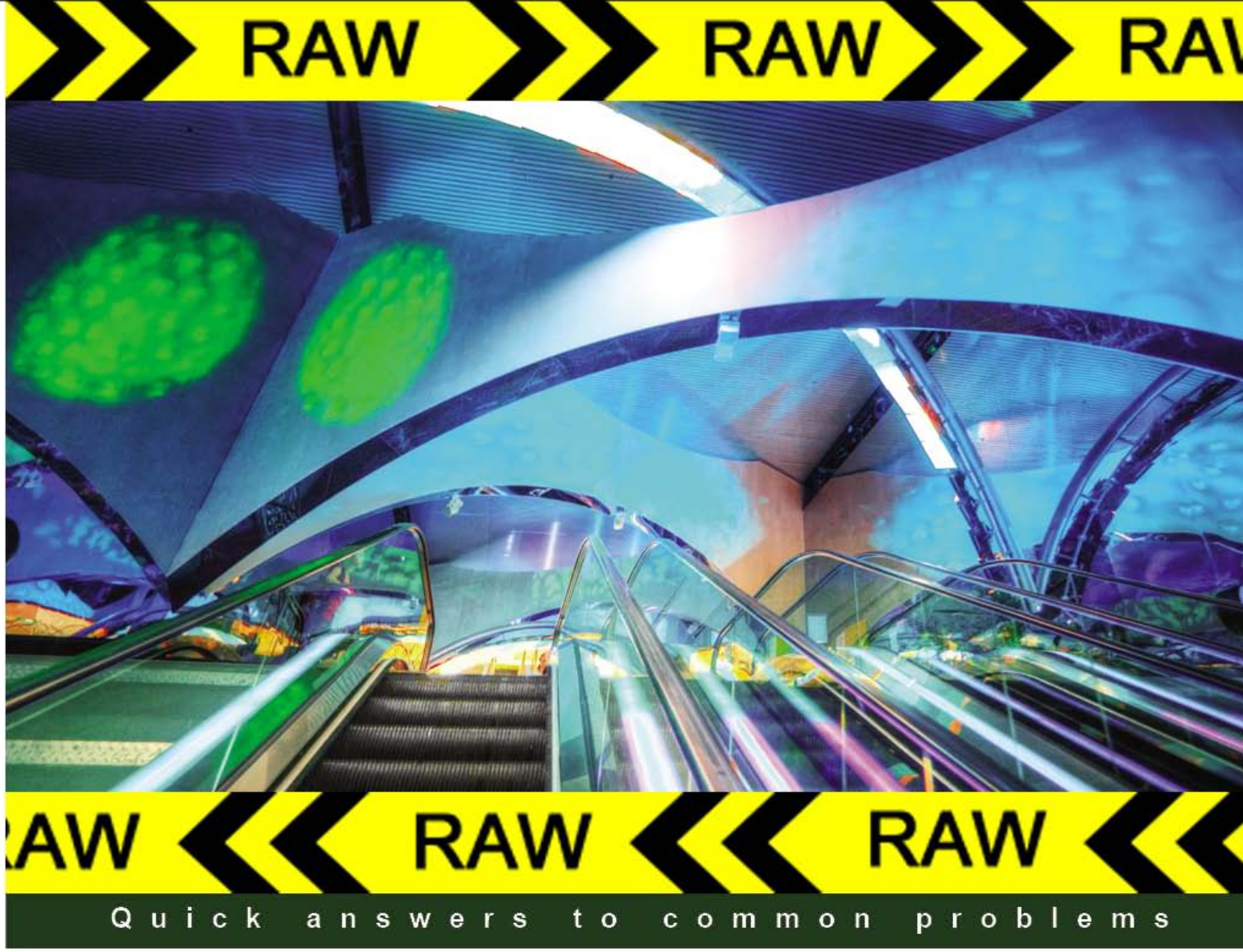


Oracle JDeveloper 11gR2 Cookbook: RAW

Over 120 simple but incredibly effective recipes for using JDeveloper 11gR2 to build ADF applications

Nick Haralabidis

[PACKT] PUBLISHING
enterprise
professional expertise distilled

Oracle JDeveloper 11g Cookbook

RAW Book

Over 120 simple but incredibly effective recipes for using JDeveloper 11gR2 to build ADF applications with this book and e-book

Nick Haralabidis

BIRMINGHAM - MUMBAI

Oracle JDeveloper 11g Cookbook

Current RAW Publication: OCTOBER 2009

RAW Production Reference: 2031011

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-84968-476-7

www.packtpub.com

About the Author

Nick Haralabidis has over 20 years experience in the Information Technology industry, and a multifaceted career in such positions as Senior IT Consultant, Senior Software Engineer and Project Manager for a number of U.S. and Greek corporations (Compuware, Chemical Abstracts Service, NewsBank, CheckFree, Intrasoft International, Unisystems, MedNet International and others). His many years of experience have exposed him to a wide range of technologies ranging from Java, J2EE, C++, C, and Tuxedo, to a number of database technologies.

For the last four years Nick has been actively involved in large implementations of next generation enterprise applications utilizing Oracle's JDeveloper, Application Development Framework (ADF) and SOA technologies.

He holds a B.S. in Computer Engineering and an M.S. in Computer Science from the University of Bridgeport.

When he is not pursuing ADF professionally, he writes on his blogs JDeveloper Frequently Asked Questions (`http://jdeveloperfaq.blogspot.com`) and ADF Code Bits (`http://adfcodebits.blogspot.com`). He is active in the Oracle Technology Network (OTN) JDeveloper and ADF forum where he both learns from and helps other forum users.

Table of Contents

Preface

Welcome to Oracle JDeveloper 11g Cookbook the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

This book contains a wealth of resources covering Oracle's JDeveloper 11g release and the Application Development Framework (ADF) and how these technologies can be applied for the design, construction, testing and optimizing of Fusion Web applications. Being a vast and complex framework, an attempt has been made to cover a wide range of topics related specifically to Fusion Web Application development with ADF. These topics are presented in the form of recipes most of them derived from the author's working experience covering real world use cases. The topics include, but not limited to, foundational recipes related to laying out the project groundwork, recipes related to the ADF Business Components, recipes related to ViewController, recipes related to security, optimization and so on.

In the maze of information related to Fusion Web Application development with ADF, it is the author's hope that aspiring ADF developers will find in this book the information they are looking for. So lift up your sleeves, put on your ADF cooking hat, pick up a recipe or two and let's start cooking!

What's in This RAW Book

In this RAW book, you will find these chapters:

Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations covers a number of recipes related to foundational concepts related to Fusion Web Application development with ADF. By applying and expanding as needed these foundational recipes during the early architectural and design phases, subsequent application development takes a form, a structure and the necessary uniformity. Many if not most of the recipes on the following chapters rely on these recipes.

Chapter 2, Dealing with Basics: Entity Objects starts our journey into the wonderful world of ADF Business Components (BC). First stop: Entity objects. The recipes in this chapter deal with some of the most common framework functionality that is overridden in real world applications to provide customized business functionality.

Chapter 3, A Different Point of View: View Objects Techniques covers a number of recipes related to View objects. This chapter explains how to control attribute updatability, how to set attribute default values, how to iterate View object rowsets, and many more.

What's Still to Come?

We mentioned before that the book isn't finished, and here is what we currently plan to include in the remainder of the book:

Chapter 4, Important Contributors: List of Values, Bind Variables, View Criteria

Chapter 5, Putting them all together: Application

Chapter 6, Go with the Flow: Task Flows

Chapter 7, Face Value: ADF Faces, JSPX Pages and Components

Chapter 8, Backing not Baking: Bean Recipes

Chapter 9, Handling Security, Session Timeouts, Exceptions and Errors

Chapter 10, Deploying ADF Applications

Chapter 11, Refactoring, Debugging, Profiling, Testing

Chapter 12, Optimizing, Fine-tuning and Monitoring

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We retrieve the authenticated user's name by calling `getUserPrincipal().getName()` on the `SecurityContext`."

A block of code will be set as follows:

```
public void adjustCommission(BigDecimal commissionPctAdjustment) {
// get the service proxy
    HrComponentsAppModuleService service =
           (HrComponentsAppModuleService)ServiceFactory
                       .getServiceProxy(HrComponentsAppModuleService.
NAME);
// call the adjustCommission() service
    service.adjustCommission(commissionPctAdjustment);
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
<StringRefAddr addrType="jndiFactoryInitial">
<Contents>weblogic.jndi.WLInitialContextFactory</Contents>
</StringRefAddr>
<StringRefAddr addrType="jndiProviderURL">
<Contents>t3://localhost:7101</Contents>
</StringRefAddr>
```

Any command-line input and output is written as follows:

```
$ chmod u+x ./jdevstudio11120install.bin
```

New terms and important words are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "On the **Choose Install Type** page select **Complete** to ensure that JDeveloper, ADF and WebLogic Server are installed".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

What Is a RAW Book?

Buying a Packt RAW book allows you to access Packt books before they're published. A RAW (Read As we Write) book is an eBook available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

Is a RAW Book a Proper Book?

Yes, but it's just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final version is published.

When Do Chapters Become Available?

As soon as a chapter has been written and we are happy for it to go into the RAW book, the new chapter will be added into the RAW eBook in your account. You will be notified that another chapter has become available and be invited to download it from your account. eBooks are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

How Do I Know When New Chapters Are Released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new eBook. Packt will also update the book's page on its website with a list of the available chapters.

Where Do I Get the Book From?

You download your RAW book much in the same way as any Packt eBook. In the download area of your Packt account, you will have a link to download the RAW book.

What Happens If I Have Problems with My RAW Book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

Is There Source Code Available During the RAW Phase?

Any source code for the RAW book can be downloaded from the **Support** page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

How Do I Post Feedback and Errata for a RAW Title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawbooks@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

1

Prerequisites to Success: ADF Project Setup and Foundations

In this chapter, we will cover:

- Installation of JDeveloper on Linux
- Breaking up the application in multiple workspaces
- Setting up BC base classes
- Setting up logging
- Using a custom exception class
- Using a stored procedure helper class
- Using ADFUtils/JSFUtils
- Using page templates
- Using a generic View Controller actions framework

Introduction

JDeveloper and **ADF (Application Development Framework)** are amazing technologies. What makes them even more incredible is their sheer complexity and the amount of knowledge and effort put into them which lays covered underneath the declarative – almost magical - front. What amazes me is that once you scratch the surface, you never stop realizing how much you really don't know. Given this complexity, it becomes obvious that certain development guidelines and practices must be established and followed early in the architectural and design phases of an ADF project. Establishing these development practices and following them consistently throughout the development process will ensure that things are done consistently and according to these practices and standards.

This chapter presents a number of recipes that are geared towards establishing some of these development practices. In particular you will see content which serves as a starting point in making your own application modular when using the underlying technologies. You will also learn the importance of extending the **Business Components** framework base classes early in the development cycle. We will talk about the importance of laying out other application foundational components, again early in the development process, and continue by presenting a helper class that you can use for calling database stored procedures in the *Using a stored procedure helper class* recipe. Finally we will address reusability and consistency at the **ViewController** layer**.**

The chapter starts with a recipe about installing and configuring JDeveloper on Linux. So, let's get started and don't forget, have fun as you go along. If you get in trouble at any point, take a look at the accompanying source code or feel free to contact me anytime at `nharalabidis@gmail.com`.

Installation of JDeveloper on Linux

Installation of JDeveloper is in general a straightforward task. So, "why have a recipe for this?", you might ask. Did you notice the title? It says "on Linux". You will be amassed of the number of questions asked about this topic on a regular basis on the *JDeveloper and ADF OTN Forum*. Besides, in this recipe we will also talk about configuration options and the usage of 64-bit JDK along with JDeveloper.

