (ADF)** workspace that comprises business components, task flows, JSF pages, associated page definition files, and managed beans. For this purpose, we will use the `MainApplication` and `HRComponents` workspaces. These workspaces were developed in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects* respectively.

How to do it...

1. To refactor a business component, right-click on it in the **Application Navigator**, select **Refactor** from the context menu and a refactoring option (**Rename...** or **Move...**). To delete a business component, select **Delete** from the context menu. Alternatively, select **Rename...**, **Move...**, or **Delete** from the **Refactor** main menu.
2. To refactor a business component attribute, double-click on the business component in the **Application Navigator** to open the **Overview** editor and select the **Attributes** tab. Right-click on the attribute to refactor and select any of the **Rename...**, **Delete**, or **Change Type...** option.

3. To refactor a task flow, right-click on it in the **Application Navigator** and select any of the **Rename...**, **Move...**, **Delete** under the **Refactor** selection in the context menu.
4. To refactor a JSF page, right-click on the page in the **Application Navigator** and select any of the refactoring options available under the **Refactor** menu.
5. To refactor a page definition file, select any of the refactoring options under the **Refactor** main menu.
6. To refactor a page definition binding object, open the page data binding definition **Overview** editor and right-click on the binding object to refactor in the **Bindings** or **Executables** lists. Use the options available under the **Refactor** menu.
7. To refactor a managed bean, right-click on the managed bean in the **Application Navigator** and select any of the refactoring options available under the **Refactor** menu.
8. To refactor a plain file, select the file in the **Application Navigator** and use any of the available refactor options under the **Refactor** main menu.

How it works...

In steps 1 through 8, we have shown how to refactor almost any ADF Fusion web application component. In most cases, the refactoring options are available in both the main menu and context menu **Refactor** selections. In certain cases, such as when refactoring a page definition filename, the refactoring options are available only in the main menu **Refactor** selection. In other cases, as in the case of refactoring managed beans, additional options exist. Finally, observe what happens when you try to delete a component that is referenced by another component. A **Confirm Delete** dialog is displayed giving you the ability to discover the component's usages. The **Find Usages** feature is also separately available and can be used to determine the component's references prior to refactoring it.

There's more...

To refactor (rename) a deployment profile defined for a project, open the project configuration file (`.jpr`) in a text editor and locate the `oracle.jdeveloper.deploy.dt.DeploymentProfiles` node. Rename the profile identified by the `profileName` value. Similarly, you can rename a deployment profile defined for the workspace. Open the workspace configuration file (`.jws`) and locate the `oracle.jdeveloper.deploy.dt.DeploymentProfiles` node. Rename the profile identified by the `ProfileName` value. Alternatively, you can create a new deployment profile.

For information on how to manually refactor (move) the ADF business components project configuration file (`.jpx`), refer to the *Moving the ADF Business Components Project Configuration File (.jpx)* section in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

For information on how to refactor the data bindings registry file `DataBindings.cpx`, refer to section *Refactoring the DataBindings.cpx File* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

Finally, to rename a workspace project, you can use the **File** | **Rename** menu.

For a comprehensive reference to refactoring ADF components in JDeveloper, refer to the chapter *Refactoring a Fusion Web Application* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at `http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Configuring and using remote debugging

Remote debugging allows you to debug an ADF Fusion web application deployed and running remotely on a Standalone WebLogic Server directly from JDeveloper. Once both the remote WebLogic Server and the ADF project(s) in JDeveloper are configured to support it, a remote debugging session can be started in JDeveloper through the **Debug** menu selection. The session does not differ from a local debugging session for an application running on the Integrated WebLogic Server, but offers a number of advantages when compared to it. Some of these advantages are the ability to easily break inside any of the application's ADF Library JARs, the separation of the development process from the debugging of the application, freeing resources in JDeveloper, and using a Standalone WebLogic Server that closely matches the production environment configuration. When WebLogic is running on a separate machine, also consider the resources that are saved in the developer's machine.

In this recipe, we will see how to configure a managed WebLogic Server instance and JDeveloper to support remote debugging. We will also see how to initiate a remote debugging session in JDeveloper.

Getting ready

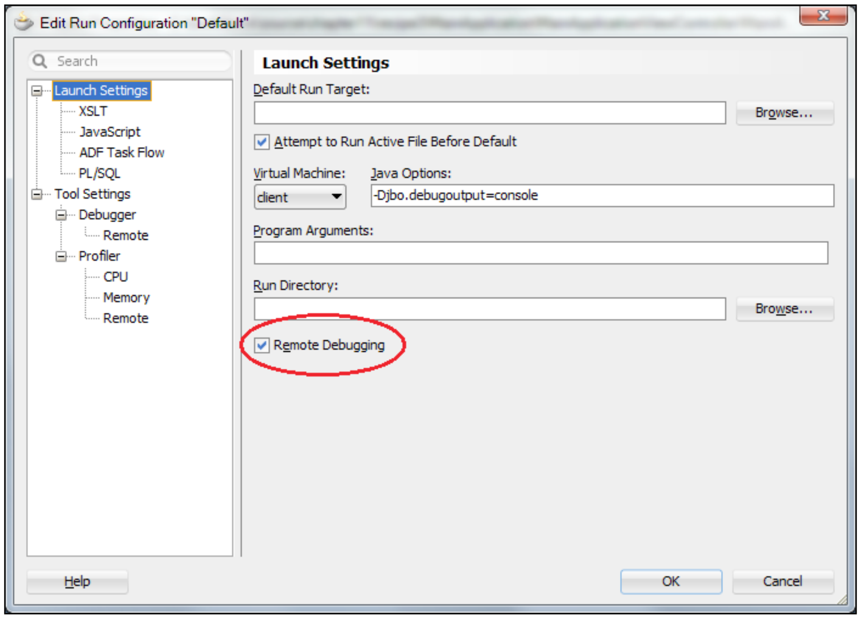
You will need a Standalone WebLogic Server, configured and started as explained in *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*. You will also need an ADF Fusion web application deployed to the Standalone WebLogic Server. For this, you can consult *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it...

1. Open the `startManagedWebLogic` script in a text editor located in the `bin` directory under the domain directory. Add the following definitions to it before calling the `startWebLogic` script:

   ```
   @REM Configuring and using remote debugging
   if "%SERVER_NAME%"=="ManagedServer1" (
     set debugFlag=true
     set DEBUG_PORT=4001
   )
   ```

2. While in the `startManagedWebLogic` script, remove the `nodebug` argument when calling the `startWebLogic` script.
3. Restart the WebLogic domain and log in to the WebLogic administrator console. Go to the **Summary of Servers** page by clicking **Environment** | **Servers** in the **Domain Structure** tree.
4. Click on the `ManagedServer1` managed server instance and then on the **Protocols** | **General** tabs. Click on the **Enable Tunneling** checkbox and then on the **Save** button. Log out from the WebLogic administrator console and restart the WebLogic domain.
5. In JDeveloper, double-click on the project that you want to configure for remote debugging to open the **Project Properties** dialog. In the **Project Properties** dialog, select **Run/Debug/Profile**.
6. Click on the **Edit...** button to edit the **Default** run configuration. Alternatively, you can create a new run configuration specifically for remote debugging. In the **Edit Run Configuration** dialog, Launch **Settings** page to ensure that the **Remote Debugging** checkbox is selected.



Launch Settings

Default Run Target:

☑ Attempt to Run Active File Before Default

Virtual Machine: client

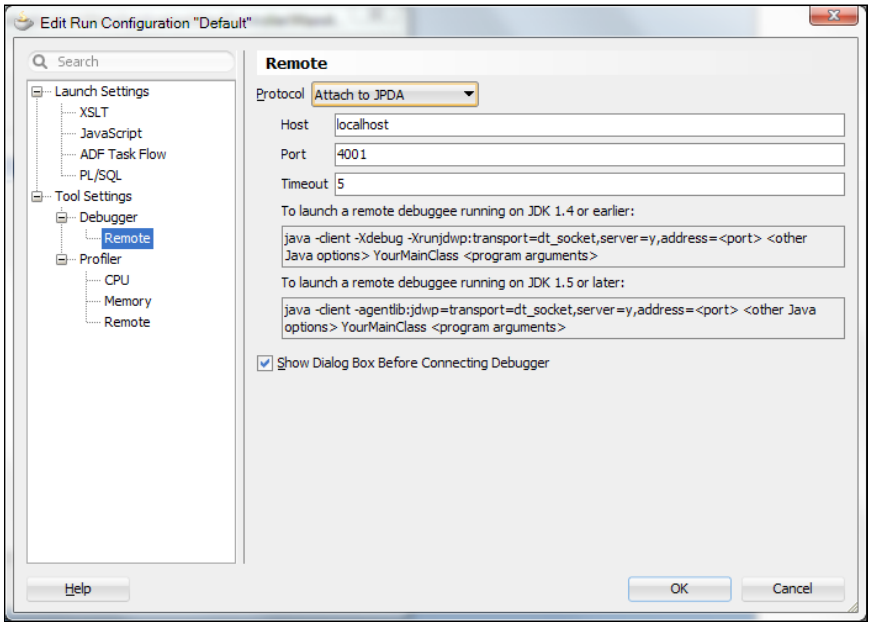
Java Options: -Djbo.debugoutput=console

Program Arguments:

Run Directory:

☑ Remote Debugging

7. While at the **Edit Run Configuration** dialog, select **Tool Settings** | **Debugger** | **Remote**. Ensure that the **Protocol** is set to **Attach to JPDA** and enter the information for the **Host**, **Port**, and **Timeout** fields. Make sure that you enter the debug port specified in step 1, that is, 4001 for this recipe.



Remote

Protocol: Attach to JPDA

Host: localhost

Port: 4001

Timeout: 5

To launch a remote debuggee running on JDK 1.4 or earlier:

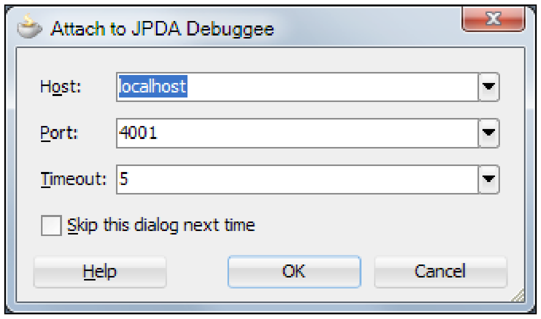
```
java -client -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=<port> <other
Java options> YourMainClass <program arguments>
```

To launch a remote debuggee running on JDK 1.5 or later:

```
java -client -agentlib:jdwp=transport=dt_socket,server=y,address=<port> <other Java
options> YourMainClass <program arguments>
```

☑ Show Dialog Box Before Connecting Debugger

8. Dismiss the **Edit Run Configuration** and **Project Settings** dialogs by clicking **OK** to save the configuration changes.
9. To start a remote debugging session, right-click on the specific project that was configured in the **Application Navigator** and select **Debug**. Verify the connection settings in the **Attach to JPDA Debuggee** dialog and click **OK**.


Attach to JPDA Debuggee

Host: localhost

Port: 4001

Timeout: 5

Skip this dialog next time

10. Observe in the **Debugging Log** that the connection to the remote WebLogic Server was successful. Set the necessary breakpoints in your code and start the application in the web browser.

How it works...

In steps 1 through 4, we configured the WebLogic managed server instance that we want to enable for remote debugging. This was done by editing the `startManagedWebLogic` script and setting the `debugFlag` environment variable to `true`. This is the script that we use to start a managed WebLogic server instance. By setting the `debugFlag` to `true`, the managed server will start to support remote debugging. This is actually done in the `setDomainEnv` script where the `JAVA_DEBUG` environment variable is set. Following are the debug configuration parameters specified in `setDomainEnv`:

```
set JAVA_DEBUG=-Xdebug -Xnoagent -
  Xrunjdwp:transport=dt_socket,address=%DEBUG_PORT%,server=y,
  suspend=n -Djava.compiler=NONE
```

The remote connection debug port is specified with the `DEBUG_PORT` environment variable, which was also set in step 1. The changes in step 1 were specified for the Windows operating system.