Getting ready

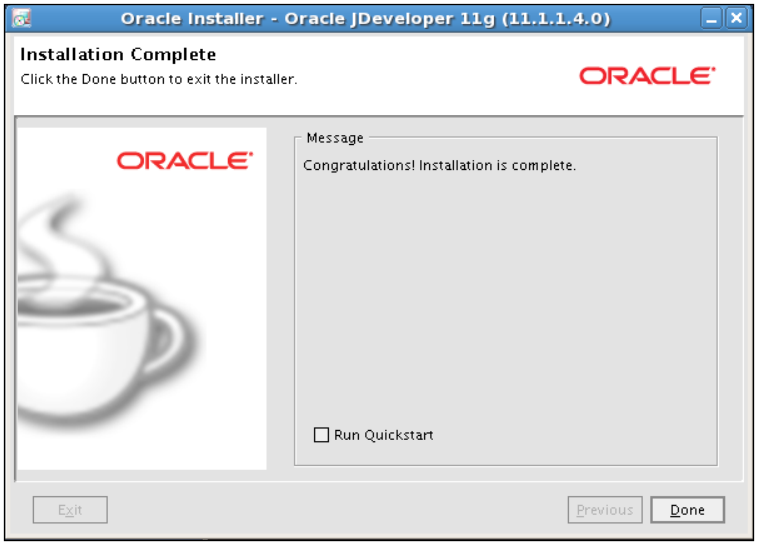
You will need a Linux installation to go along with this recipe. For the 64-bit configuration you will need a 64-bit Linux distribution and a 64-bit version of the Java SDK. We will install the latest version of JDeveloper which is version 11.1.2.0.0 at the time of this writing.

How to do it

1. Download JDeveloper from the **Oracle JDeveloper Software** download page: `http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html`.
2. Accept the license agreement, select **Linux Install,** and click **Download** to begin with the download.
3. Once the file is downloaded, open a console window, give the file execute permissions, and start the installation as shown below:

```
$ chmod u+x ./jdevstudio11120install.bin
$ ./jdevstudio11120install.bin
```

4. On the **Choose Middleware Home Directory** page, select **Create a new Middleware Home** and enter the **Middleware Home Directory**.
5. On the **Choose Install Type** page select **Complete** to ensure that JDeveloper, ADF and WebLogic Server are installed.
6. Once you confirm your selections, proceed with the installation.
7. Upon a successful installation you will see the **Installation Complete** page. Uncheck the **Run Quickstart** checkbox and click **Done**.



8. To start JDeveloper go to the `/jdeveloper/jdev/bin` directory under the Middleware Home Directory you specified during the installation and type:

```
$ ./jdev
```

9. To make things easier, create an application launcher on your Linux desktop for the specific path indicated above.

How it works

As noted earlier, installing JDeveloper on Linux is a straightforward task. You simply have to download the binary executable archive and run it. Ensure that you give execute permissions to the installation archive file and run it as noted. If you are having trouble seeing the **Welcome** page in graphical mode, ensure that the **$DISPLAY** environment variable is set correctly. The important thing to know here is the name of the file to execute in order to start JDeveloper. As mentioned, it is called `jdev` and it is located in the `/jdeveloper/jdev/bin` directory under the Middleware Home Directory.

There's more

Now that you have successfully installed JDeveloper, let's spend some time to configure it for optimal performance. Configuration parameters are added to any of the `jdev.conf` or `ide.conf` files located in the `/jdeveloper/jdev/bin` and `/jdeveloper/ide/bin` directories respectively under the Middleware Home Directory.

Here is a list of the important tuning configuration parameters with some recommendations for their values:

Parameter	*Description*
AddVMOption -Xmx	*This parameter is defined in the* `ide.conf` file and indicates the maximum that you will allow the JVM heap size to grow to. In plain words the maximum memory that JDeveloper will consume on your system. When setting this parameter consider the available memory on your system, the memory needed by the OS, the memory needed by other applications running concurrently with JDeveloper and so on. On a machine used exclusively for development with JDeveloper as a general rule of thumb consider setting it to around 50% of the available memory.
AddVMOption -Xms	*This parameter is also defined in the* `ide.conf` and indicates the initial JVM heap size. This is the amount that will be allocated initially by JDeveloper and it can grow up to the amount specified by the –Xmx parameter above. When setting this parameter consider whether you want to give JDeveloper a larger amount in order to minimize frequent adjustments to the JVM heap. Setting this parameter to the same value as the one indicated by the –Xmx parameter will supply a fixed amount of memory to JDeveloper.

AddVMOption -XX:MaxPermSize	*This parameter indicates the size of the JVM permanent generation used to store class definitions and associated metadata. Increase this value if needed in order to avoid java.lang.OutOfMemoryError: PermGen space* errors.
AddVMOption -DVFS_ENABLE	*Set it to true in* `jdev.conf` if your JDeveloper projects consist of a large number of files especially if you will be enabling a version control system from within JDeveloper.

Configuring JDeveloper with a 64-bit JDK

The JDeveloper installation is bundled by default with a 32-bit version of the Java JDK, which is installed along with JDeveloper. On a 64-bit system consider running JDeveloper with a 64-bit version of the JDK. First download and install the 64-bit JDK. Then configure JDeveloper via the `SetJavaHome` configuration parameter in the `jdev.conf`. This parameter should be changed to point to the location of the 64-bit JDK.

Configuring the JDeveloper user directory

This is the directory used by JDeveloper to identify a default location where files will be stored. JDeveloper also uses this location to create the integrated WebLogic domain and to deploy your Web applications when running them or debugging them inside JDeveloper. It is configured via the `SetUserHomeVariable` parameter in the `jdev.conf` file. It can set to a specific directory or to an environment variable usually named `JDEV_USER_DIR`.

Before starting your development in JDeveloper, consider setting the XML file encoding for the XML files that you will be creating in JDeveloper. These files include, among others, the JSF pages, the Business Component metadata files, application configuration files, and so on. You set the encoding via the **Tools à Preferences...** menu. Select the **Environment** node on the left of the **Preferences** dialog and the encoding from the **Encoding** drop down. The recommended setting is **UTF-8** to support multi-lingual applications.

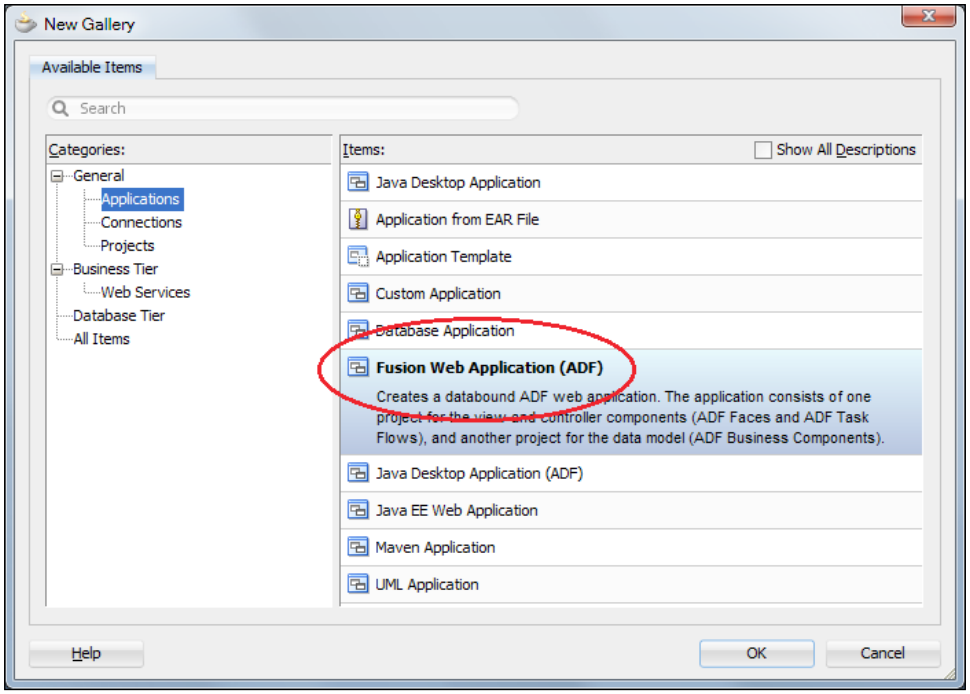
The minimum recommended open file descriptors limit for JDeveloper on a Linux system is 4096. Use the command `ulimit -n` to determine the open file descriptors limit for your installation and change it if needed in the `limits.conf` file located in `/etc/security/` directory.

Breaking up the application in multiple workspaces

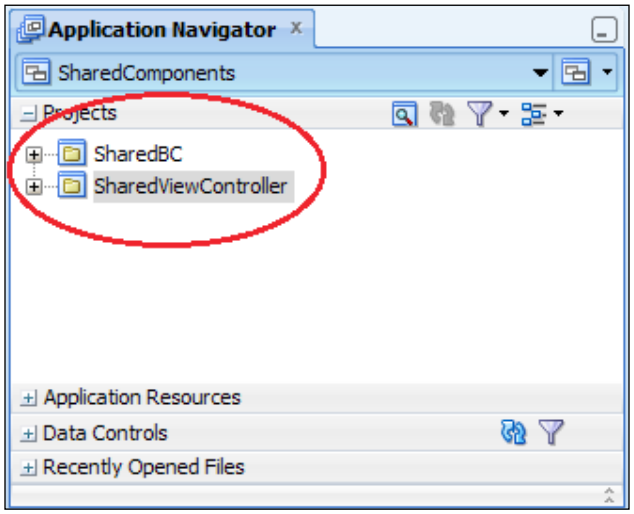
When dealing with large enterprise scale applications, the organization and structure of the overall application in terms of JDeveloper workspaces, projects, and libraries is essential. Organizing and packaging ADF application artifacts such as **Business Components**, **Task Flows**, **Templates**, java code, and so on in libraries will promote and ensure modularity, and the reuse of these artifacts throughout the application. In this recipe we will create an application that is comprised of reusable components. We will construct reusable libraries for shared components, business domain specific components, and a main application for consuming these components.

How to do it

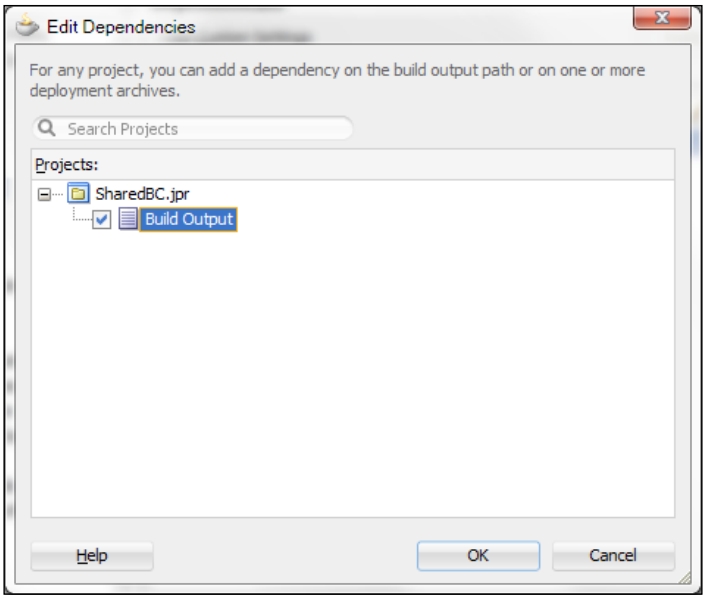
1. To create the shared components library, start by selecting **New Application...** in the **Application Navigator**. This will start the application creation wizard.
2. In the **New Gallery** dialog click on the **Applications** node (under the **General** category) and select **Fusion Web Application (ADF)** from the **list of Items**.



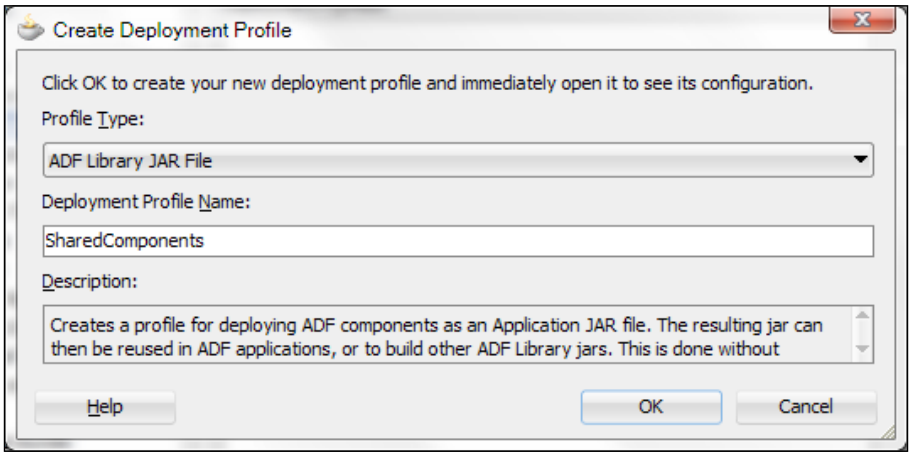
3. In the **Name your application** page enter the **Application Name**, **Directory** and the **Application Package Prefix**.
4. In the **Name your project** page, enter the Business Components **Project Name** and **Directory**.
5. In the **Configure Java settings** page for the Business Components project, accept the defaults for **Default Package**, **Java Source Path** and **Output Directory**.
6. Similarly in the **Name your project** page for the **ViewController** project enter the **Project Name** and **Directory**.
7. Accept the defaults in the **Configure Java settings** and click **Finish** to proceed with the creation of the workspace.
8. Now in the **Application Navigator** you should see the two projects comprising the shared components workspace on for the **BC**s and another for the **ViewController**.



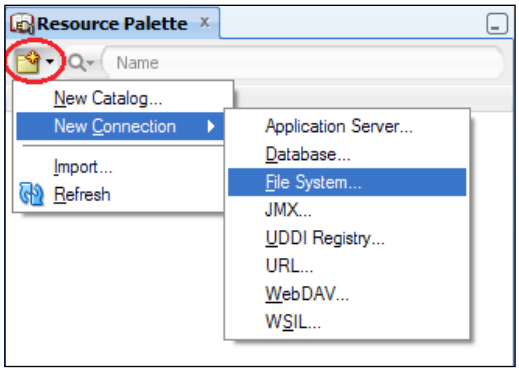
9. You will be using this workspace to add reusable **BC** and **ViewController** components. For now we will package the workspace into an **ADF library JAR** without any components yet in it. In order to do this you will need to first setup the project dependencies. Double-click on the **ViewController** project to bring up **the Project Properties** dialog and select **Dependencies**.
10. Click on **Edit Dependencies** (the small pen icon) to bring up the **Edit Dependencies** dialog and then click on the **Build Output** checkbox.
11. Click **OK** to close the dialog and return to the **Project Properties** dialog.



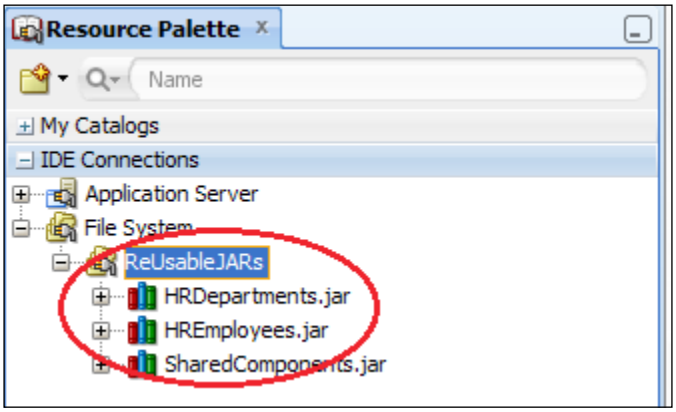
12. The next step is to set up the **Deployment Profile**. While at the **ViewController Project Properties** dialog, click on the **Deployment** node.
13. Since we will not be deploying this application as a WAR, select the default WAR deployment profile that was generated automatically by JDeveloper and delete it.
14. Then click **New...** to create a new deployment profile.
15. On the **Create Deployment Profile** dialog select **ADF Library JAR File** for the **Profile Type** and enter the name of the deployment profile. For this recipe we have called the deployment profile **SharedComponents**. Click **OK** to proceed with its creation.



16. In the **Edit ADF Library JAR Deployment Profile Properties** dialog that is opened, select **JAR Options** and specify a location where you will be placing all the reusable JAR libraries. For this recipe we will place all reusable libraries in a directory called `ReUsableJARs`.
17. When done, exit completely from the **Project Properties** dialog, saving your changes by clicking **OK**.
18. The last step involves the creation of the **ADF Library JAR**. You do this by right-clicking on the **ViewController** project in the **Application Navigator** selecting **Deploy** and then the name of the deployment profile name (**SharedComponents** in this case).
19. Select **Deploy to ADF Library JAR file** in the **Deployment Action** page and click **Finish** to initiate the deployment process. The deployment progress will begin. Its status is shown in the **Deployment** tab of the **Log** window.
20. To create the `HRDepartments` components library, similarly create a new **Fusion Web Application** for the `HR Department` components. Follow the same steps as above to setup the project dependencies.
21. Create the deployment profile and deploy the ADF Library JAR. We will not be placing any components yet in this library.
22. To create the `HREmployees` components library, repeat the steps above once more in order to create another **ADF Library JAR** for the `HR Employee`-related reusable components.
23. Now create another Fusion Web Application, which will be used as the main application. This application will consume any of the components that reside in the ADF Library JARs that you created in the previous steps.
24. This can easily be done via the **Resource Palette** by creating a **File System** Connection to the directory where we saved the reusable ADF Library JARs, i.e. the directory called `ReUsableJARs`.
25. If the **Resource Palette** is not visible, select **View** à **Resource Palette** to show it. In the **Resource Palette** click on the **New** icon and select **New Connection** à **File System...**



26. In the **Create File System Connection** dialog that is displayed, enter the name of the connection and the directory where you have deployed the reusable components in the previous steps.
27. Click **OK** to continue. You should be able to see the new **File System Connection** in the **Resource Palette.**



28. To consume reusable components, first select the appropriate project on the **Application Navigator**, then right-click on the **ADF Library JAR** on the **Resource Palette** and select **Add to Project...**
29. On the **Confirm Add ADF Library** dialog click on the **Add Library** button to proceed.
30. Alternatively expand the ADF Library JAR and drag-and-drop the reusable component onto its appropriate place in the workspace.

How it works

When you deploy a project as an ADF Library JAR, all ADF reusable components and code are packaged in it and they become available to other consuming applications and libraries. Reusable components include Business Components, Database Connections, Data Controls, Task Flows, Task Flow Templates, Page Templates, Declarative Components, and of course Java code. By setting up the dependencies among the BC and ViewController projects in the way that we did, i.e. the build output of the Business Components project to be included during the deployment of the ViewController project, you will be producing a single ADF Library JAR file with all the components from all the projects in the workspace. When you add an ADF Library JAR to your project, the library is added to the project's class path. The consuming project can then use any of the components in library. The same happens when you drag and drop a reusable component into your project.

There's more

For this recipe we packaged both of the Business Components and ViewController projects in the same ADF Library JAR. If this strategy is not working for you, you have other options such as adjusting the dependencies among the two and packaging each project in separate ADF Library JARs. In this case you will need an additional deployment profile and a separate deployment for the Business Components project.

Adding the ADF Library JAR manually

You can add an ADF Library JAR into your project manually using the **Project Properties** dialog. Select the **Libraries and Classpath** node and click on the **Add Library...** button. This will display the **Add Library** dialog. On it, click the **New...** button to display the **Create Library** dialog. Enter a name for the library, select **Project** for the library location, and click on the **Deployed by Default** check button. Finally click on **Add Entry...** button to locate the ADF Library JAR. The **Deployed by Default** check box when checked indicates that the library will be copied during deployment of the consuming application to the application's destination archive. If you leave it unchecked, then the library won't be copied and it must be located in some other way (deployed separately as a shared library on the Application Server).

Defining the Application Module Granularity

One related topic that also needs to be addressed in the early architectural stages of the ADF project is the granularity for the Application Modules. By granularity it is meant how the data model will be divided into Application Modules. As a general rule of thumb, each Application Module should satisfy a particular use case. Related use cases and therefore Application Modules can then be packaged into the same reusable ADF Library JAR. In general, avoid creating too few monolithic Application Modules that satisfy a large number of use cases each.

Entities, List-of-Values (LOVs), Validation queries

Entities, List-of-Values (LOVs) and Validation queries should be defined only once for each Business Components project. To avoid duplication of Entities, LOVs and Validation queries around multiple Business Components projects, consider defining them only once in a separate Business Components Project.

Structuring of the overall ADF application in reusable components should be well thought and incorporated in the early design and architectural phases of the project.

As your application grows it is important to watch out for and eliminate circular dependencies among the reusable components that you develop. When they occur, this in most cases indicates a flow in your design. Use available dependency analyzer tools, such as **Dependency Finder** (it can be downloaded from `http://depfind.sourceforge.net`), during the development process to detect and eliminate any circular dependencies that may occur.

Setting up BC base classes

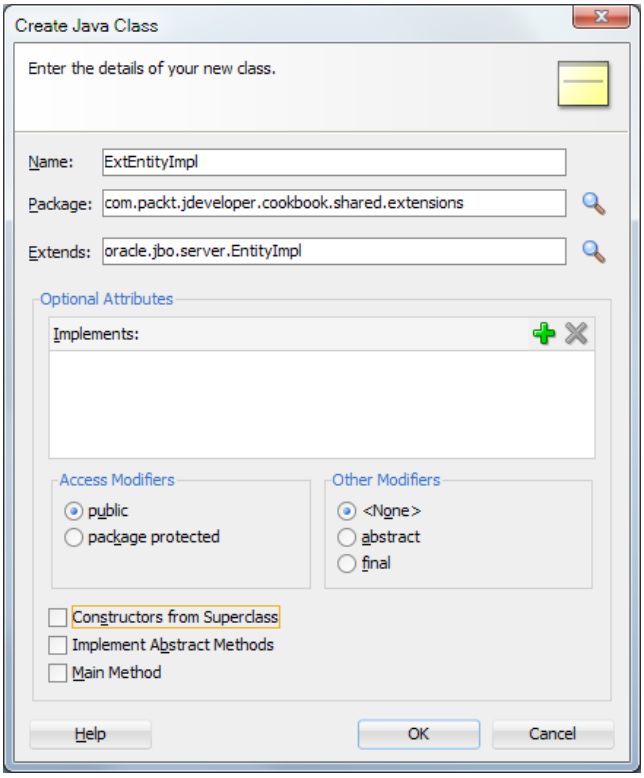
One of the first things to consider when developing large scale enterprise applications with ADF **Business Components** is to allow for the ability to extend the framework's base classes early on in the development process. It is imperative that you do this before creating any of your business objects, even though you have no practical use of the extended framework classes at that moment. This will guarantee that all of your business objects are correctly derived from your framework classes. In this recipe you will expand on the previous recipe and add Business Components framework extension classes to the shared components **ADF Library JAR**.

Getting ready

You will be adding the Business Components framework extension classes to the shared components project. See the section *Creation of the shared components library* in the previous recipe for information on how to create one.

How to do it

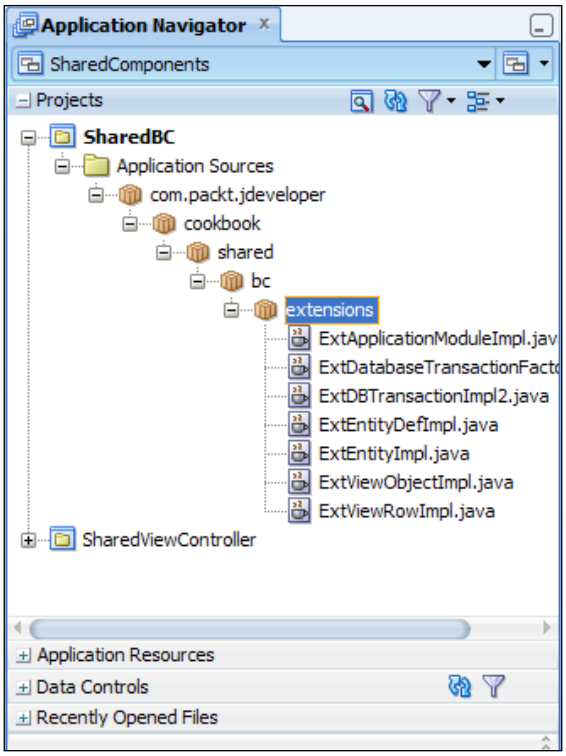
1. To create framework extension classes for the commonly used Business Components, start with the creation of an extension class for the **Entity Object**s. Open the shared components workspace in JDeveloper and right-click on the Business Components project.
2. From the context menu select **New...** to bring up the **New Gallery** dialog. Select **Class** from the **Java** category (under the **General** category) and click **OK.**
3. On the **Create Java Class** dialog that is displayed, enter the name of the custom Entity Object class, the package where it will be created, and for the **Extends** enter the base framework class, which in this case is `oracle.jbo.server.EntityImpl`.