Note in step 1 how we check for the specific `ManagedServer1` managed server instance in order to set the remote debugging configuration parameters. Following this strategy, you will be able to enable remote debugging only for the specific server instances that you are interested. This will also allow you to specify different remote debugging ports for each managed server. Also, note in step 2 that we had to remove the `nodebug` argument when calling the `startWebLogic` script from within the `startManagedWebLogic` script.

In step 3, we restarted the WebLogic domain with the new configuration. Then, using the administration console, we enabled HTTP tunnelling for the `ManagedServer1` instance (step 4). This will enable WebLogic to simulate a T3 protocol connection using an HTTP connection and allow remote debugging to commence via a stateful connection between JDeveloper and WebLogic.

In steps 5 through 8, we configure the specific ADF project to allow for remote debugging. This configuration is done by configuring a project **Run Configuration**. A **Run Configuration** is available in the **Project Properties** dialog. Part of the configuration is to specify the host and remote connection port (4001) used in step 1.

To start a remote debugging session, ensure that the WebLogic domain is up and running. Right-click on the project configured for remote debugging in the **Application Navigator** and select **Debug**. Debugging is done as usual.

There's more...

To break inside an ADF Library JAR that is part of the application, you will need to enable remote debugging for the specific ADF Library JAR project as it is outlined in steps 5 through 8. In this case, if a remote debugging session is currently in progress, you need to first detach from it by clicking on the **Terminate** debug button and selecting **Detach** in the **Terminate Debuggee Process** dialog.

See also

- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*
- *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Logging Groovy expressions

Groovy is a Java-like scripting language that is integrated in the context of ADF business components, and is used in a declarative manner in expressions ranging from attribute and bind variable initializations to entity object validation rules and error messages. It runs in the same JVM as the application, is interpreted at runtime and is stored as metadata in the corresponding business component definitions. JDeveloper does not currently offer a debugging facility for Groovy expressions. In this recipe, we will implement a Groovy helper class that will allow us to log and debug Groovy expressions throughout the application.

Getting ready

We will add the Groovy logger class to the `SharedComponents` workspace. This workspace was created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `SharedComponents` workspace and create a new Java class called `GroovyLogger` for the business components project.
2. Open the `GroovyLogger` Java class in the Java editor and add the following code to it:

```
private static ADFLogger LOGGER =
  ADFLogger.createADFLogger(GroovyLogger.class);
public GroovyLogger() {
  super();
}
public static <T> T log(String groovyExpression, T data) {
  LOGGER.info("GroovyLogger ==> Expression: " +
    groovyExpression + ", Data: " + data);
  return data;
}
```

3. Redeploy the shared components workspace to an ADF Library JAR.

How it works...

We have added a `GroovyLogger` class to the `SharedComponents` workspace to allow for the logging and debugging of Groovy expression. The class implements a `log()` method, which accepts the Groovy expression to log, along with the expression data. It uses an `ADFLogger` to log the Groovy expression. The expression data is then returned to be used by the ADF framework.

Following is an example of how the `GroovyLogger` helper class can be used in your ADF business components Groovy expressions:

```
com.packt.jdeveloper.cookbook.shared.bc.logging.GroovyLogger
.log("adf.context.securityContext.userName",
  adf.context.securityContext.userName)
```

To debug your Groovy expressions, use the `GroovyLogger` class in your expressions as shown in the previous example and set a breakpoint anywhere in the `log()` method. Then inspect or watch the Groovy expressions using the available debug tools in JDeveloper.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*

Dynamically configuring logging in WebLogic Server

In the recipe *Setting up logging* in *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations,* we introduced the Oracle Diagnostics Logging (ODL) framework and how it could be utilized in an ADF Fusion web application through the `ADFLogger` class. In this recipe, we will demonstrate how to dynamically configure the ODL log level for a WebLogic Server instance at runtime. Specifically, we will configure the `oracle.jbo` business components logger for the `ManagedServer1` WebLogic Server instance to use the `NOTIFICATION` log level. `ManagedServer1` was created in *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*. Dynamic log configuration is done via the **WLST** WebLogic administration utility. This program allows for the execution of custom scripts written in jython (an implementation of Python written in Java) to configure ODL.

Getting ready

You will need a Standalone WebLogic Server domain configured and started. This was explained in *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it…

1. With the WebLogic Standalone Server started, run the WLST program located in the `oracle_common/common/bin` directory under the Middleware home. You do this by typing `wlst` in the shell command line.
2. Connect to the WebLogic administration server instance by issuing the following WLST command:

```
connect('weblogic','weblogic1','t3://localhost:7001')
```

3. Change the log level of the `oracle.jbo` logger to `NOTIFICATION` by issuing the following WLST command:

```
setLogLevel(target="ManagedServer1", logger="oracle.jbo",
  level="NOTIFICATION")
```

4. Verify that the `oracle.jbo` logger's log level was changed successfully by entering the following command:

   ```
   getLogLevel(target="ManagedServer1",logger='oracle.jbo')
   ```

5. Exit from WLST by typing `exit()`.

How it works...

In step 1, we started the WLST WebLogic script tool located in the Oracle home directory. This is the directory `oracle_common/common/bin` under the Middleware home. It is important that you run WLST in the specific directory because it supports custom commands to manage WebLogic logging.

In step 2, we connected to the WebLogic administration server instance using the `connect()` command. To do so, we have specified the administrator's authentication credentials and the administration server instance URL using the T3 protocol.

We changed the log level of the `oracle.jbo` logger to `NOTIFICATION` in step 3. The `oracle.jbo` logger is defined in the `logging.xml` logging configuration file located in the `config/fmwconfig/servers/ManagedServer1` directory under the domain directory, and it is utilized by the ADF Business Components framework. The log level was changed by issuing the command `setLogLevel()` and specifying the target server instance, the logger, and the new log level. The log level can be specified either as an ODL or as a Java log level. Valid Java levels are any of the following: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, or `FINEST`. On the other hand valid ODL levels include a message type followed by a colon and a message level. The valid ODL message types are: `INCIDENT_ERROR`, `ERROR`, `WARNING`, `NOTIFICATION`, and `TRACE`. The message level is represented by an integer value that qualifies the message type. Possible values are from 1 (highest severity) through 32 (lowest severity).

To verify that the log level has been successfully changed, we issued the command `getLogLevel()`(in step 4) specifying the WebLogic Server instance target and the logger. We exited from WLST by typing `exit()` in step 5.

There's more...

WLST includes additional commands for dynamically configuring logging in WebLogic, which allow you to configure log handlers and to list loggers and log handlers. For a comprehensive reference of the custom logging commands supported by WLST, refer to the *Logging Custom WLST Commands* chapter in the *WebLogic Scripting Tool Command Reference* document. This document is part of the WebLogic Server Documentation Library available online currently at the address `http://docs.oracle.com/cd/E14571_01/wls.htm`.

See also

- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Performing log analysis

A possibly lesser known feature of JDeveloper is its ability to perform ODL log analysis, known as the Oracle Diagnostic Log Analyzer. This feature allows you to open a diagnostics log file (or use the log file currently in the **Log** window in JDeveloper) and do a limited yet useful log analysis. For the Standalone WebLogic Server, diagnostics log files are produced by applications running on the specific WebLogic Server instance. The log files are produced and saved by WebLogic in a directory configured by the WebLogic administrator. This directory defaults to the `logs` directory under the `servers` directory for the specific server instance; that is, for a server instance called `ManagedServer1` they can be found in `servers/ManagedServer1/logs`. The `servers` directory is located under the specific domain directory.

In this recipe, we will see how to analyze a diagnostics log produced when running an ADF Fusion web application on a Standalone WebLogic Server. Alternatively, you can run the application in JDeveloper and analyze the log produced in the **Log** window.

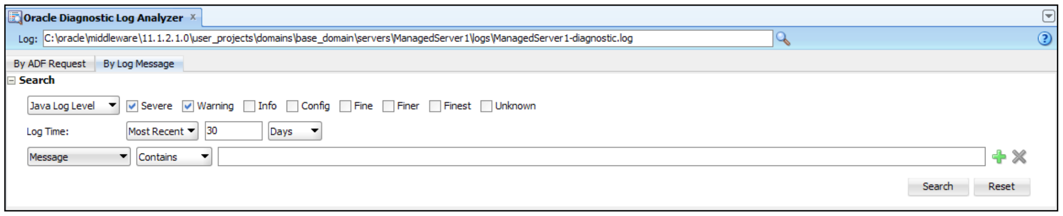
Getting ready

You will need a Standalone WebLogic Server domain configured and started. You will also need your Fusion web application deployed to the Standalone WebLogic Server. For more information on these topics, refer to *Configuring and using the Standalone WebLogic Server* and *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it...

1. Run the application deployed on the Standalone WebLogic Server, so that a diagnostics log file is generated. Alternatively, if you already have a diagnostics log file to analyze, you can ignore this step.
2. In JDeveloper, select **Tools** | **Oracle Diagnostic Log Analyzer** from the main menu.
3. Click on the **Browse Log Files** button (the search icon) to locate the diagnostics file and open it.

4. Click on the **By Log Message** tab and specify the search criteria in the **Search** section. Press the **Search** button to commence with the search.



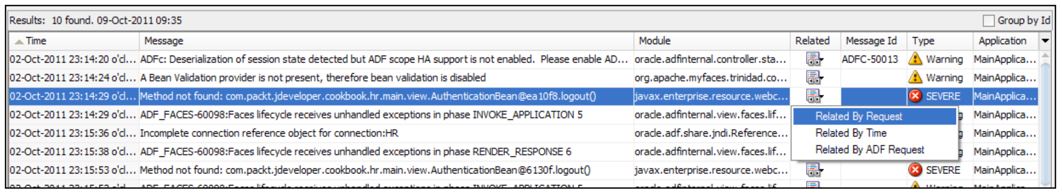
Log: C:\oracle\middleware\11.1.2.1.0\user_projects\domains\base_domain\servers\ManagedServer1\logs\ManagedServer1-diagnostic.log

Java Log Level [✓ Severe] [✓ Warning] Info Config Fine Finer Finest Unknown

Log Time: Most Recent 30 Days

Message Contains

5. In the **Results** table, click on a value inside the **Related** column for a log entry of interest and select **Related By Request** from the context menu.



Results: 10 found. 09-Oct-2011 09:35

Time	Message	Module	Related	Message Id	Type	Application
02-Oct-2011 23:14:20 o'cl...	ADFc: Deserialization of session state detected but ADF scope HA support is not enabled. Please enable AD...	oracle.adfinternal.controller.sta...		ADFC-50013	Warning	MainApplica...
02-Oct-2011 23:14:24 o'cl...	A Bean Validation provider is not present, therefore bean validation is disabled	org.apache.myfaces.trinidad.co...			Warning	MainApplica...
02-Oct-2011 23:14:29 o'cl...	Method not found: com.packt.jdeveloper.cookbook.hr.main.view.AuthenticationBean@ea10f8.logout()	javax.enterprise.resource.webc...			SEVERE	MainApplica...
02-Oct-2011 23:14:29 o'cl...	ADF_FACES-60098:Faces lifecycle receives unhandled exceptions in phase INVOKE_APPLICATION 5	oracle.adfinternal.view.faces.lif...			[illegible]	MainApplica...
02-Oct-2011 23:15:36 o'cl...	Incomplete connection reference object for connection:HR	oracle.adf.share.jndi.Reference...			[illegible]	MainApplica...
02-Oct-2011 23:15:38 o'cl...	ADF_FACES-60098:Faces lifecycle receives unhandled exceptions in phase RENDER_RESPONSE 6	oracle.adfinternal.view.faces.lif...			[illegible]	MainApplica...
02-Oct-2011 23:15:53 o'cl...	Method not found: com.packt.jdeveloper.cookbook.hr.main.view.AuthenticationBean@6130f.logout()	javax.enterprise.resource.webc...			SEVERE	MainApplica...

How it works...

Steps 1 through 5 give the details of the process of analyzing a diagnostics log file using the Oracle Diagnostics Log Analyzer feature in JDeveloper. The Oracle Diagnostics Analyzer is accessible via the **Tools** | **Oracle Diagnostic Log Analyzer** menu selection. Once started, you will need to load the specific diagnostics log file to analyze. We have done this in step 3. You can search the diagnostics log entries using either the **By ADF Request** or the **By Log Message** tab and specifying the search criteria. The **By ADF Request** tab will display only the log entries related to ADF requests made when a page is submitted. On the other hand the **By Log Message** tab will search all log entries in the log file by their log level. Moreover, the search criteria in both tabs allow you to search for diagnostic log entries based on their **Log Time** and based on the message content (**Message Id**, **User**, **Application**, **Module**, and so on).

The results of the search are displayed in the **Results** table. The results data are sortable by clicking on the column headers. To display all related log entries, click inside the **Related** column for a log entry of interest and select any of the choices available in the context menu. These choices are:

Related By	Results
Time	Filter diagnostic log entries to view all log entries leading up to the specific entry. You can refine the time before the entry using the dropdown.
Request	Filter diagnostic log entries to view all log entries for the same web request.
ADF Request	Switches to the **By ADF Request** tab to display the diagnostic log entries in a hierarchical arrangement to show their execution dependencies.

See also

- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*
- *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Using CPU profiler for an application running on a standalone WebLogic server

Profiling allows you to connect to a Standalone WebLogic Server instance and gather profiling statistics for your application. Profiling statistics can be subsequently used to identify and correct performance issues. JDeveloper supports both a CPU and a memory profiler. The CPU profiler gathers statistics related to CPU usage by the application. The memory profiler identifies how the application utilizes memory and can be used to diagnose memory leaks.

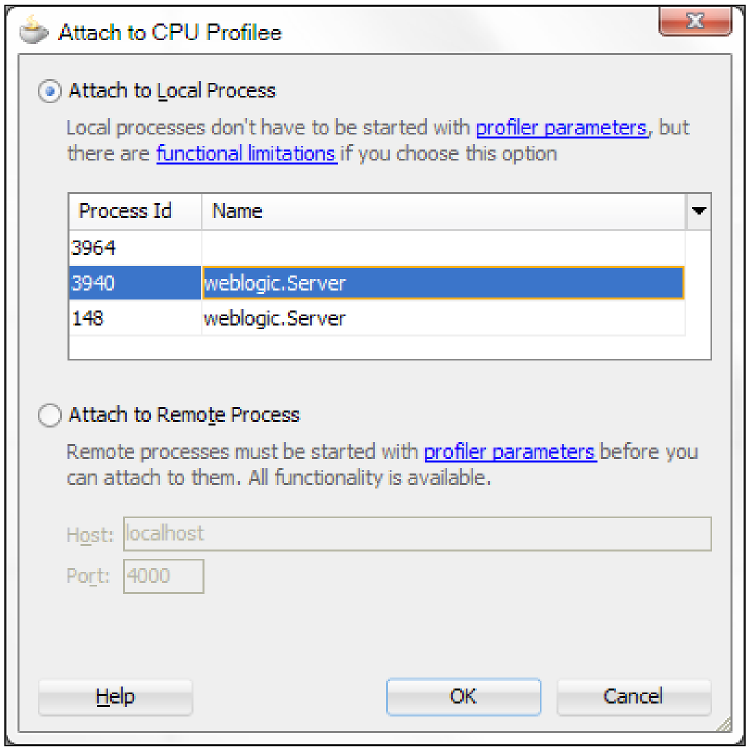
In this recipe, we will demonstrate how to use the CPU profiler to profile an ADF Fusion web application deployed to a Standalone WebLogic managed server instance running on the local machine.

Getting ready

You will need a Standalone WebLogic Server configured and started as explained in *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*. You will also need an ADF Fusion web application deployed to the Standalone WebLogic Server. For this, you can consult *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it...

1. In JDeveloper, double-click on the project that you want to profile in the **Application Navigator** to bring up its **Project Properties** dialog.
2. Select **Run/Debug/Profile** and click on the **Edit...** button to edit the **Default Run Configuration**.
3. In the **Tool Settings** | **Profiler** page, enter `com.packt.jdeveloper.cookbook.*` in the **Start Filter**. Click **OK** a couple of times to dismiss the **Project Settings** dialog, saving the changes.
4. Select **Run** | **Attach to** | **CPU Profilee** from the main menu. On the **Attach to CPU Profilee** dialog, select the **WebLogic profiler agent** process and click **OK**. The profiler agent process is started along with the Standalone WebLogic Server.



[✓ Attach to Local Process]

Local processes don't have to be started with profiler parameters, but there are functional limitations if you choose this option

Process Id	Name
3964	
3940	weblogic.Server
148	weblogic.Server

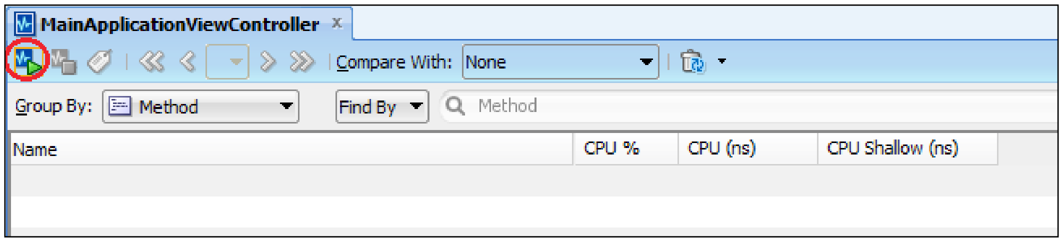
Attach to Remote Process

Remote processes must be started with profiler parameters before you can attach to them. All functionality is available.

Host: localhost

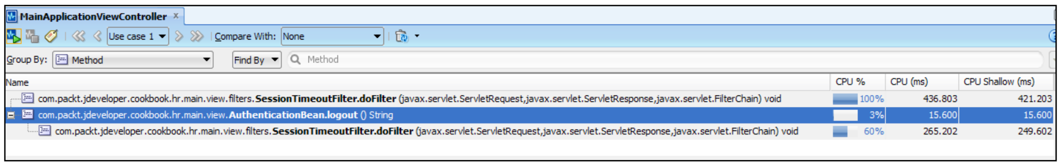
Port: 4000

5. Once attached to the profiler agent, the **Profiling <project_name>** tab is displayed, where `<project_name>` is the name of project you are profiling. Click on the **Begin Use Case** button (the first icon in the toolbar) to initiate a new profiling use case.



Name	CPU %	CPU (ns)	CPU Shallow (ns)

6. To generate profiler statistics, run the application in the web browser. To terminate the profiling session, click on the **Terminate Profiling** button (the red box icon) in the main toolbar.



Name	CPU %	CPU (ms)	CPU Shallow (ms)
com.packt.jdeveloper.cookbook.hr.main.view.filters.**SessionTimeoutFilter.doFilter** (javax.servlet.ServletRequest,javax.servlet.ServletResponse,javax.servlet.FilterChain) void	100%	436.803	421.203
com.packt.jdeveloper.cookbook.hr.main.view.**AuthenticationBean.logout** () String	3%	15.600	15.600
com.packt.jdeveloper.cookbook.hr.main.view.filters.**SessionTimeoutFilter.doFilter** (javax.servlet.ServletRequest,javax.servlet.ServletResponse,javax.servlet.FilterChain) void	60%	265.202	249.602

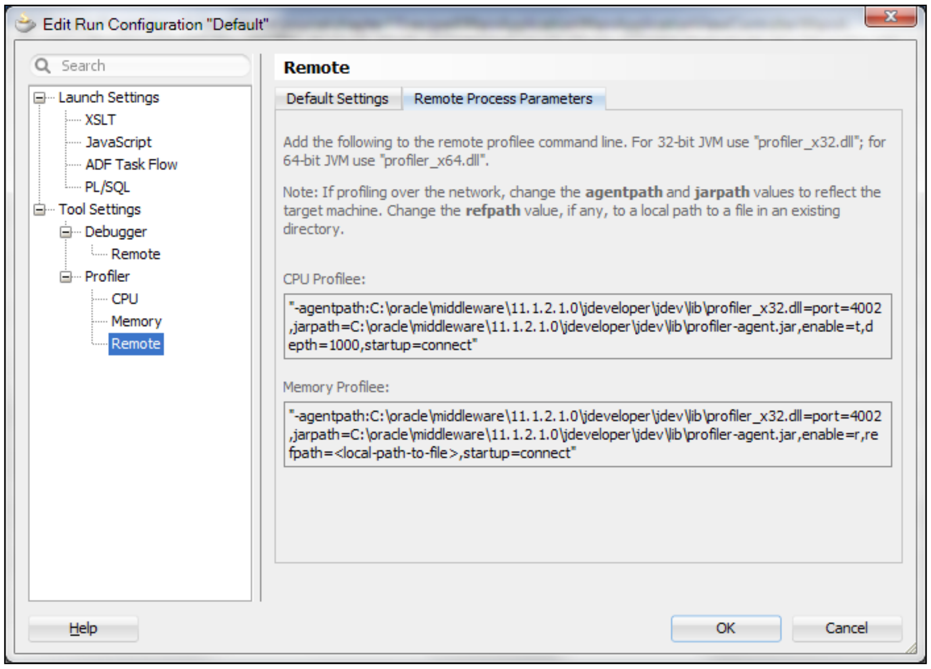
How it works...

Steps 1 through 3 demonstrate how to configure a project for profiling. Observe in step 3, how we have indicated the specific package filter based on which we would like to filter the profiler results. Profiler data will be collected only for those stack levels whose class name satisfies the **Stack Filter** entry. Multiple filters can be entered, separated with spaces. You can also click on the **Advanced** button in the **Profiler** page to select the classes you want to profile.

Steps 4 through 6 show how to start a profiling session and how to create a new use case to collect profiling statistics. Observe in step 4, our choice for connecting to the profiler agent. As we are running the Standalone WebLogic Server locally, we have chosen the profiler agent from the **Attach to Local Process** list.

There's more...

To profile an ADF Fusion web application running on a WebLogic Server on a remote machine, the profiler agent must also be started on the remote machine as part of the WebLogic start-up configuration. To determine the profiler agent start-up configuration parameters, select **Tool Settings** | **Profiler** | **Remote** in the **Edit Run Configuration** dialog and then the **Remote Process Parameters** tab. Adjust the remote process port as needed in the **Default Settings** tab.



Add the following to the remote profilee command line. For 32-bit JVM use "profiler_x32.dll"; for 64-bit JVM use "profiler_x64.dll".

Note: If profiling over the network, change the **agentpath** and **jarpath** values to reflect the target machine. Change the **refpath** value, if any, to a local path to a file in an existing directory.