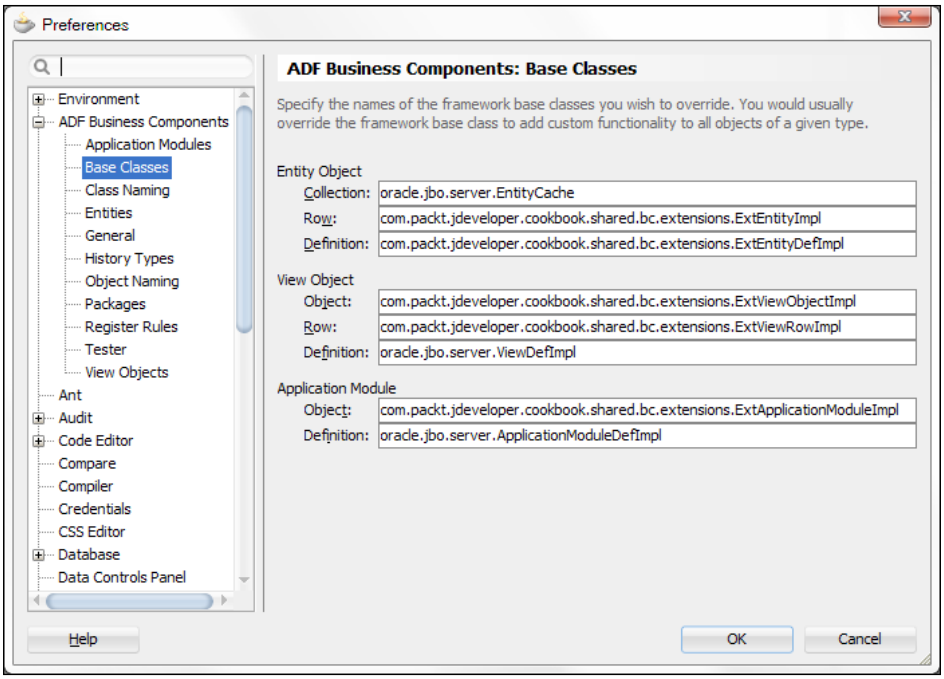
4. Now repeat the same steps to create framework extension classes for the following components:

Business Component	*Framework Class Extended*
Entity Definition	*oracle.jbo.server.EntityDefImpl*
View Object	*oracle.jbo.server.ViewObjectImpl*
View Row	*oracle.jbo.server.ViewRowImpl*
Application Module	*oracle.jbo.server.ApplicationModuleImpl*
Database Transaction Factory	*oracle.jbo.server.DatabaseTransactionFactory*
Database Transaction	*oracle.jbo.server.DBTransactionImpl2*

5. Once you are done, your project should look like this:



6. The next step is to configure JDeveloper so that all new Business Components that you create from this point forward will be inherited from the framework extension classes you've just defined. Open the **Preferences** dialog from the **Tools** menu, expand the **ADF Business Components** node and select **Base Classes**.
7. Then enter the framework extension classes that you created previously, each one in its corresponding category.



How it works

Defining and globally configuring Business Components framework extension classes via the **ADF Business Components Base Classes** settings on the **Preferences** dialog causes all subsequent Business Components that you create for all your projects to inherit from them. This is true for both XML-only components and for components with custom java implementation classes. For XML-only components observe that the `ComponentClass` attribute in the object's XML definition file points to your framework extension class.

There's more

You can configure your Business Components framework extension classes at two additional levels. At project level and at the individual component level.

- Configuration at the project level is done via the **Project Properties Base Classes** selection under the **ADF Business Components** node. These configuration changes will affect only the components created for the specific project.
- Configuration at the component level is done via the component's **Java Options** dialog - in the component's definition **Java** page - by clicking on the **Classes Extend...** button and overriding the default settings. The changes will only affect the specific component.

Do not directly attempt to change or remove the `extends` Java keyword in your component's implementation class. By doing so you will be making half the change because the component's XML definition will still point to the original class. Instead use the **Classes Extend...** button on the component's **Java Options** dialog to make the change.

See also

Creating and using generic extension interfaces, chapter 5

Breaking up the application in multiple workspaces, chapter 1

Setting up logging

Logging is one of those areas during the initial phases of application design that does not necessarily take the merit that it should. There are a few obvious logging framework choices to use in your application – one commonly used is **Apache**'s **log4j** - and others less obvious. In this recipe we will demonstrate the usage of the `ADFLogger` and **Oracle Diagnostics Logging** (**ODL**). The main advantage of using ODL when compared to other logging frameworks is its tight integration with WebLogic and JDeveloper. In WebLogic, the logs produced conform to and integrate with the diagnostics logging facility. Diagnostic logs include, in addition to the message logged, additional information such as the session and user that produced the log entry at run-time. This is essential when analyzing the logs. In JDeveloper the log configuration and analysis is integrated via the **Oracle Diagnostics Logging Configuration** and **Oracle Diagnostics Log Analyzer** respectively.

Getting ready

We will be adding logging to the **Application Module** framework extension class that we developed in the previous recipe.

How to do it

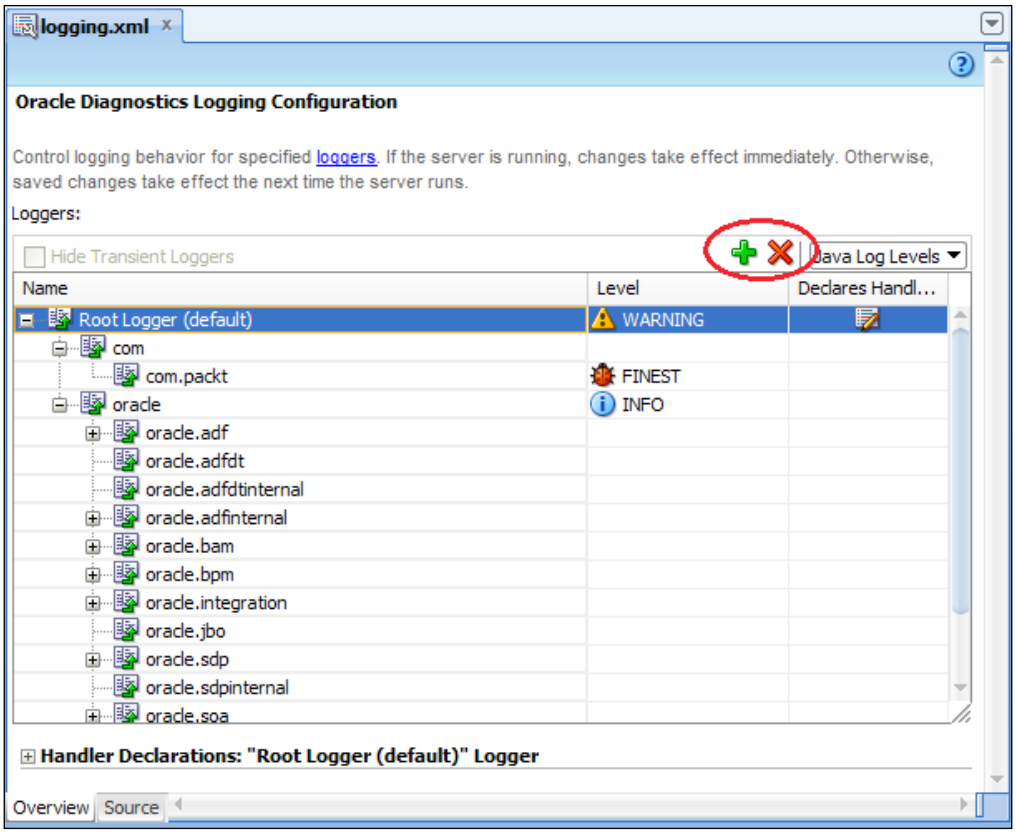
1. **ODL** logs can be generated programmatically from within your code by using the `ADFLogger` class. Instantiate an `ADFLogger` via the static `createADFLogger()` method and use its `log()` method.

Here is an example of adding logging support to the Application Module framework extension class we developed in the previous recipe:

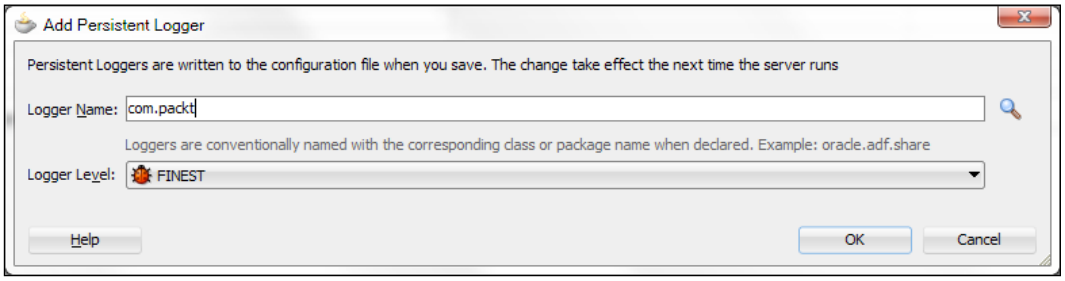
```
package com.packt.jdeveloper.cookbook.shared.bc.extensions;
```

```
import oracle.adf.share.logging.ADFLogger;
import oracle.jbo.server.ApplicationModuleImpl;

public class ExtApplicationModuleImpl extends ApplicationModuleImpl {

    // create an ADFLogger
    private static final ADFLogger LOGGER =
        ADFLogger.createADFLogger(ExtApplicationModuleImpl.class);

    public ExtApplicationModuleImpl() {
        super();
        // log a trace
        LOGGER.log(ADFLogger.TRACE,
                   "ExtApplicationModuleImpl was constructed");
    }
}
```

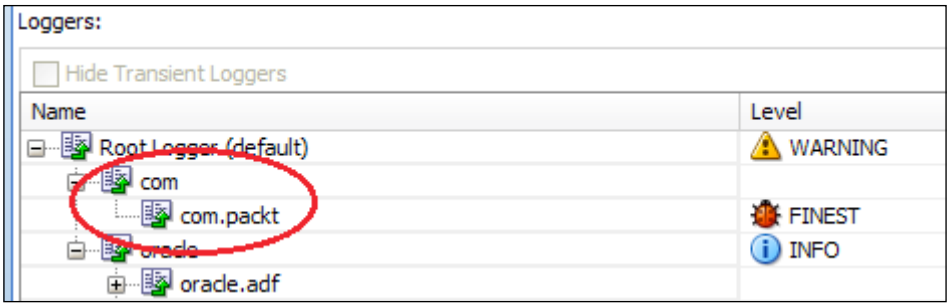
2. The next step involves the configuration of the logger in the logging configuration file `logging.xml`. The file is located in the `config\fmwconfig\servers` directory under the WebLogic domain for the appropriate server you are configuring.
3. For the integrated WebLogic server this file is located in the `%JDEV_USER_DIR%\system11.1.2.0.38.60.17\DefaultDomain\config\fmwconfig\servers\DefaultServer` directory. The exact location can vary slightly depending on the version of JDeveloper that you use.
4. Open the file in JDeveloper and create a custom logger called `com.packt` by clicking on the **Add Persistent Logger** icon as it shown below.



5. This will display the **Add Persistent Logger** dialog to add your logger. Enter `com.packt` for the **Logger Name** and choose `FINEST` for the **Logger Level**.



6. You will need to repeat this step to add another logger named `com` if one with that name does not exist. The final result should look like this:



7. One more step that is required to complete the configuration is to use the `-Djbo.debugoutput=adflogger` and `-Djbo.adflogger.level=FINEST` options when starting the JVM. You can do this in JDeveloper by double-clicking on the main application's **ViewController** project to bring up the **Project Properties** dialog and selecting the **Run/Debug/Profile** node.
8. Then select the appropriate **Run Configuration** on the right and click on the **Edit...** button.
9. On the **Edit Run Configuration** dialog that is displayed enter these java options in the **Java Options**.

How it works

In the example above we declared a `static ADFLogger` and we associated it with a specific class, namely `ExtApplicationModuleImpl` by passing `ExtApplicationModuleImpl.class` as a parameter during its construction. We declared the `ADFLogger` static so we don't have to worry about passivating it. We then used its `log()` method to do our logging. The `log()` method accepts a `java.util.logging.Level` parameter indicating the log level of the message and it can be any of the following values: `ADFLogger.INTERNAL_ERROR`, `ADFLogger.ERROR`, `ADFLogger.WARNING`, `ADFLogger.NOTIFICATION` and `ADFLogger.TRACE`.

`ADFLogger` leverages the **Java Logging API** to provide logging functionality. Because standard Java Logging is used, it can be configured through the `logging.xml` configuration file. This file is located under the WebLogic domain directory `config\fmwconfig\servers` for the specific server that you are configuring. The file is opened and a logger is added.

Logging is controlled at the package level. In the example above we added a logger for the `com.packt` package but we can fine-tune it for these additional levels: `com.packt.jdeveloper`, `com.packt.jdeveloper.cookbook`, `com.packt.jdeveloper.cookbook.shared` and so on. The class name that we passed as an argument to the `ADFLogger` during its instantiation, i.e. `ExtApplicationModuleImpl.class` represents a logger in the logging configuration file and can be defined as such.

Each logger configured in the `logging.xml` is associated with a log handler. There are a number of handlers defined in the `logging.xml` namely a `console-handler` to handle logging to the console, an `odl_handler` to handle logging for ODL and others.

There's more

You can also use the `ADFLogger` methods `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()` and `finest()` to do your logging.

When you configuring logging, ensure that you make the changes to the appropriate `logging.xml` file for the WebLogic server you are configuring.

See also

Configuring diagnostics logging, chapter 11

Dynamically configure ADF trace logs on WebLogic, chapter 11

Breaking up the application in multiple workspaces, chapter 1

Using a custom exception class

In this recipe we will go over the steps necessary to set up a custom application exception class derived from the `JboException` base exception class. Some of the reasons why you want to do this are to:

- Customize the exception error message
- Use error codes to locate the error messages in the resource bundle
- Use a single resource bundles per locale for the error messages and the error message parameters

Getting ready

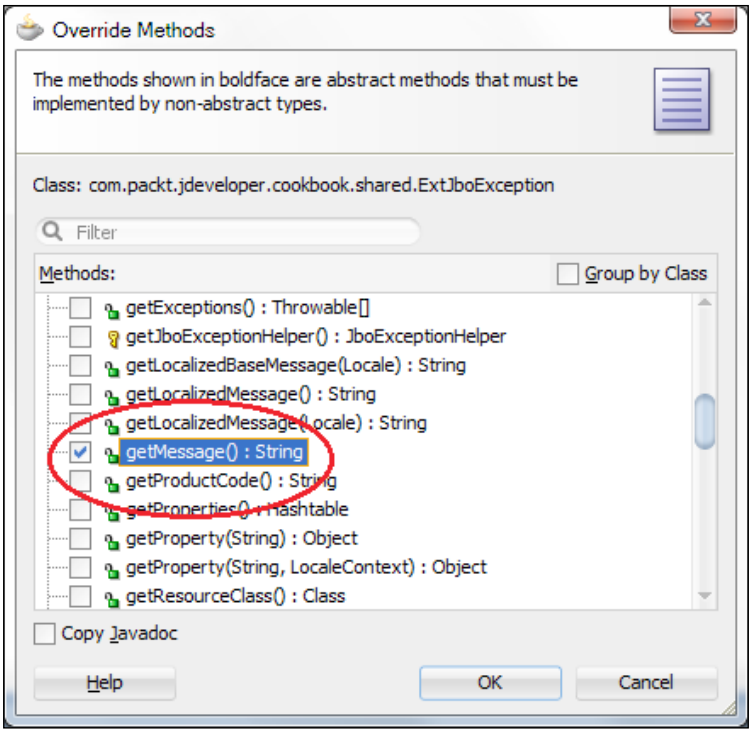
We will add the custom application exception class to the shared components project we created in the *Breaking up the application in multiple workspaces* recipe.

How to do it

1. Start by opening the shared components workspace.
2. Create a new class called `ExtJboException` by right-clicking on the Business Components project and selecting **New...**.
3. Then select **Java** under the **General** category and **Java Class** from list of **Items** on the right.
4. Click **OK** to display the **Create Java Class** dialog. Enter `ExtJboException` for the **Name**, `com.packt.jdeveloper.cookbook.shared.bc.exceptions` for the **Package** and `oracle.jbo.JboException` for the **Extends**.
5. Click **OK** to proceed with the creation of the custom exception class.
6. The next step would be to add two additional constructors to allow for the instantiation of the custom application exception using a standard error message code with optional error message parameters. This is what the additional constructors look like:

```
/**
 * Constructor to create an exception using a standard error code and
 * error message parameters
 * @param errorCode, the error message code
 * @param errorParameters, the error message parameters
 */
public ExtJboException(final String errorCode,
                       final Object[] errorParameters) {
   super(ResourceBundle.class, errorCode, errorParameters);
}

public ExtJboException(final String errorCode) {
   super(ResourceBundle.class, errorCode, null);
}
```

7. Now click on the **Override Methods...** icon on the top of the editor window and select to override the `getMessage()` method as it shown below.



8. Enter the following code for `getMessage()`:

```
public String getMessage() {
  // default message
  String errorMessage = "";
  try {
    // get access to the error messages bundle
    final ResourceBundle messagesBundle =
      ResourceBundle.getBundle(ERRORS_BUNDLE, Locale.getDefault());
    // construct the error message
    errorMessage =
        this.getErrorCode() + " - " + messagesBundle.getString(MESSAGE_PREFIX
+
                                                this.getErrorCode());

    // get access to the error message parameters bundle
    final ResourceBundle parametersBundle =
```

```
            ResourceBundle.getBundle(PARAMETERS_BUNDLE,
                                     Locale.getDefault());
          // loop for all parameters
          for (int i = 0; i < this.getErrorParameters().length; i++) {
            // get parameter value
            final String parameterValue =
              parametersBundle.getString(PARAMETER_PREFIX +
                                         (String)this.getErrorParameters()[i]);
            // replace parameter placeholder in the error message string
            errorMessage =
                errorMessage.replaceAll("\\{" + (i + 1) + "}", parameterValue);
          }
        } catch (Exception e) {
          // log the exception
          LOGGER.warning(e);
        }

        return errorMessage;
      }
```

9. Make sure that you also add the following constants:

```
private static final String ERRORS_BUNDLE =
    "com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages.
ErrorMessages";
private static final String PARAMETERS_BUNDLE =
    "com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages.ErrorParams";
private static final String MESSAGE_PREFIX = "message.";
private static final String PARAMETER_PREFIX = "parameter.";

private static final ADFLogger LOGGER =
    ADFLogger.createADFLogger(ExtJboException.class);
```

10. For testing purposes add the following `main()`:

```
// for testing purposes; remove or comment if not needed

public static void main(String[] args) {
  // throw a custom exception with error code "00001" and two parameters
  throw new ExtJboException("00001",
                new String[] { "FirstParameter", "SecondParameter" });
}
```

How it works

We created a custom exception at the BC level by overriding the `JboException` class. In order to use application-specific error codes, we introduced two new constructors. Both of them accept the error code as a parameter. One of them also accepts the message error parameters.

```
public ExtJboException(final String errorCode,
                       final Object[] errorParameters) {
    super(ResourceBundle.class, errorCode, errorParameters);
}
```

In our constructor we called the base class' constructor and passed the message error code and parameters to it.

Then we overrode the `getMessage()` in order to construct the exception message. In `getMessage()` first we got access to the error messages resource bundle by calling `ResourceBundle.getBundle()` as it shown below:

```
final ResourceBundle messagesBundle =                ResourceBundle.
getBundle(ERRORS_BUNDLE, Locale.getDefault());
```

This method accepts the name of the resource bundle and the locale. For the name of the resource bundle we passed the constant `ERRORS_BUNDLE`, which we defined as `com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages.ErrorMessages`. This is the `ErrorMessages.properties` file in the `com/packt/jdeveloper/cookbook/shared/bc/exceptions/messages` directory where we have added all of our messages. For the locale we used the default locale by calling `Locale.getDefault()`.

Then we proceeded by loading the error message from the bundle:

```
errorMessage = this.getErrorCode() + " - " + messagesBundle.
getString(MESSAGE_PREFIX + this.getErrorCode());
```

An error message definition in the messages resource bundle looks like this:

```
message.00001=This is an error message that accepts two parameters.
The first parameter is '{1}'. The second parameter is '{2}'
```

As you can see we have added the string prefix `message.` to the actual error message code. How you form the error message identifiers in the resource bundle is up to you. You could for example use a module identifier for each message and change the code in `getMessage()` appropriately. Also we used braces, i.e. `{1}, {2}` as placeholders for the actual message parameter values. Based on all these, we constructed the message identifier by adding the message prefix to the message error code as in:

```
MESSAGE_PREFIX + this.getErrorCode()
```

and called `getString()` on the `messagesBundle` to load it.

Then we proceeded with iterating the message parameters. In a similar fashion we called `getString()` on the parameters bundle the load the parameter values.

The parameter definitions in the parameters resource bundle look like this:

```
parameter.FirstParameter=Hello

parameter.SecondParameter=World
```

So we added the prefix `parameter.` to the actual parameter identifier before loading it from the bundle.

The last step was to replace the parameter placeholders in the error message with the actual parameter values. We did this by calling `replaceAll()` on the raw error message loaded as it shown below:

```
errorMessage = errorMessage.replaceAll("\\{" + (i + 1) + "}",
parameterValue);
```

For testing purposes we added a `main()` method to test our custom exception. You will `throw` the exception in your BC code similarly.

```
throw new ExtJboException("00001", // message code

new String[] { "FirstParameter", "SecondParameter" } // message
parameters

);
```

There's more

You can combine if you want the error message and the error message parameters bundles into a single resource bundle and change the `getMessage()` as needed to load both from the same resource bundle.

Bundled Exceptions

Exceptions by default are bundled at the transaction level for ADF BC-based web applications. By bundling of exceptions it is meant that all exceptions during attribute and entity validations are saved and reported once the validation process is complete. In other words, the validation will not stop on the first error, rather it will continue until the validation process completes and report all exceptions in a single error message. Bundled validation exceptions are implemented by wrapping exceptions as details of a new parent exception that contains them. For instance, if multiple attributes in a single entity object fail attribute validation, these multiple **ValidationException** objects are wrapped in a `RowValException`. This wrapping exception contains the row key of the row that has failed validation. At transaction commit time, if multiple rows do not successfully pass the validation performed during commit, then all of the `RowValException` objects will get wrapped in an enclosing `TxnValException` object. Then you can use the `getDetails()` method of the `JboException` base exception class to recursively process the bundled exceptions contained inside it.

Bundling of exceptions can be configured at the transaction level by calling `setBundledExceptionMode()` on the `oracle.jbo.Transaction`. This method accepts a boolean true/false indicating that bundled transactions will be used or not respectively.

Note that in the *Using a generic backing bean actions framework* recipe in this chapter we refactored the code in `getMessage()` to a reusable `BundleUtils.loadMessage()` method. Consequently we changed the `ExtJboException getMessage()` in that recipe to the following:

```
public String getMessage() {

  return BundleUtils.loadMessage(this.getErrorCode(),

                this.getErrorParameters());

}
```

See also

Handling security, session timeouts, exceptions and errors, chapter 9

Breaking up the application in multiple workspaces, chapter 1

Using a stored procedure helper class

In this recipe you will develop a helper class that you can use in your ADF BC application to call database stored procedures and functions. We will call the helper class `SQLProcedure` and we will make it general enough to support calling procedures and functions accepting and returning any number and types of parameters.

Getting ready

You will add the stored procedure helper to the shared components project that we developed in the *Breaking up the application in multiple workspaces* recipe. You will also need a database connection to the `HR` schema so you can test the helper class.