CPU Profilee:

```
"-agentpath:C:\oracle\middleware\11.1.2.1.0\jdeveloper\jdev\lib\profiler_x32.dll=port=4002
,jarpath=C:\oracle\middleware\11.1.2.1.0\jdeveloper\jdev\lib\profiler-agent.jar,enable=t,d
epth=1000,startup=connect"
```

Memory Profilee:

```
"-agentpath:C:\oracle\middleware\11.1.2.1.0\jdeveloper\jdev\lib\profiler_x32.dll=port=4002
,jarpath=C:\oracle\middleware\11.1.2.1.0\jdeveloper\jdev\lib\profiler-agent.jar,enable=r,re
fpath=<local-path-to-file>,startup=connect"
```

See also

- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*
- *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Configuring and using JUnit for unit testing

JUnit is a unit testing framework for Java code. Unit testing refers to programmatically testing individual pieces of code and it is actually part of the software development and construction process. In JDeveloper, JUnit is supported via the **BC4J JUnit Integration** and **JUnit Integration** extensions available through the **Official Oracle Extensions and Updates** update center. The BC4J JUnit Integration extension makes available wizards for constructing JUnit unit test cases, suites, and fixtures specifically for business components projects. On the other hand, the JUnit Integration extension includes wizards to help you setup generic JUnit artifacts. Upon installation, these extensions make available the **Unit Tests** category under the **General** category in the **New Gallery** dialog.

A unit test class is a class that contains unit test methods. Unit test classes are grouped in a test suite that runs all of the test cases together when executed. A unit test fixture is a special class used to configure the unit tests.

In this recipe, we will implement a JUnit test suite that will test the functionality of an application module and the view objects that are part of its data model.

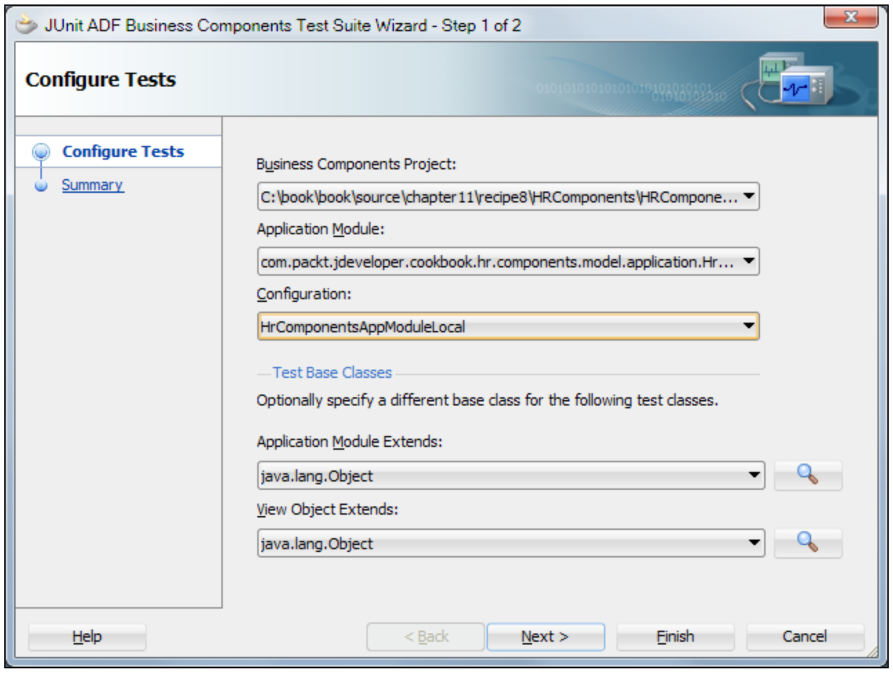
Getting ready

You will need access to the `HRComponents` workspace created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

How to do it...

1. In JDeveloper, select **Help** | **Check for Updates...** from the main menu. This will start the **Check for Updates** wizard.
2. In the **Source** page, select **Official Oracle Extensions and Updates** and click **Next**.
3. In the **Updates** page, select the **BC4J JUnit Integration** and **JUnit Integration** extensions and click **Next**.
4. Accept the JUnit license agreement and click **Next**. This will initiate the download of the JUnit extensions. Once the download is complete, in the **Summary** page, click on the **Finish** button. On the **Confirm Exit** dialog, click on the **Yes** button to restart JDeveloper.
5. Open the `HRComponents` workspace and create a project by selecting **Custom Project** from the **General** | **Projects** category in the **New Gallery** dialog.
6. In the **Name your project** page of the **Create Custom Project** wizard, enter `HRComponentsUnitTests` for the **Project Name** and click on the **Finish** button.
7. Right-click on the `HRComponentsUnitTests` project in the **Application Navigator** and select **New...**. From the **General** | **Unit Tests** category, select **ADF Business Components Test Suite** and click **OK**.

8. In the **Configure Tests** page of the **JUnit ADF Business Components Test Suite Wizard**, make sure that the appropriate **Business Components Project**, **Application Module**, and **Configuration** are selected. For this recipe, we will select `HRComponentsBC.jpr`, `HrComponentsAppModule`, and `HrComponentsAppModuleLocal` respectively. Then click **Next**.



Business Components Project: C:\book\book\source\chapter11\recipe8\HRComponents\HRCompone...

Application Module: com.packt.jdeveloper.cookbook.hr.components.model.application.Hr...

Configuration: HrComponentsAppModuleLocal

Test Base Classes

Optionally specify a different base class for the following test classes.

Application Module Extends: java.lang.Object

View Object Extends: java.lang.Object

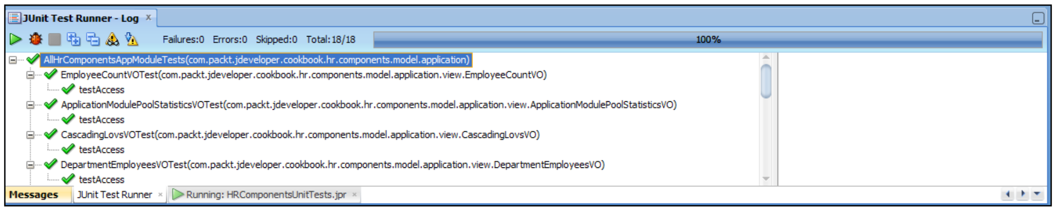
9. In the **Summary** page, review the JUnit classes that will be generated and click **Finish** to proceed.
10. Edit the `HrComponentsAppModuleAMTest` class and add the following code to the `setup()` method:

```
// get the application module from the JUnit test fixture
HrComponentsAppModuleAMFixture fixture =
  HrComponentsAppModuleAMFixture.getInstance();
_amImpl = (HrComponentsAppModule)fixture
  .getApplicationModule();
```

11. Add the following code to the `testExportEmployees()` method:

```
String employees = _amImpl.exportEmployees();
```

12. To run the unit tests, right-click on the `AllHrComponentsAppModuleTests.java` file in the **Application Navigator** and select **Run**. Observe the status of the unit tests in the **JUnit Test Runner Log** window.



```
Failures:0  Errors:0  Skipped:0  Total:18/18  100%
AllHrComponentsAppModuleTests(com.packt.jdeveloper.cookbook.hr.components.model.application)
  EmployeeCountVOTest(com.packt.jdeveloper.cookbook.hr.components.model.application.view.EmployeeCountVO)
    testAccess
  ApplicationModulePoolStatisticsVOTest(com.packt.jdeveloper.cookbook.hr.components.model.application.view.ApplicationModulePoolStatisticsVO)
    testAccess
  CascadingLovsVOTest(com.packt.jdeveloper.cookbook.hr.components.model.application.view.CascadingLovsVO)
    testAccess
  DepartmentEmployeesVOTest(com.packt.jdeveloper.cookbook.hr.components.model.application.view.DepartmentEmployeesVO)
    testAccess
```

How it works…

In steps 1 through 4, we downloaded the JUnit JDeveloper extensions using the **Check for Updates…** facility. As stated earlier, there are two separate extensions for JUnit one being specific to ADF business components projects.

In steps 5 and 6, we created a custom project to house the JUnit unit tests. Then (in steps 7 through 9), we created a JUnit business components test suite using the **ADF Business Components Test Suite Wizard**. We have indicated the `HRComponentsBC` business components project, and selected the `HrComponentsAppModule` application module and its `HrComponentsAppModuleLocal` configuration. Upon completion, the wizard creates the JUnit test suite, a test fixture class for the application module and unit test case classes for the application module and all view object instances in the application module data model. The unit tests that are included in the test suite are indicated by the `@Suite.SuiteClasses` annotation in the test suite, as shown in the following code snippet:

```
@Suite.SuiteClasses( { EmployeeCountVOTest.class,
  ApplicationModulePoolStatisticsVOTest.class,
  CascadingLovsVOTest.class,
  DepartmentEmployeesVOTest.class,
  EmployeesManagedVOTest.class,
  DepartmentsManagedVOTest.class, DepartmentsVOTest.class,
  EmployeesVOTest.class,
  HrComponentsAppModuleAMTest.class })
```

Furthermore, observe the code in the constructor of the `HrComponentsAppModuleAMFixture` fixture class. It uses the `oracle.jbo.client.Configuration` `createRootApplicationModule()` method to create the `HrComponentsAppModule` application module based on the configuration indicated in step 8. The `HrComponentsAppModule` application module is then available via the `getApplicationModule()` getter method.

The JUnit test cases created by the wizard are empty in most cases. In step 11, we have added test code to the `testExportEmployees()` application module test case to actually call the `exportEmployees()` `HrComponentsAppModule` application module method. To do this, we used the application module class variable `_amImpl`. This variable was initialized with a reference to the `HrComponentsAppModule` by calling the `HrComponentsAppModuleAMFixture` `getApplicationModule()` method in step 10.

Finally, we run the `AllHrComponentsAppModuleTests.java` file in the **Application Navigator** in step 11 to execute the JUnit test suite.

There's more...

Note the `@Test` annotation to indicate a test method in the test case class. You can add additional test methods to the unit test class by simply preceding them with this annotation. Also, observe the `@Before` and `@After` annotations on methods `setup()` and `teardown()` to indicate methods that are executing before and after the unit test case.

To include additional test cases to the test suite, implement the JUnit test case class and add it to the `@Suite.SuiteClasses` annotation in the test suite class.

JUnit unit test suites can be integrated with ant and be part of a continuous integration framework that runs your unit tests each time a new build of your application is being made. For a continuous integration example using Hudson, take a look at *Using Hudson as a continuous integration framework*, *Chapter 10*, *Deploying ADF Applications*.

See also

- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

12
Optimizing, Fine-tuning, and Monitoring

In this chapter, we will cover:

- Using Update Batching for entity objects
- Limiting the rows fetched by a view object
- Limiting large view object query result sets
- Limiting large view object query result sets by using required view criteria
- Using a Work Manager for processing of long running tasks
- Monitoring the application using JRockit Mission Control

Introduction

The ADF framework offers a number of optimization and tuning settings related to entity objects, view objects, and application modules. Many of these settings are accessible in JDeveloper in the **General** tab **Tuning** section of the corresponding **Overview** editor. Others are programmatic techniques that optimize the performance of the application, such as limiting the result set produced by a view object query, or providing query optimizer hints for the underlying view object query. Yet more are implemented by utilizing facilities offered by the application server, such as the use of work managers in the WebLogic Server.

When it comes to monitoring, profiling, and stress testing an ADF Fusion web application, in addition to the tools offered by JDeveloper (that is, the CPU and Memory Profiler) other external tools can be useful. Such tools include the JRockit Mission Control, Enterprise Manager Fusion Middleware Control, and Apache JMeter.

Using Update Batching for entity objects

When multiple entity objects of the same type are modified, the number of DML (`INSERT`, `UPDATE`, and `DELETE`) statements that are issued against the database corresponds to one for each entity object that was modified. This can be optimized by using entity object update batching optimization. When update batching is used, the DML statements are grouped per DML statement type (`INSERT`, `UPDATE`, and `DELETE`) and bulk-posted based on a configured threshold value. This threshold value indicates the number of entity objects of the same type that would have to be modified before update batching can be triggered.

In this recipe, we will see how to enable update batching for an entity object.