How to do it

1. Start by right-clicking on the shared components project in JDeveloper and selecting to create a Java class.
2. On the **Create Java Class** dialog enter `SQLProcedure` for the class name and `com.packt.jdeveloper.cookbook.shared.bc.database` for the class package. This class is the wrapper class for the database stored procedure or function.
3. Repeat this step to create another Java class called `SQLParameter`. This class represents a parameter passed to or returned by the database stored procedure or function.
4. We will start with this class because it really simple. Add the following declarations to the `SQLParameter`:

```
// the type of parameters
public static class TYPE {
   public static final int NULL = -1;
   public static final int IN = 1;
   public static final int OUT = 2;
   public static final int INOUT = 3;
   public static final int RETURN = 4;
}

// the parameter type (any of IN, OUT, INOUT, RETURN, NULL)
private int type = TYPE.NULL;
// the parameter number in the callable staement
private int parameterNumber = -1;
```

```
// the data associated with an IN, INOUT parameter
private Object inputData;
// the data associated with an OUT, INOUT, RETURN parameter
private Object outputData;
// the parameter return type
private int returnType = Types.NULL;
```

Also add the following constructor:

```
public SQLParameter(final int parameterNumber, final int
    type, final Object inputData, final int returnType) {
  this.parameterNumber = parameterNumber;
  this.type = type;
  this.inputData = inputData;
  this.returnType = returnType;
}
```

You will also need to provide the accessors for the class member variables.

Now add the following member variables to the `SQLProcedure` class:

```
// the transaction associated with procedure call
private DBTransaction transaction;
// a list of procedure parameters
private List<SQLParameter> parameters = new ArrayList<SQLParameter>();
// the callable statement
private CallableStatement statement = null;
// the statement string
private String statementString = null;
// the procedure parameter count
private int parameterCount = 0;
```

5. The class is constructed by specifying the stored procedure or function in the `procedure` parameter and supplying the associated Application Module transaction. This is what the constructor looks like:

```
public SQLProcedure(final String procedure,
                    final DBTransaction transaction) {
  this.statementString = procedure;
  this.transaction = transaction;
}
```

6. Next you will need to provide a number of setters, one for each parameter type. To set a parameter of type `IN` for instance, you will need to implement a `setIN()` method. This is what the `setIN()` method looks like:

```
public void setIN(final Object data) {
   ++parameterCount;
   parameters.add(new SQLParameter(parameterCount,
       SQLParameter.TYPE.IN, data, 0));
}
```

7. Similarly you will need to provide methods to set an `OUT`, `INOUT` and a `RETURN` parameter type. This is what the `setRETURN()` looks like:

```
public void setRETURN(final int returnType) {
   // do not allow multiple RETURNs
   SQLParameter returnParameter = hasRETURN();
   if (returnParameter != null) {
      // update RETURN parameter type
      returnParameter.setReturnType(returnType);
   } else {
   ++parameterCount;

      // check RETURN parameter count; should always be the 1
      if (parameterCount != 1) {
         shiftParameters();
      }

      // add RETURN parameter
      parameters.add(new SQLParameter(1, SQLParameter.TYPE.RETURN, null,
returnType));
   }
}
```

8. As you can see, we have taken special provision not allow for more than one `RETURN` type. We check to see if a `RETURN` type has already been specified by calling `hasRETURN()`.
9. Also we did some additional work to ensure that the `RETURN` parameter is always the first parameter in the callable statement. This is done by calling `shiftParameters()`.

10. To execute the stored procedure we call the `execute()` method, as expected. This is what `execute()` looks like:

```
public void execute() {
   try {
      // create statement for parameters
      createStatementFromParameters();
      // add parameters to statement
      addParameterValues();
      // execute callable statement
      statement.execute();
      // retrieve returned data
      retrieveData();
      // close callable statement
      statement.close();

   } catch (SQLException e) {
      throw new JboException(e);
   }
}
```

11. We need to also provide methods for retrieving the output data. For this purpose we implemented the `getOUT()` and `getRETURN()` methods. This is what `getRETURN()` looks like.

```
public Object getRETURN() {
   Object returnedData = null;
   for (SQLParameter parameter : parameters) {
      if (parameter.getType() == SQLParameter.TYPE.RETURN) {
         returnedData = parameter.getOutputData();
      }
   }
   return returnedData;
}
```

How it works

To call a database stored procedure or function, first instantiate a `SQLProcedure` by specifying the name of the database procedure or function that you want to call and the `oracle.jbo.server.DBTransaction` associated with the **Application Module**. Then specify the procedure's parameters by calling any of `setIN()`, `setOUT()`, `setINOUT()` and `setRETURN()` methods and execute it by calling `execute()`.

Finally, get any data returned by the procedure by calling `getOUT()` and/or `getRETURN()`. An example is shown below. In this case we call a database stored function called `TEST_FUNC_IN_OUT` in the `TEST_PKG` database package. The function accepts an `IN OUT` parameter indicating the employee's manager last name and returns the manager's first name in the same parameter and the manager's email with a `RETURN`.

```
        SQLProcedure funcInOut =
            new SQLProcedure("TEST_PKG.TEST_FUNC_IN_OUT", this.
getDBTransaction());
        funcInOut.setRETURN(Types.CHAR);
        funcInOut.setINOUT(managerLastName, Types.CHAR);
        funcInOut.execute();
        String managerFirstName = (String)funcInOut.getOUT(2);
        String email = (String)funcInOut.getRETURN();
```

Here are a few more details about the `SQLProcedure` wrapper class. The class uses an `ArrayList` called `parameters` to store the procedure parameters. It also keeps track of the number of parameters that you specify. Each time you call any of the `set…()` methods to set a parameter a new `SQLParameter` is created and added to the `parameters ArrayList`. As noted earlier, we take special care not to add a `RETURN` parameter more than once and that it is always the first parameter.

When we call `execute()` to execute the stored procedure, we first create the JDBC `CallableStatement` by calling the helper `createStatementFromParameters()`. The method iterates all the parameters specified, builds the statement string and finally the callable statement.

```
    private void createStatementFromParameters() {
        String paramString = "";
        for (SQLParameter parameter : parameters) {
            switch (parameter.getType()) {
            case SQLParameter.TYPE.IN:
            case SQLParameter.TYPE.OUT:
            case SQLParameter.TYPE.INOUT:
                // add parameter placeholder in statement
                paramString += (paramString.length() > 0) ? ",?" :
"?";
                break;
            case SQLParameter.TYPE.RETURN:
                // add return placeholder in statement
                statementString = "?:=" + statementString;
                break;
            }
        }
```

```
            // add complete parameter string to statement
            if (paramString.length() > 0) {
                statementString += "(" + paramString + ")";
            }

            // wrap statement with BEGIN/END
            statementString = "BEGIN " + statementString + "; END;";

            // create CallableStatement from statement
            statement = transaction.createCallableStatement(statementStri
    ng, 0);
        }
```

Then the parameter values are added by calling the helper `addParameterValues()`. In this method the parameters are iterated and for each parameter depending on the parameter type we call `setObject()` on the statement to set the parameter data – for `IN` and `IN OUT` parameters – and/or `registerOutParameter()` to register an output parameter – for `OUT`, `IN OUT` and `RETURN` types.

```
        private void addParameterValues() throws SQLException {
            for (SQLParameter parameter : parameters) {
                switch (parameter.getType()) {
                case SQLParameter.TYPE.IN:
                    statement.setObject(parameter.getParameterNumber(),
                                        parameter.getInputData());
                    break;
                case SQLParameter.TYPE.OUT:
                    statement.registerOutParameter(parameter.
    getParameterNumber(),
                                                   parameter.
    getReturnType());
                    break;
                case SQLParameter.TYPE.INOUT:
                    statement.setObject(parameter.getParameterNumber(),
                                        parameter.getInputData());
                    statement.registerOutParameter(parameter.
    getParameterNumber(),
                                                   parameter.
    getReturnType());
                    break;
                case SQLParameter.TYPE.RETURN:
                    statement.registerOutParameter(parameter.
    getParameterNumber(),
                                                   parameter.
    getReturnType());
```

```
                    break;
                }
            }
        }
```

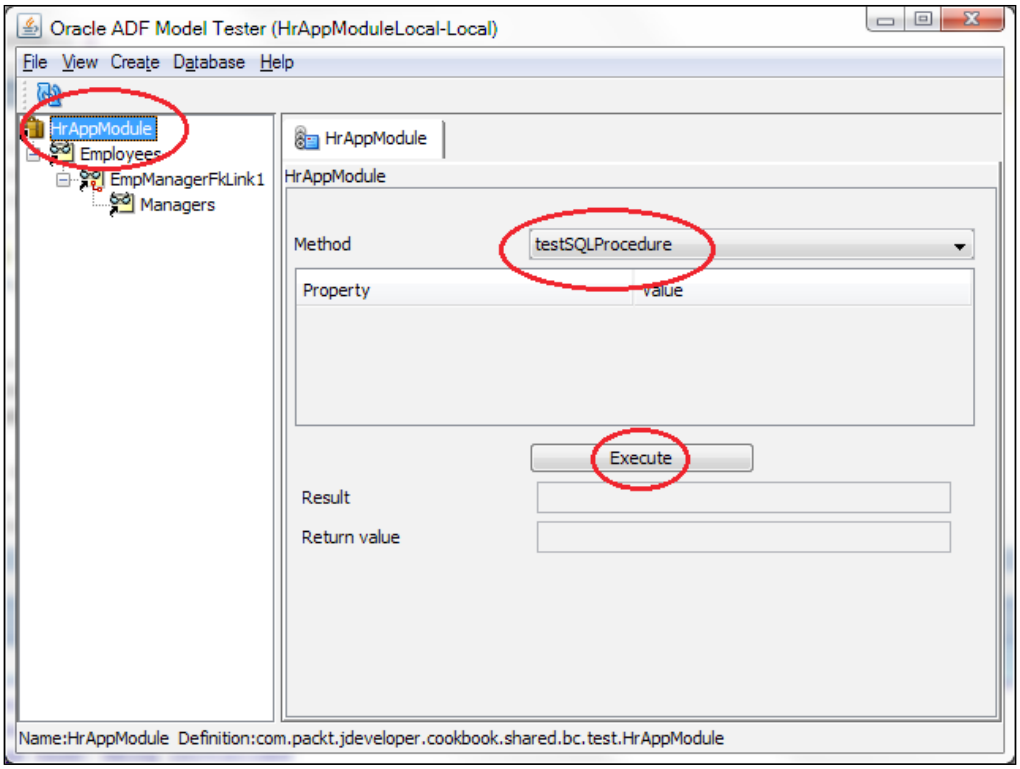
We call `execute()` on the JDBC `CallableStatement` to execute the procedure and the helper `retrieveData()` to retrieve the output parameter data. The `retrieveData()` iterates all parameters for any output type parameter calls `getObject()` on the `CallableStatement` to retrieve the data returned by the procedure. We save the output data in the `SQLParameter` itself by calling its `setOutputData()` method.

```
        private void retrieveData() throws SQLException {
            for (SQLParameter parameter : parameters) {
                if (SQLParameter.TYPE.INOUT == parameter.getType() ||
                    SQLParameter.TYPE.OUT == parameter.getType() ||
                    SQLParameter.TYPE.RETURN == parameter.getType()) {
                    parameter.setOutputData(statement.getObject(parameter.
    getParameterNumber()));
                }
            }
        }
```

The wrapper's public interface includes the methods `getOUT()` and `getRETURN()` to return the data to the caller. For the `getOUT()`, the parameter number must be specified in order to identify the parameter.

There's more

In the source code supplied with the book we have provided an `HrAppModuleImpl` Application Module implementation class and a method called `testSQLProcedure()` to test the `SQLProcedure` wrapper class. We added this method to the Application Module client interface so that we can call it through the **ADF Model Tester**. Using the **ADF Model Tester** utility you can then double-click on the `HrAppModule` to open the Application Module client interface methods, pick the `testSQLProcedure` method, and click on the **Execute** button to execute it.



The `HrAppModuleImpl` can be found in the `com.packt.jdeveloper.cookbook.shared.bc.test` package. In the `dbscripts` directory under it you will also find a database script that creates the `TEST_PKG` package. Run this script in the `HR` schema.