Getting ready

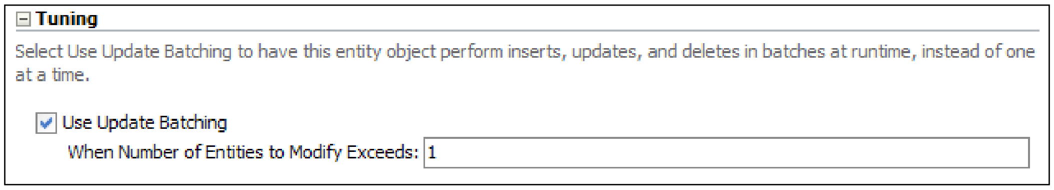
We will enable update batching for the `Department` entity object. This entity object is part of the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `HRComponents` workspace. In the **Application Navigator** expand the `HRComponentsBC` components project and locate the `Department` entity object. Double-click on it to open the **Overview** editor.
2. In the **General** tab, expand the **Tuning** section and check the **Use Update Batching** checkbox.

3. Enter `1` for the **When Number of Entities to Modify Exceeds**.



Tuning

Select Use Update Batching to have this entity object perform inserts, updates, and deletes in batches at runtime, instead of one at a time.

☑ Use Update Batching

When Number of Entities to Modify Exceeds: 1

4. Redeploy the `HRComponents` workspace to an ADF Library JAR.

How it works...

We have enabled update batching for the `Department` entity object by opening the entity object **Overview** editor and clicking on the **Use Update Batching** checkbox in the **Tuning** section of the **General** tab. We have also indicated the update batching threshold by entering a number in the **When Number of Entities to Modify Exceeds**. This threshold indicates the number of `Department` entity objects that would have to be modified in order for update batching to be triggered by the ADF framework. If the threshold is satisfied, then the framework will use a cursor to bulk-post the DML operations (one post per DML operation type). Otherwise, separate DML statements will be posted for each modified entity object.

There's more...

Using update batching will not affect the number of times an overridden `doDML()` will be called by the framework. This method will be called consistently for each modified entity object, regardless of whether the entity object uses update batching or not.

Furthermore, note that update batching cannot be used for entity objects that fall in any of the following categories (in these cases, update batching is disabled in JDeveloper).

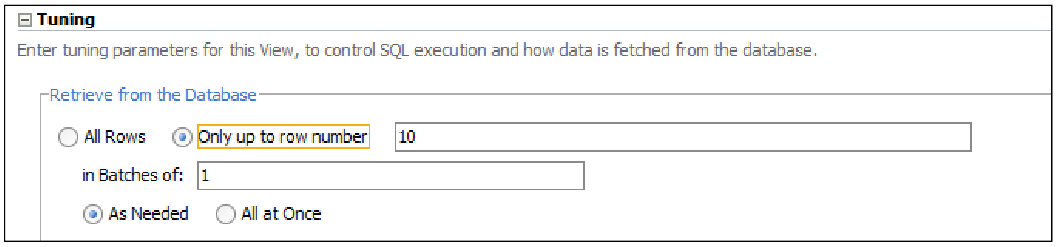
- An entity object that defines attributes that are refreshed on inserts and/or updates (**Refresh on Insert**, **Refresh on Update** properties).
- An entity object that defines `BLOB` attributes.
- An entity object that defines a ROWID-type attribute as a primary key. This attribute is also refreshed on inserts.

See also

- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Limiting the rows fetched by a view object

The ADF Business Components framework allows you to declaratively and/or programmatically set an upper limit for the number of rows that can fetched from the database layer by a view object. Declaratively, this can be accomplished through the view object **Tuning** section in the **General** page of the view object **Overview** editor. You can do this by selecting **Only up to row number** in the **Retrieve from the Database** section and providing a row count.


Tuning

Enter tuning parameters for this View, to control SQL execution and how data is fetched from the database.

Retrieve from the Database

All Rows [✓ Only up to row number] 10

in Batches of: 1

[✓ As Needed] All at Once

This can also be accomplished programmatically by calling a view object's `setMaxFetchSize()` method and specifying an upper row limit.

To globally set an upper limit for the number of rows that can be fetched by all view objects in an ADF Fusion web application, the global configuration setting `rowLimit` in the `adf-config.xml` configuration file can be used instead. Then, by overriding the framework `getRowLimit()` method, you can adjust this upper limit for individual view objects as needed. When an attempt is made to fetch rows beyond this upper limit, the framework will generate an `oracle.jbo.RowLimitExceededWarning` exception. This exception can then be caught by your custom `DCErrorHandlerImpl` implementation and presented as a Faces warning message box (see *Using a custom error handler to customize how exceptions are reported to the ViewController*, *Chapter 9*, *Handling Security, Session Timeouts, Exceptions and Errors*).

In this recipe, we will see how to globally limit the number of rows fetched by all view objects and how to override this global setting for specific view objects.

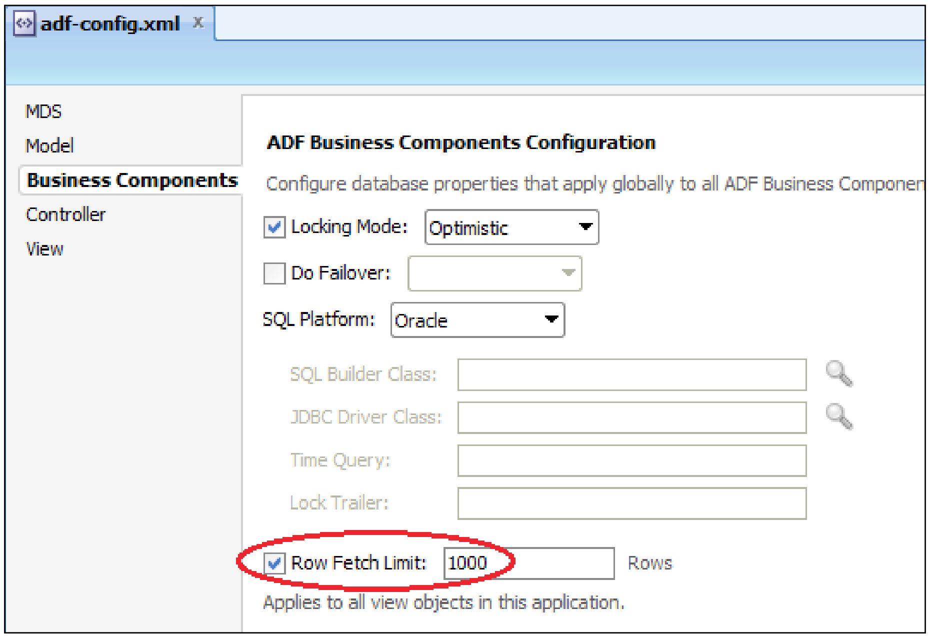
Getting ready

We will set an upper limit for the number of rows fetched by all view objects used in the `MainApplication` workspace. This workspace was created in *Breaking up the application in multiple workspaces Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*. We will also update the `Employees` view object to override this upper limit. This view object is part of the `HRComponents` workspace developed in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `MainApplication` workspace and locate the `adf-config.xml` file. The file is located in the **Application Resources** section of the **Application Navigator** under the **Descriptors** | **ADF META-INF** node. Double-click on the file to open it.
2. In the **Overview** page, click on the **Business Components** tab.
3. Click on the **Row Fetch Limit** checkbox and specify `1000` for the upper rows fetched limit.



ADF Business Components Configuration

Configure database properties that apply globally to all ADF Business Componen

✓ Locking Mode: Optimistic

Do Failover:

SQL Platform: Oracle

SQL Builder Class:

JDBC Driver Class:

Time Query:

Lock Trailer:

✓ Row Fetch Limit: 1000 Rows

Applies to all view objects in this application.

4. Now, open the `HRComponents` workspace and edit the `EmployeesImpl.java` view object custom implementation class.
5. Override the `getRowLimit()` method and replace the call to `super.getRowLimit()` with the following:

```
// return -1 to indicate no row fetch limit for the
// Employees View object
return -1;
```

6. Redeploy the `HRComponents` workspace to an ADF Library JAR.

How it works...

In steps 1 through 3, we have used the overview editor for the `adf-config.xml` ADF application configuration file to specify a global threshold value for the number of rows fetched by all view objects. For this recipe, we have indicated that up to 1000 rows can be fetched by all view objects throughout the application. Then, in steps 4 and 5, we have overridden the `getRowLimit()` method of the `Employees` view object to set a different fetch limit specifically for the `Employees` view object. In this case, by returning -1 we have indicated that there would be no fetch limit and that all rows should be fetched for this specific view object.

There's more...

Note that the maximum fetch limit of a view object is specified by -1, which indicates that all rows can be fetched from the database. This does not mean that all rows will be fetched by the view object at once, but that if you iterate over the view object result set, you will eventually fetch all of them. As stated earlier, when a fetched row limit is set, an attempt to iterate over the view object result set past this limit will produce an `oracle.jbo.RowLimitExceededWarning` exception.

See also

- *Breaking up the application in multiple workspaces*, Chapter 1, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, Chapter 2, *Dealing with Basics: Entity Objects*

Limiting large view object query result sets

In the recipe *Limiting the rows fetched by a view object* in this chapter, we have seen how to limit the number of rows that can be fetched from the database by a view object. While this technique limits the number of rows fetched from the database to the middle layer, it will not limit the view object query that runs in the database. In this case, a query that produces a result set in the thousands of records will still be executed, which would be detrimental to the application's performance. This recipe takes a different approach - actually limiting the view object query to a predefined row count defined by the specific view object using a custom property.

Getting ready

The recipe uses the `SharedComponents` and `HRComponents` workspaces. These workspaces were created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects* recipes respectively.

The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace. Locate and open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor. Add the following helper methods to it. Also ensure that you add a constant definition for `QUERY_LIMIT` to "`QueryLimit`".

```
private boolean hasQueryLimit() {
  // return true if the View object query has a limit
  return this.getProperty(QUERY_LIMIT) != null;
}
private long getQueryLimit() {
  long queryLimit = -1;
  // check for query limit
  if (hasQueryLimit()) {
    // retrieve the query limit
    queryLimit = new Long((String)this.getProperty(QUERY_LIMIT));
  }
  // return the query limit
  return queryLimit;
}
```

2. Override the `buildQuery(int, boolean)` method. Replace the call to `return super.buildQuery(i, b)` generated by JDeveloper with the following code:

```
// get the View object query from the framework
String qryString = super.buildQuery(i, b);
// check for query limit
if (hasQueryLimit()) {
  // limit the View object query based on the
  // query limit defined
  String qryStringLimited = "SELECT * FROM (" + qryString
    + " ) WHERE ROWNUM <= " + getQueryLimit();
    qryString = qryStringLimited;
}
return qryString;
```

3. Redeploy the `SharedComponents` workspace to an ADF Library JAR.
4. Open the `HRComponents` workspace. Locate and open the `Employees` view object in the **Overview** editor.
5. In the **Custom Properties** section of the **General** tab, add a custom property called `QueryLimit`. Set its **Value** to the number of rows that view object query will be limited to.
6. Redeploy the `HRComponents` workspace to an ADF Library JAR.

How it works...

In step 1, we have added two helper methods called `hasQueryLimit()` and `getQueryLimit()` which respectively determine the presence and retrieve the value of a view object custom property called `QueryLimit`. The `QueryLimit` custom property, when added to a view object, specifies a maximum number of rows threshold that the specific query is allowed to produce.

In step 2, we have overridden the view object `buildQuery()` method in order to check for the definition of the `QueryLimit` custom property by the view object and, if this is indeed the case, to construct a wrapper query that will limit the rows returned by the original view object query. The ADF Business Components framework calls the `buildQuery()` method when it needs to construct the view object query prior to its execution. The view object query is limited by adding a `WHERE` clause for a `ROWNUM` upto the value specified by the `QueryLimit` custom property. Note that these methods were added to the `ExtViewObjectImpl` framework extension class, part of the `SharedComponents` workspace, making this functionality generic and available to all view objects throughout the ADF application. We redeployed the `SharedComponents` workspace to ensure that this functionality is part of the ADF Library JAR.

In steps 4 through 6, we have updated the `Employees` view object, part of the `HRComponents` workspace, by adding to it the `QueryLimit` custom property and setting its value to the number of rows that the query is limited to.

There's more...

You can present a message informing the user that the query results for a particular search were limited, by adding this additional functionality to the application:

1. Add the following code to the `ExtViewObjectImpl` view object framework extension class:

```
private void setQueryLimitApplied(Boolean queryLimitApplied) {
  this.queryLimitApplied = queryLimitApplied;
}
private Boolean isQueryLimitApplied() {
  return this.queryLimitApplied;
}
```

```
public String queryLimitedResultsMessage() {
  String limitedResultsError = null;
  // check for query limit having been applied
  if (isQueryLimitApplied()) {
  // return a message indicating that the
  // query was limited
  limitedResultsError =
    BundleUtils.loadMessage("00008", new String[] {
    String.valueOf(this.getQueryLimit()) });
  }
  return limitedResultsError;
}
```

2. While editing the `ExtViewObjectImpl` framework extension class, override the `executeQueryForCollection()` method and add the following code after the `super.executeQueryForCollection()` line generated by JDeveloper:

```
// set the queryLimitApplied indicator appropriately
if (hasQueryLimit()
  && this.getEstimatedRowCount() > getQueryLimit()) {
  this.queryLimitApplied = true;
} else {
  this.queryLimitApplied = false;
}
```

3. Add the `queryLimitedResultsMessage()` method to the client interface for the specific view object that its query is limited (`Employees` in this example).
4. Create a method binding for the `queryLimitedResultsMessage` method for the specific JSF page where the query is used.
5. Add to a managed bean with the necessary code to programmatically invoke the method binding, as shown in the following sample code:

```
public String getQueryLimitedResultsMessage() {
  return (String)ADFUtils.findOperation(
    "queryLimitedResultsMessage").execute();
}
```

6. Use an `af:outputText` on the JSF to display the message, as shown in the following sample code:

```
<af:outputText id="ot1" value="#{SomeManagedBean.
  queryLimitedResultsMessage}"
  partialTriggers="qry1" visible="#{bindings.
  EmployeesIterator.currentRow != null}"/>
```

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Limiting large view object query result sets by using required view criteria

In the recipe *Limiting large view object query result sets* in this chapter, we presented a programmatic technique to limit the result set produced by a view object query. A simpler way to accomplish this in a declarative manner is to add named view criteria to the view object ensuring that some of the criteria items are required. This will force the user at runtime to enter values for those required criteria, thus limiting the size of the query result set.

In this recipe, we will add named view criteria to a view object and make the criteria items required.

Getting ready

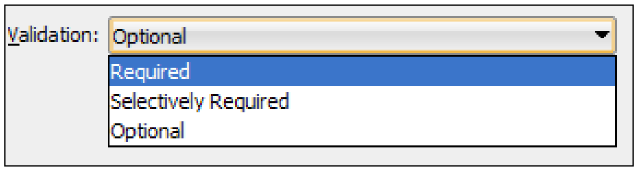
We will add named view criteria to the `Employees` view object. It is part of the `HRComponents` workspace, which was created in *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it…

1. Open the `HRComponents` workspace and locate the `Employees` view object.
2. Open the `Employees` **Overview** editor and go to the **Query** tab.
3. Click on the **Create new view criteria** button (the green plus sign icon) in the **View Criteria** section.

4. In the **Create View Criteria** dialog, add criteria items by clicking on the **Add Item** button. To ensure that a specific criteria item is required, select **Required** from the **Validation** drop-down list.

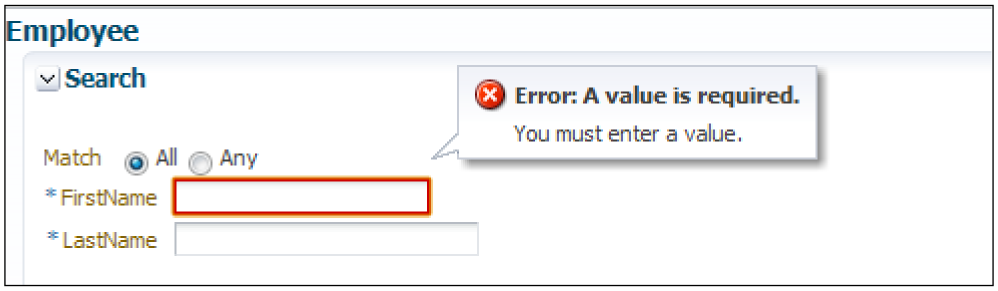


5. Redeploy the `HRComponents` workspace to an ADF Library JAR.

How it works...

Steps 1 through 4 show you how to add named view criteria to the `Employees` view object with required criteria items. View criteria are added to the view object by navigating to the **Query** tab of the view object **Overview** editor and clicking on the **Create new view criteria** button. You add criteria items to the view criteria by clicking on the **Add Item** button in the **Create View Criteria** dialog. To make a criteria item required for the query to be executed, ensure that you set the criterion **Validation** to **Required**.

At runtime, required criteria will appear with an asterisk (*) in front of them. If you attempt to execute the query without specifying values for any of the required criteria, a validation error message will be shown. To proceed with the query execution, you will need to provide values for all required criteria.


Error: A value is required.
You must enter a value.

There's more...

The **Selectively Required** option for the view criteria item **Validation** indicates that the specific criteria item will be required only as long as no other values have been supplied for any of the other criteria items. In this case, a validation exception will be raised indicating that the criterion is required. If a value has been supplied for any of the other criteria items, then specifying a value for the specific criterion is not required.

See also

- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*

Using a work manager for processing of long running tasks

Work managers allow for the concurrent execution of multiple threads within the WebLogic Server. They provide an alternative to the `java.lang.Thread` API (this API should not be utilized by Java EE applications) for running a work, that is an isolated piece of Java code, concurrently (or serially) as separate WebLogic-managed threads.

Work managers in the WebLogic Server fall in three categories: default, global and application-specific work managers. The default work manager is used for applications that do not specify a work manager. This may be sufficient for most applications. Global work managers are WebLogic Server domain-specific and are defined explicitly in WebLogic. Applications utilizing the same global work manager create their own instance of the work manager to handle the threads associated with each application. Application-specific work managers are defined for specific applications only, making them available for use by the specific applications only.

Programmatically, work managers are supported through the interfaces defined in the work manager API. The API is defined in the `commonj.work` package in the `weblogic.jar` library.

In this recipe, we will define a global work manager in WebLogic and implement a wrapper framework around the work manager API. Then we will demonstrate how to utilize the wrapper framework to run part of an ADF Fusion web application on the global work manager.

Getting ready

You will need access to the `SharedComponents`, `HRComponents` and `MainApplication` workspaces before delving into this recipe. These workspaces were created in *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations* and *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*.

The `HRComponents` workspace requires a database connection to the `HR` schema.

You will also need access to a configured standalone WebLogic server domain and your application deployed on it. For information on these topics, take a look at *Configuring and using the Standalone WebLogic Server* and *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it...

1. Open the `SharedComponents` workspace. Add the following `ExtWorkManager`, `ExtWork` and `ExtWorkListener` classes to the `SharedBC` business components project. When done, redeploy the workspace to an ADF Library JAR.

```
public class ExtWorkManager {
  private final static ADFLogger LOGGER =
    ADFLogger.createADFLogger(ExtWorkManager.class);
  private static final String DEFAULT_MANAGER_NAME =
    "MyWorkManager";
  private String managerName = DEFAULT_MANAGER_NAME;
  private WorkManager workManager;
  private WorkListener workListener;
  private List<ExtWork> works = new ArrayList<ExtWork>();
  List<WorkItem> workList = new ArrayList<WorkItem>();
  // run the Work Manager serially by default
  private long waitType = WorkManager.INDEFINITE;
  public ExtWorkManager() {
  }
  public ExtWorkManager(String managerName) {
    // check for valid name; used default name otherwise
    if (managerName == null || !"".equals(managerName)) {
      this.managerName = DEFAULT_MANAGER_NAME;
    }
  }
  public void addWork(ExtWork work) {
    works.add(work);
  }
  public void run() {
    LOGGER.info("WorkManager.run()");
    try {
      // get the Work Manager from the context
      InitialContext ctx = new InitialContext();
      workManager = (WorkManager)ctx.lookup("java:comp/env/"
        + managerName);
      // create a listener
      if (workListener == null) {
      workListener = new ExtWorkListener(this);
    }
    // schedule work items in a work list
    workList = new ArrayList<WorkItem>();
    for (ExtWork work : works) {
      WorkItem workItem = workManager.schedule(work,
        workListener);
```

```
          workList.add(workItem);
        }
        // run the Work Manager work list
        workManager.waitForAll(workList, waitType);
      } catch (Exception e) {
      LOGGER.severe(e);
      throw new ExtJboException(e);
      }
    }
    public List<ExtWork> getResult() {
      List<ExtWork> resultList = new ArrayList<ExtWork>();
      try {
      // iterate all work items and add their results
      // to the results list
      for (WorkItem workItem : workList) {
        resultList.add((ExtWork)workItem.getResult());
      }
    } catch (Exception e) {
      throw new ExtJboException(e);
    }
    // return the results list
    return resultList;
    }
    // see book's source code for complete listing
    }
    public abstract class **ExtWork** implements Work {
      private final static ADFLogger LOGGER =
        ADFLogger.createADFLogger(ExtWork.class);
      // parameters list
      protected List<Object> parameters =
        new ArrayList<Object>();
        public ExtWork(Object... parameters) {
          super();
          // add parameters to the parameter list
          for (Object parameter : parameters) {
            this.parameters.add(parameter);
        }
      }
      public abstract Object getResult();
      // see book's source code for complete listing
    }
    public class **ExtWorkListener** implements WorkListener {
      private final static ADFLogger LOGGER =
        ADFLogger.createADFLogger(ExtWorkListener.class);
```

```
  private ExtWorkManager manager;
  public ExtWorkListener(ExtWorkManager manager) {
    super();
    this.manager = manager;
  }
  public void workAccepted(WorkEvent workEvent) {
    LOGGER.info("Work accepted for work manager '" +
      manager.getManagerName() + "' at " + getTime());
  }
  private String getTime() {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf =
      new SimpleDateFormat("HH:mm:ss");
    return sdf.format(cal.getTime());
  }
  // see book's source code for complete listing
}
```

2. Open the `HRComponents` workspace and add the following `ExportEmployeesWork` class to it:

```
public class ExportEmployeesWork extends ExtWork {
  private final static ADFLogger LOGGER =
    ADFLogger.createADFLogger(ExportEmployeesWork.class);
  private StringBuilder employeeStringBuilder;
  public ExportEmployeesWork() {
    super();
  }
  public ExportEmployeesWork(Object... parameters) {
    super(parameters);
  }
  @Override
  public Object getResult() {
    // return the employees CSV string buffer
    return employeeStringBuilder;
  }
  @Override
  public void run() {
    LOGGER.info("ExportEmployeesWork.run()");
    // the Employees rowset iterator was passed as a
    // parameter when we created this work
    RowSetIterator iterator =  (RowSetIterator)parameters.get(0);
    // get additional parameters as needed
    // Object param1 = parameters.get(1);
    // build the employees CSV string buffer
    employeeStringBuilder = new StringBuilder();
```

```
        iterator.reset();
        while (iterator.hasNext()) {
          EmployeesRowImpl employee =
            (EmployeesRowImpl)iterator.next();
          employeeStringBuilder.append(
          employee.getLastName() + " "
            + employee.getFirstName());
          if (iterator.hasNext()) {
            employeeStringBuilder.append(",");
          }
        }
        // done with the rowset iterator
        iterator.closeRowSetIterator();
      }
    }
```

3. Add the following `exportEmployeesOnWorkManager()` method to the `HrComponentsAppModuleImpl` custom implementation class.

```
public String exportEmployeesOnWorkManager() {
  // create a Work Manager
  ExtWorkManager mngr = new ExtWorkManager("MyWorkManager");
  // add the export employees work to the Work Manager
  mngr.addWork(new ExportEmployeesWork(
    getEmployees().createRowSetIterator(null)));
  // run the Work Manager
  mngr.run();
  // get the result from the Work Manager
  List<ExtWork> works = mngr.getResult();
  StringBuilder employeeStringBuilder = new StringBuilder();
  for (ExtWork work : works) {
    ExportEmployeesWork exportWork = (ExportEmployeesWork)work;
    employeeStringBuilder.append(exportWork.getResult());
  }
  // return the employees CSV string buffer
  return employeeStringBuilder.toString();
}
```

4. Ensure that the `exportEmployeesOnWorkManager()` method is added to the `HrComponentsAppModule` application module client interface. Then, redeploy the `HRComponents` workspace to an ADF Library JAR.