As stated earlier when we instantiate the `SQLProcedure` wrapper class we pass to it the `oracle.jbo.server.DBTransaction` associated with the Application Module. We then store it in the `transaction` member variable and use it in `createStatementFromParameters()` to create a `CallableStatement` by calling its `createCallableStatement()` method. This will only work as long as we construct and execute a `SQLProcedure` during the same user request. This is because ADF BC uses an **Application Module Pool** to get an Application Module instance for each request. Thus there is no guarantee at runtime that the same Application Module instance will be used across multiple requests.

Using ADFUtils/JSFUtils

In this recipe we will talk about how to incorporate and use the `ADFUtils` and `JSFUtils` utility classes in your ADF project. These are utility classes used at the **ViewController** level that encapsulate a number of lower level ADF and JSF calls into higher level methods. Integrating these classes in your ADF application early in the development process, and subsequently using them, will be of great help to you as a developer and contribute to the overall project's clarity and consistency. The `ADFUtils` and `JSFUtils` utility classes at the time of writing are not part of any official JDeveloper release. You will have to locate them, configure them, and possibly maintain them yourself for your project. There is no official version for them either.

Getting ready

We will be adding the `ADFUtils` and `JSFUtils` classes to the shared components ViewController project that we developed in the *Breaking up the application in multiple workspaces* recipe.

How to do it

1. To get the latest version of these classes, download and extract in your PC the latest version of the **Fusion Order Demo** application. This sample application can be found currently in the **Fusion Order Demo (FOD) - Sample ADF Application** page in the following address: `http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html`.
2. The latest version of **the** Fusion Order Demo application is R1PS3 at the time of this writing and is bundled in a zipped file. So go ahead download and extract the Fusion Order Demo application in your PC.
3. You should be able to locate the `ADFUtils` and `JSFUtils` classes in the location where you have extracted the **Fusion Order Demo** application.
4. If multiple versions of the same class are found, compare them and use the ones that are most up to date. For this recipe we have included in the source code the `ADFUtils` and `JSFUtils` found in the `SupplierModule\ViewController\src\oracle\fodemo\supplier\view\utils` directory.
5. Then copy them to a specific location in your shared ViewController components project. For this recipe we have copied them into the `SharedComponents\SharedViewController\src\com\packt\jdeveloper\cookbook\shared\view\util` directory.

6. Once copied, open both files with JDeveloper and change their `package` to to reflect their new location, in this case to `com.packt.jdeveloper.cookbook.shared.view.util`.

How it works

The public interfaces of both `ADFUtils` and `JSFUtils` define `static` methods, so you can call them directly without any class instantiations. Here are some of the methods commonly used.

Locating an iterator binding

To locate an iterator in the bindings use the `ADFUtils.findIterator()` method. The method accepts the bound iterator's identifier and returns an `oracle.adf.model.binding.DCIteratorBinding`. Here is an example:

```
DCIteratorBinding it = ADFUtils.findIterator("IteratorID");
```

Locating an operation binding

To locate an operation in the bindings use the `ADFUtils.findOperation()` method. This method accepts the bound operation's identifier and returns an `oracle.binding.OperationBinding`.

```
OperationBinding oper = ADFUtils.findOperation("OperationID");
```

Locating an attribute binding

Use `ADFUtils.findControlBinding()` to retrieve an attribute from the bindings. This method accepts the bound attribute's identifier and returns an `oracle.binding..AttributeBinding`.

```
AttributeBinding  attrib = ADFUtils.findControlBinding("AttributeId");
```

Getting and setting an attribute binding value

To get or set a bound attribute's value use the `ADFUtils.getBoundAttributeValue()` and `ADFUtils.setBoundAttributeValue()` methods respectively. Both of these methods accept the identifier of the attribute binding as an argument. The `getBoundAttributeValue()` method returns the bound attribute's data value as a `java.lang.Object`. The `setBoundAttributeValue()` accepts the value to set the attribute binding to as a `java.lang.Object`.

```
// get some bound attribute data
String someData = (String)ADFUtils.getBoundAttributeValue("Attribute
Id");
```

```
// set some bound attribute data
ADFUtils.setBoundAttributeValue("AttributeId", someData);
```

Getting the binding container

You can get the `oracle.adf.model.binding.DCBindingContainer` binding container by calling the `ADFUtils.getDCBindingContainer()` method.

```
DCBindingContainer bindings = ADFUtils.getDCBindingContainer();
```

Adding Faces messages

Use the `JSFUtils.addFacesInformationMessage()` and `JSFUtils.addFacesErrorMessage()` methods to display **Faces** information and error messages respectively. These methods accept the message to display as a `String` argument.

```
JSFUtils.addFacesInformationMessage("Information message");
JSFUtils.addFacesErrorMessage ("Error message");
```

Finding a component in the root View

To locate a UI component in the root View based on the component's identifier use the `JSFUtils.findComponentInRoot()` method. This method returns a `javax.faces.component.UIComponent` if one matching the specified component identifier is found.

```
UIComponent component = JSFUtils.findComponentInRoot("ComponentID");
```

Getting and setting Managed Bean values

Use the `JSFUtils.getManagedBeanValue()` and `JSFUtils.setManagedBeanValue()` methods to get and set a managed bean value respectively. These methods both accept the managed bean name. The `JSFUtils.getManagedBeanValue()` method returns the managed bean value as a `java.lang.Object`. The `JSFUtils.setManagedBeanValue()` method accepts as an argument the managed bean value to set as a `java.lang.Object`.

```
Object filePath = JSFUtils.getManagedBeanValue("bindings.FilePath.
inputValue");
JSFUtils.setManagedBeanValue("bindings.FilePath.inputValue", null);
```

Using page templates

In this recipe we will go over the steps required to create a JSF page template that you can use to create JSF pages throughout your application. It is very likely that for a large enterprise-scale application you will need to construct and use a number of different templates each serving a specific purpose. Using templates to construct the actual application JSF pages will ensure that pages throughout the application are consistent both in structure and content, and provide a familiar look and feel to the end user. You can follow the steps presented in this recipe to construct your page templates and adapt them as needed to fit your own requirements.

Getting ready

We will be adding the JSF template to the shared components ViewController project that we developed in the *Breaking up the application in multiple workspaces* recipe.

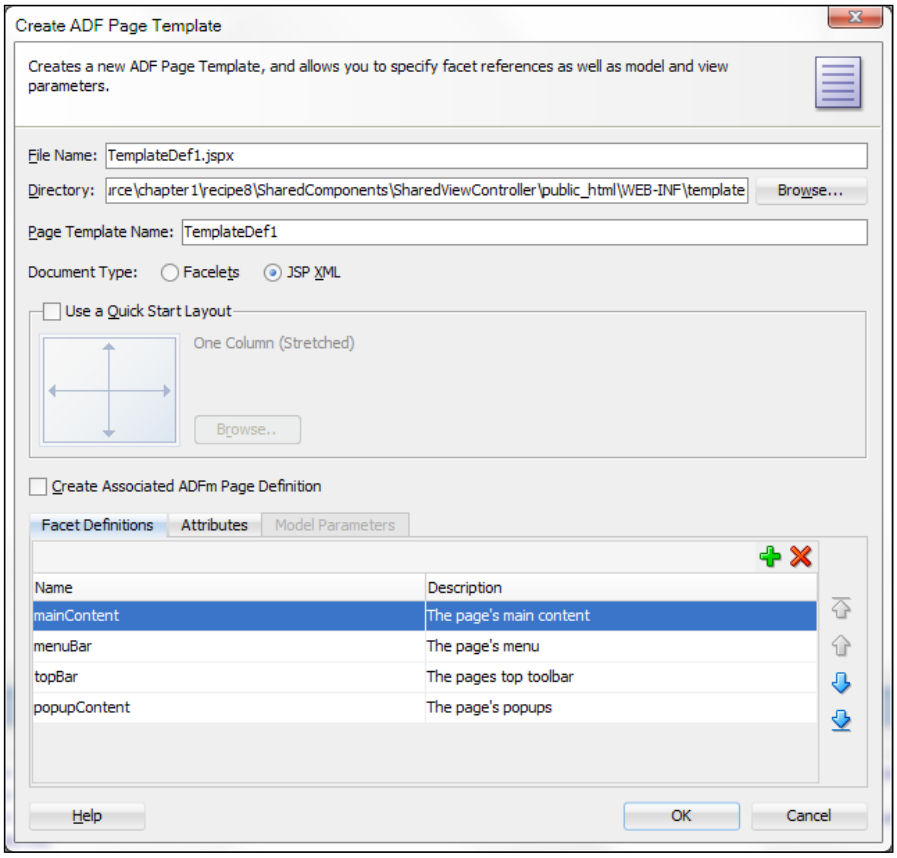
How to do it

1. Start by right-clicking on the ViewController project in the shared components workspace and selecting **New...**.
2. On the **New Gallery** dialog select **JSF/Facelets** from the list of **Categories** and **ADF Page Template** from the **Items** on the right.
3. Click **OK** to proceed. This will display the **Create ADF Page Template** dialog.
4. Enter the name of the template on the **Page Template Name**. Note that as you change the template name, the template **File Name** also changes to reflect the template name. For this recipe we will simply call the template `TemplateDef1`.
5. Now click on the **Browse...** button and select the directory where the template will be stored.
6. On the **Choose Directory** dialog navigate to the `public_html/WEB-INF` directory and click on the **Create new subdirectory** icon to create a new directory called `templates`.
7. For the **Document Type**, select **JSP XML**.
8. We will not be using any of the pre-defined templates, so uncheck the **Use a Quick Start Layout** checkbox.
9. Also since we will not be associating any data bindings to the template, uncheck the **Create Associated ADFm Page Definition** checkbox.

10. Next you will be adding the template facets. You do this by selecting the **Facet Definitions** tab and clicking on the **New** icon button. Enter the following facets:

Facet	Description
`mainContent`	This facet will be used for the page's main content.
`menuBar`	This facet will be used to define a menu on the top part of the page.
`topBar`	This facet will be used to define a toolbar under the page's menu.
`popupContent`	This facet will be used to define the page's popups.

11. Now click **OK** to proceed with the creation of the ADF page template.

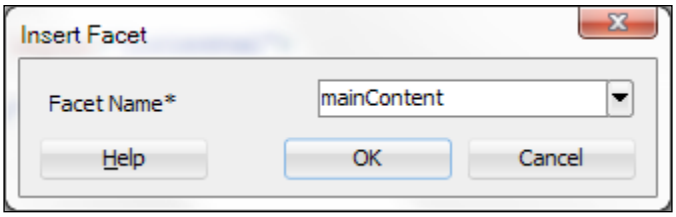


12. Once the template is created it is opened in the JDeveloper editor. If you followed the steps above, the template should look similar to this:

```
<af:pageTemplateDef var="attrs">
 <af:xmlContent>
  <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
   <display-name>TemplateDef1</display-name>
   <facet>
    <description>The page's main content</description>
    <facet-name>mainContent</facet-name>
   </facet>
   <facet>
    <description>The page's menu</description>
    <facet-name>menuBar</facet-name>
   </facet>
   <facet>
    <description>The page's top toolbar</description>
    <facet-name>topBar</facet-name>
   </facet>
   <facet>
    <description>The page's popups</description>
    <facet-name>popupContent</facet-name>
   </facet>
  </component>
 </af:xmlContent>
</af:pageTemplateDef>
```

13. As you can see, at this point the template contains only its definition in an `af:xmlContent` tag with no layout information whatsoever. We will proceed by adding the template's layout content.
14. From the **Layout** components in the **Component Palette**, grab a `Form` component and drop it into the template.
15. From the **Layout** container, grab a `Panel Stretch Layout` and drop it into the `Form` component. Remove the `top`, `bottom`, `start` and `end` facets.
16. From the **Layout** container, grab a `Panel Splitter` component and drop it on the `center` facet of the `Panel Stretch Layout`. Using **the Property Inspector** change the `Panel Splitter Orientation` to `vertical`. Also adjust the `SplitterPosition` to around 100.
17. Add your application logo by drag and dropping an `Image` component from the **General Controls** onto the `first` facet of the `Panel Splitter`. For this recipe, we have created a `public_html\images` directory and we copied a `logo.jpg` logo image there. We then specified `/images/logo.jpg` as image `Source` for the `Image` component.