5. Open the main application workspace. Create a new JSPX page called `exportEmployeesUsingWorkManager.jspx` and add the following code to it:

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
xmlns:f="http://java.sun.com/jsf/core"
  xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document title="exportEmployees
      UsingWorkManager.jspx" id="d1">
    <af:messages id="m1"/>
      <af:form id="f1">
        <af:panelStretchLayout id="psl1">
          <f:facet name="top"/>
          <f:facet name="center">
            <af:toolbar id="t1">
              <af:commandButton text="Export Employees" id="cb1">
                <af:fileDownloadActionListener filename=
                  "employees.csv"method="#{ExportEmployees
                  UsingWorkManagerBean.exportEmployees}"/>
              </af:commandButton>
            </af:toolbar>
          </f:facet>
          <f:facet name="bottom"/>
        </af:panelStretchLayout>
      </af:form>
    </af:document>
  </f:view>
</jsp:root>
```

6. Create a page definition file for the `exportEmployeesUsingWorkManager.jspx` page and add a method action binding for the `exportEmployeesOnWorkManager()` method. It is available under the `HrComponentsAppModuleDataControl` data control.

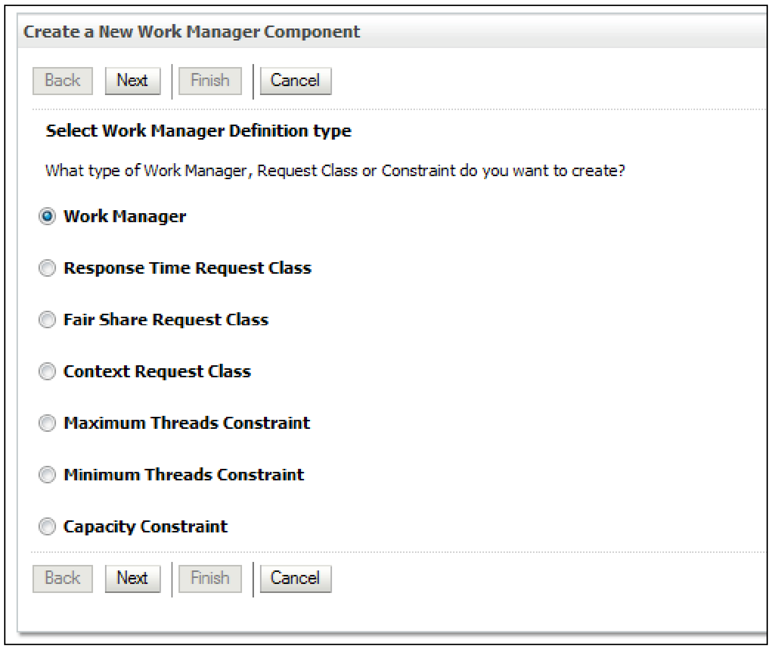
7. Create a managed bean called `ExportEmployeesUsingWorkManagerBean` and add the following `exportEmployees()` method to it:

```
public void exportEmployees(FacesContext facesContext,
  OutputStream outputStream) {
  // get the employees CSV data
  String employeesCSV = (String)ADFUtils.findOperation(
    "exportEmployeesOnWorkManager").execute();
  try {
    // write the data to the output stream
    OutputStreamWriter writer = new
      OutputStreamWriter(outputStream, "UTF-8");
    writer.write(employeesCSV);
    writer.close();
    outputStream.close();
  } catch (IOException e) {
    // log exception
  }
}
```

8. Open the `web.xml` deployment descriptor in the **Source** editor and add the following resource reference to it:

```
<resource-ref>
  <res-ref-name>MyWorkManager</res-ref-name>
  <res-type>commonj.work.WorkManager</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

9. Ensure that the standalone WebLogic server domain is started, then log in into the administration console using the following URL: `http://serverHost:serverPort/console`, where `serverHost` is the hostname or IP of the WebLogic Server machine and `serverPort` is the administration server's port.
10. Select **Environment** | **Work Managers** from the **Domain Structure** tree.
11. In the **Summary of Work Managers** page, click on the **New** button under the **Global Work Managers, Request Classes and Constraints** table.
12. In the **Select Work Manager Definition type** page, select **Work Manager** and click **Next**.

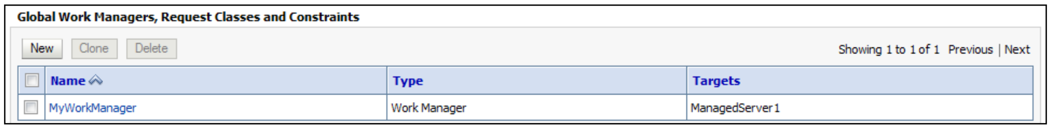


Select Work Manager Definition type

What type of Work Manager, Request Class or Constraint do you want to create?

- [✓ Work Manager]
- Response Time Request Class
- Fair Share Request Class
- Context Request Class
- Maximum Threads Constraint
- Minimum Threads Constraint
- Capacity Constraint

13. In the **Work Manager Properties** page, enter `MyWorkManager` for the work manager **Name** and click **Next**.
14. In the **Select deployment targets** page, select your managed server instance from the list of **Available targets** and click **Finish**. The work manager should now be visible in the **Global Work Managers, Request Classes and Constraints** table in the **Summary of Work Managers** page.


Global Work Managers, Request Classes and Constraints

Name	Type	Targets
MyWorkManager	Work Manager	ManagedServer1

15. Click on `MyWorkManager` in the **Global Work Managers, Request Classes and Constraints** table in the **Summary of Work Managers** page. In the **Settings for MyWorkManager** page, select **Ignore Stuck Threads** and click on the **Save** button.
16. Restart the standalone WebLogic server domain and deploy to it the main application.

How it works...

To ease the task of dealing with work managers, we have introduced the following three classes (in step 1):

- `ExtWorkManager`: A wrapper around the functionality provided by the `commonj.work` work manager API. The following methods implemented by this class make it easy to get going with using work managers in our application:
    - `ExtWorkManager(String managerName)`: Constructs a work manager identified by its name
    - `addWork(ExtWork work)`: Adds `ExtWork` works to the work manager
    - `run()`: Executes the work manager
    - `getResult()`: Returns the work manager result(s)
- `ExtWork`: An abstract class built on top of the `commonj.work.Work` interface. It accepts generic parameters during construction, which it stores in the `parameters` class variable. Concrete classes must implement its `run()` and `getResult()` methods. Class `ExportEmployeesWork` in step 2 is an example of a concrete implementation of this class.
- `ExtWorkListener`: Implements the `commonj.work.WorkListener` interface.

In this recipe, we have identified the functionality of exporting data from a database table, which was originally implemented in *Exporting data to a file*, *Chapter 7*, *Face Value: ADF Faces, JSPX Pages and Components*, that can run on the work manager. It is implemented by the method `exportEmployees()` in the `HrComponentsAppModuleImpl` custom application module implementation class, part of the `HRComponents` workspace. Steps 2 and 3 illustrate how it is done:

- Create a class that extends the `ExtWork` class. This class identifies a piece of code that can run on a work manager. In our case, this was done with the `ExportEmployeesWork` class in step 2. The actual code that will be executed is then implemented by the `run()` method of the class.
- Create an `ExtWorkManager` class and call its `addWork()` method to add specific pieces of "work" to be executed by it. These are classes that extend `ExtWork`. In our case, this was done in step 3 when we called `addWork()` specifying `ExportEmployeesWork` as the specific `ExtWork` class:

  ```
  mngr.addWork(new ExportEmployeesWork(getEmployees().
    createRowSetIterator(null)));
  ```

- Call the `ExtWorkManager` class `run()` method to commence with the execution of the works added to the work manager.

Observe the constructor of the `ExtWork` derived classes. It accepts a variable number of parameters that are stored in the `parameters` class variable. For instance, in our example, the `ExportEmployeesWork` was constructed specifying the `Employees` `RowSetIterator`, as shown in the following line of code:

```
new ExportEmployeesWork(getEmployees().createRowSetIterator(null))
```

These parameters can then be accessed as shown in the `ExportEmployeesWork` `run()` method, as follows:

```
RowSetIterator iterator = (RowSetIterator)parameters.get(0);
```

To retrieve the results produced by the work manager, you iterate over the `ExtWork` works and you call `getResult()` for each one. The works managed by the work manager are retrieved by calling `getResult()` on it. This is implemented in step 3 and is shown as follows:

```
List<ExtWork> works = mngr.getResult();
StringBuilder employeeStringBuilder = new StringBuilder();
for (ExtWork work : works) {
  ExportEmployeesWork exportWork = ExportEmployeesWork)work;
  employeeStringBuilder.append(exportWork.getResult());
}
```

As you can see in step 2, the ExportEmployeesWork getResult() method returns the CSV string buffer employeeStringBuilder that was built in the run() method when iterating over the Employees view object:

```
public Object getResult() {
  // return the employees CSV string buffer
  return employeeStringBuilder;
}
```

Work manager export functionality is added in a separate HrComponentsAppModule method called exportEmployeesOnWorkManager which is then added to the application module's client interface and once bound, (step 6) it is invoked from a backing bean (step 7).

Steps 8 through 15 show how to create, configure, and reference a work manager. In step 8, we reference the work manager in our application by adding a resource reference to it in the web.xml deployment descriptor. The work manager that we will be creating in steps 9 through 15 is called MyWorkManager. We use this reference to get hold of the work manager via JNDI lookup in our code. This is done in the ExtWorkManager run() method in step 1 as shown in the following code snippet:

```
InitialContext ctx = new InitialContext();
workManager = (WorkManager)ctx.lookup
("java:comp/env/" + managerName);
```

In this case, managerName is specified during the construction of the ExtWorkManager. This can be seen in step 3 when the work manager is constructed:

```
ExtWorkManager mngr = new ExtWorkManager("MyWorkManager");
```

Steps 9 through 15 detail the steps of creating and configuring a global work manager in WebLogic Server. Observe how in step 15 we have enabled the **Ignore Stuck Threads** setting. This will enable us to run long-running works on the work manager without getting an indication of a stuck thread by WebLogic. A WebLogic thread that executes for more than a specified-preconfigured amount of time is considered by WebLogic to be "stuck". If the number of the stuck threads in an application grow, the application might crash.

Finally, observe how the work manager is started in the ExtWorkManager run() method in step 1. The list of works added to the work manager (by calling its addWork() method) is iterated and each work is scheduled for execution by calling its schedule() method.

```
workList = new ArrayList<WorkItem>();
for (ExtWork work : works) {
  WorkItem workItem =
    workManager.schedule(work, workListener);
  workList.add(workItem);
}
```

The `schedule()` method returns a `commonj.work.WorkItem`, which is added to a `java.util.List`. We use this list to commence the execution of the work manager by calling its `waitForAll()` method:

```
workManager.waitForAll(workList, waitType);
```

One important thing to notice here is the `waitType` argument passed to the `waitForAll()` method. It can take either of the following two values:

- `WorkManager.INDEFINITE`: Calling code pauses, waiting until the execution of all works scheduled on the work manager completes.
- `WorkManager.IMMEDIATE`: Return is passed immediately to the calling code running the works scheduled on the work manager concurrently.

Furthermore, observe that the `WorkItem` list is iterated in the `getResult()` method to retrieve the result for each `WorkItem`, as shown in the following code snippet:

```
for (WorkItem workItem : workList) {
  resultList.add((ExtWork)workItem.getResult());
}
```

There's more...

For more information on work managers, consult sections *Description of the Work Manager API* and *Work Manager Example* in the *Timer and Work Manager API (CommonJ) Programmer's Guide for Oracle WebLogic Server* documentation manual. This can be found in the *WebLogic Server Documentation Library* currently at `http://docs.oracle.com/cd/E14571_01/wls.htm`.

See also

- *Breaking up the application in multiple workspaces*, *Chapter 1*, *Pre-requisites to Success: ADF Project Setup and Foundations*
- *Overriding remove() to delete associated children entities*, *Chapter 2*, *Dealing with Basics: Entity Objects*
- *Exporting data to a file*, *Chapter 7*, *Face Value: ADF Faces, JSPX Pages and Components*
- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*
- *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Monitoring the application using JRockit Mission Control

JRockit Mission Control is a suite of tools that can be used to monitor, profile, and manage applications deployed on the WebLogic Server running on the JRockit JVM. Moreover, the JRockit Mission Control tools allow you to record and replay sessions, perform garbage collection on demand, and eliminate memory leaks.

In this recipe, we will go over the installation of JRockit Mission Control Client and the steps necessary to configure the WebLogic Server to run it. Then we will look into a monitor session of a standalone WebLogic server instance.

Getting ready

You will need a standalone WebLogic server domain configured and your ADF application deployed on it. For information about these topics, take a look at *Configuring and using the Standalone WebLogic Server* and *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*.

How to do it...

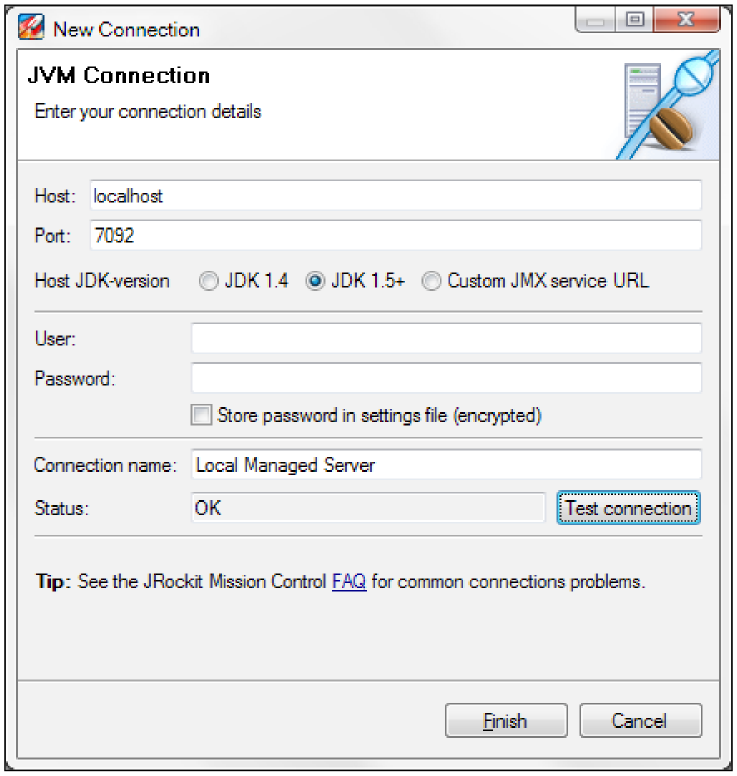
1. Download the appropriate JRockit version for your client operating system by going to the **Oracle JRockit Downloads** page. This page is currently accessible via the following URL: `http://www.oracle.com/technetwork/middleware/jrockit/downloads/index.html`.
2. Start the installation by executing the file downloaded. Make sure that during the installation, you choose to install the JRockit JRE as well.
3. Once the installation completes, ensure that you can run the JRockit Mission Control Client by running the `jrmc` program in the target installation directory.
4. Edit the `setDomainEnv` script in the WebLogic Server domain `bin` directory and ensure that the `JAVA_VENDOR` variable is set to `Oracle`. Also, verify that the `BEA_JAVA_HOME` is set correctly to JRockit JDK home directory on the WebLogic server machine. Finally, update the `JAVA_OPTIONS` environment variable to:

   ```
   set JAVA_OPTIONS=%JAVA_OPTIONS% %MNGMNT_CONCOLE_OPTIONS%
   ```

5. Edit the `startManagedWebLogic` script (in the same directory) and add the following lines:

   ```
   if "%SERVER_NAME%"=="ManagedServer1" (set MNGMNT_CONCOLE_OPTIONS=-
     Xmanagement:ssl=false, authenticate=false -
     Dcom.sun.management.jmxremote.port=7092)
   ```

6. Restart the WebLogic Server domain and ensure that when starting the server instance configured for the management console, the JMX connectors are started.
7. Start the JRockit Mission Control as indicated earlier. Right-click anywhere in the **JVM Browser** and select **New Connection**.
8. In the **New Connection** dialog, specify the standalone WebLogic server **Host** name or IP and the management connection **Port**. Enter a **Connection name** and click on the **Test connection** button to test the connection. Once successful, click on the **Finish** button.



Host: localhost

Port: 7092

Host JDK-version ○ JDK 1.4 ● JDK 1.5+ ○ Custom JMX service URL

User:

Password:

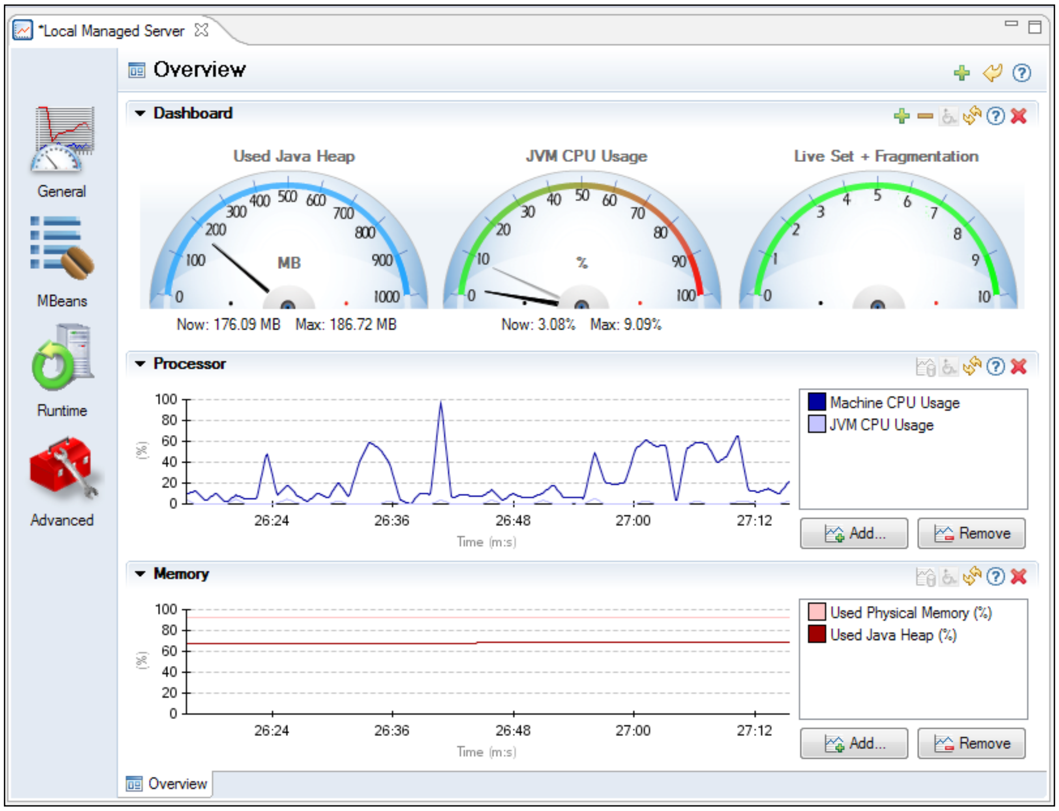
☐ Store password in settings file (encrypted)

Connection name: Local Managed Server

Status: OK

Tip: See the JRockit Mission Control FAQ for common connections problems.

9. The connection should appear under the **Connectors** node in the **JVM Browser** tree. Now, right-click on the connection and select **Start Console**. The JRockit management console **Overview** tab will be displayed, monitoring the WebLogic standalone managed server instance.



How it works...

Steps 1 through 3 go through the process of downloading and installing JRockit Mission Control. The installation process is straightforward; simply run the downloaded executable file and follow the installation wizard. As noted in step 2, ensure that the JRockit JRE is also installed.

Steps 4 through 6 demonstrate how to start a WebLogic managed server instance with management console options enabled. This will allow us to connect to it using the JRockit Mission Control Client (steps 7 through 9). First we need to ensure that the WebLogic Server is started with the JRockit JVM. This can be done by specifying `Oracle` for the `JAVA_VENDOR` environment variable in the `setDomainEnv` script (see step 4). You will also need to specify the location of the JRockit JDK path on the WebLogic Server machine using the `BEA_JAVA_HOME` environment variable (also in step 4).

In the same script file, we have also updated the `JAVA_OPTIONS` environment variable to include additional options related to the management console. These options are defined using a new environment variable called `MNGMNT_CONCOLE_OPTIONS` (step 4). Then, in step 5, we have defined the management console options specifically for our managed server instance (it is called `ManagedServer1` for this recipe). We have used the `-Xmanagement:ssl=false,authenticate=false` JVM argument to indicate that no authentication (and no SSL connection) will be required for the management console. This will allow us in step 8, when we define the JVM connection, to specify that no authentication credentials are required to access the JVM. We have also indicated the management connection port (it was set to 7092 for this recipe). In step 6, we restarted the WebLogic Server with the new management connection options.

In steps 7 through 9, we started the JRockit Mission Control Client and created a connection to the WebLogic managed server instance configured earlier (in step 8). In step 9, we started the management console to monitor the JRockit JVM instance configured. By default, the management console **Overview** tab includes a **Dashboard** with predefined Java Heap and JVM CPU dials, and monitors for the **Processor** (machine and JVM CPU usage) and **Memory** (used machine and Java heap memory). Additional JVM run-time metrics can be added to the management console by clicking on the **Add Dial** (the green plus sign icon) and the **Add...** buttons.

There's more...

In addition to the management console, the JRockit Mission Control Client includes the **flight recorder** and **memory leak detector** tools. These tools are available by right-clicking in the **JVM Browser** and selecting **Start Flight Recording...** and **Start Memleak** from the context menu respectively. For more information on these tools, consult the *JRockit JDK Tools Guide* and *JRockit Flight Recorder Run Time Guide*. These documents can be found in the *JRockit Documentation Library* currently at `http://docs.oracle.com/cd/E15289_01/index.htm`.

See also

- *Configuring and using the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*
- *Deploying on the Standalone WebLogic Server*, *Chapter 10*, *Deploying ADF Applications*

Index