18. Let's proceed by adding the main page's layout content. Drop a `Decorative Box` from the **Layout** components onto the `Panel Splitter's second` facet. We will not be using the `Decorative Box`'s `top` facet, so remove it.
19. OK, we are almost there! Drag a `Panel Stretch Layout` from the **Layout** components and drop it onto the `center` facet of the `Decorative Box`. Remove the `start` and `end` facets since we will not using them.
20. Drag a `Facet Ref` component from the **Layout components** and drop it onto the `center` facet of the `Panel Stretch Layout`. On the **Insert Facet Ref** dialog select the `mainContent` facet that you added during the template creation.



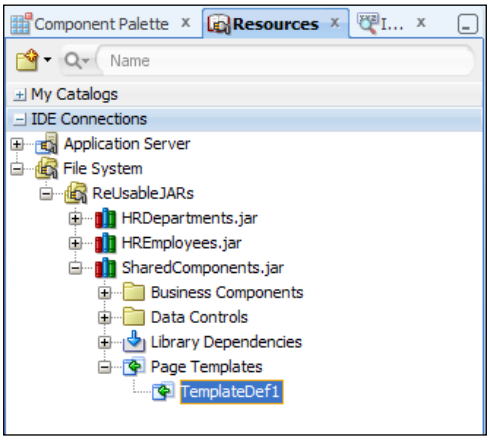
21. Finally add the following code the `Panel Stretch Layout topBar` facet:

```
<f:facet name="top">
  <af:panelGroupLayout id="pt_pgl5" layout="vertical">
  <af:facetRef facetName="popupContent"/>
  <af:menuBar id="pt_mb1">
  <af:facetRef facetName="menuBar"/>
  </af:menuBar>
  <af:panelGroupLayout id="pt_pgl2" layout="horizontal">
  <af:toolbar id="pt_t2">
  <af:facetRef facetName="topBar"/>
   </af:toolbar>
  </af:panelGroupLayout>
  </af:panelGroupLayout>
</f:facet>
```

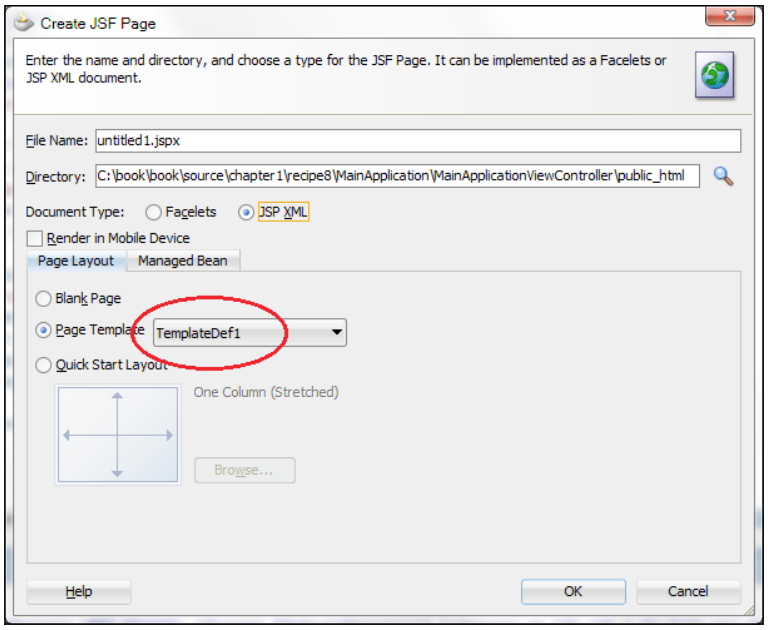
How it works

When the template was created there was no layout information in it, so we had to add it ourselves. We did this by using a variety of layout components to arrange the contained UI. Also, notice the usage of the `af:facetRef` component. It is used to reference a template facet in the specific place within the layout content. The facet then is available to you when you create a JSF page from the template. This will become obvious when we generate a JSF page from the template.

So, how do you use the JSF page template? Since we have created the template in a shared components project, we will first need to deploy the project to an **ADF Library JAR**. Then we will be able to use it from other consuming projects. How we do this was explained in the *Breaking up the application in multiple workspaces* recipe earlier in this chapter. When you do so, the template will be visible to all consuming projects as it is shown below.



Once the ADF Library JAR containing the template is added to the consuming project, you should be able to see and select the template when you create a new JSF page.



A JSF page created from this template will look like this:

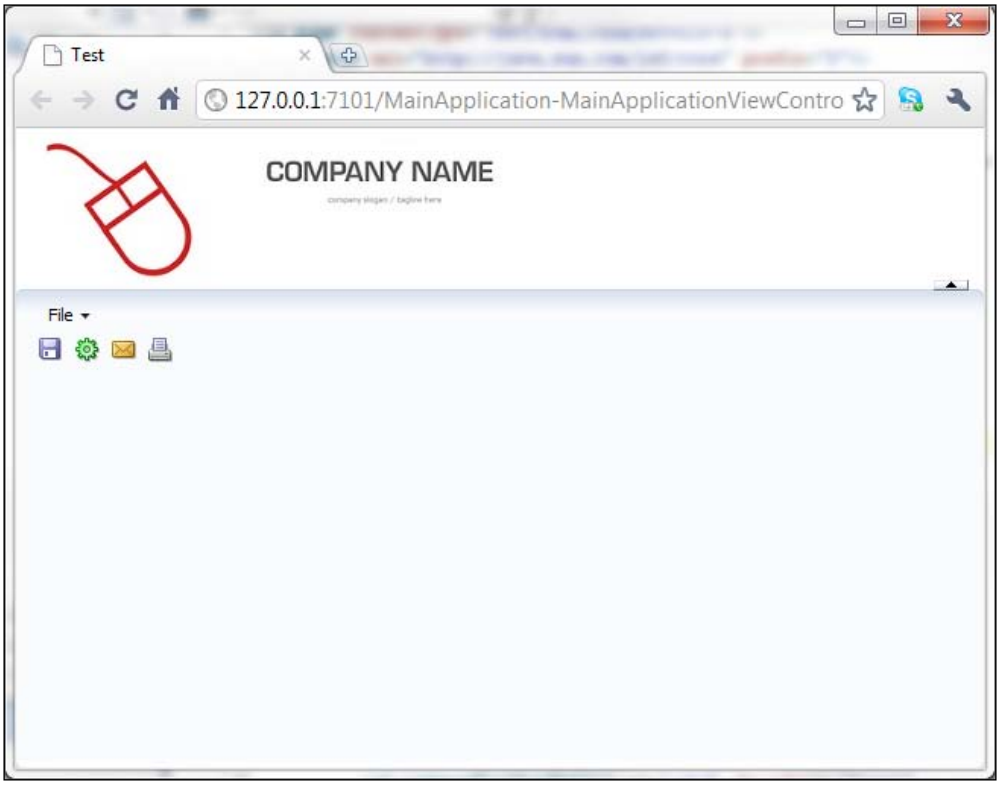
```
<f:view>
  <af:document id="d1" title="Test">
    <af:pageTemplate viewId="/WEB-INF/templates/TemplateDef1.jspx"
id="pt1">
      <f:facet name="mainContent"/>
      <f:facet name="menuBar"/>
      <f:facet name="topBar"/>
      <f:facet name="bottomBar"/>
      <f:facet name="popupContent"/>
    </af:pageTemplate>
  </af:document>
</f:view>
```

You can see in the page listing that the page references the template via the `af:pageTemplate` tag. The template facets that you had defined are available so you can enter the page-specific UI content. After adding an `af:menuBar` to the `menuBar` facet and some `af:commandToolbarButton` components to the `topBar` facet, the JSF page should look similar like this:

```
<f:view>
  <af:document id="d1" title="Test">
    <af:pageTemplate viewId="/WEB-INF/templates/TemplateDef1.jspx"
id="pt1">
      <f:facet name="mainContent"/>
      <f:facet name="menuBar">
        <af:menuBar id="mb1">
          <af:menu text="File" id="m1">
            <af:commandMenuItem text="Save" id="cmi1"
                                icon="/images/filesave.png"/>
            <af:commandMenuItem text="Action" id="cmi2"
                                icon="/images/action.png"/>
            <af:commandMenuItem text="Mail" id="cmi3"
                                icon="/images/envelope.png"/>
            <af:commandMenuItem text="Print" id="cmi4"
                                icon="/images/print.png"/>
          </af:menu>
        </af:menuBar>
      </f:facet>
      <f:facet name="topBar">
        <af:group id="g1">
          <af:commandToolbarButton id="ctb1" shortDesc="Save"
                                   icon="/images/filesave.png"/>
```

```
            <af:commandToolbarButton id="ctb2" shortDesc="Action"
                                     icon="/images/action.png"/>
            <af:commandToolbarButton id="ctb3" shortDesc="Mail"
                                     icon="/images/envelope.png"/>
            <af:commandToolbarButton id="ctb4" shortDesc="Print"
                                     icon="/images/print.png"/>
          </af:group>
        </f:facet>
        <f:facet name="popupContent"/>
      </af:pageTemplate>
    </af:document>
  </f:view>
```

Running the page in JDeveloper as is will produce the following:



There's more

Although adding a `Form` component to a template is not a recommended practice, this is not a problem for the template created in this recipe since we will not be using it for the creation of page fragments. Using a template that contains a `Form` component to create page fragments will result in a problem when a consuming page already contains a `Form` component itself.

Using a generic backing bean actions framework

In this recipe we will create a base backing bean class that we will use to encapsulate common functionality for common JSF page actions such as committing and rolling back data, creating new records, deleting records and so on. Creating and using such a generic backing bean actions framework will guarantee that you provide consistent functionality throughout the application and encapsulate common functionality at a base class level. This class is not intended to be used as another utility class. Any new helper methods that were developed to demonstrate the case were added to the `ADFUtils` utility class discussed earlier in this chapter.

Getting ready

We will be adding the generic backing bean actions framework to the shared components ViewController project that we developed in the *Breaking up the application in multiple workspaces* recipe.

How to do it

1. Right-click on the shared **ViewController** project and select **New...**.
2. On the **New Gallery** dialog select **Java** under the **General** category and **Java Class** from the list of items on the right.
3. On the **Create Java Class** dialog enter `CommonActions` for the class name and `com.packt.jdeveloper.cookbook.shared.view.actions` for the class's package.
4. Let's go ahead and add these methods to provide consistent commit functionality:

```
public void commit(ActionEvent actionEvent) {
  if (ADFUtils.hasChanges()) {
    // allow derived beans to handle before commit actions
    onBeforeCommit(actionEvent);
```

```
            // allow derived beans to handle commit actions
            onCommit(actionEvent);
            // allow derived beans to handle after commit actions
            onAfterCommit(actionEvent);
        } else {
            // display "No changes to commit" message
            JSFUtils.addFacesInformationMessage(BundleUtils.loadMessage("00002"));
        }
    }

    protected void onBeforeCommit(ActionEvent actionEvent) {
    }

    /**
    protected void onCommit(ActionEvent actionEvent) {
        // execute commit
        ADFUtils.execOperation(Operations.COMMIT);
    }

    protected void onAfterCommit(ActionEvent actionEvent) {
        // display "Changes were committed successfully" message
        JSFUtils.addFacesInformationMessage(BundleUtils.loadMessage("00003"));
    }
```

5. We have also added similar methods for consistent rollback behaviour. To provide uniform record creation/insertion functionality, let's add these methods:

```
    public void create(ActionEvent actionEvent) {
        if (hasChanges()) {
            onCreatePendingChanges(actionEvent);
        } else {
            onContinueCreate(actionEvent);
        }
    }

    protected void onBeforeCreate(ActionEvent actionEvent) {
        // commit before creating a new record
        ADFUtils.execOperation(Operations.COMMIT);
    }

    public void onCreate(ActionEvent actionEvent) {
        execOperation(Operations.INSERT);
```

```
}

protected void onAfterCreate(ActionEvent actionEvent) {
}

public void onCreatePendingChanges(ActionEvent actionEvent) {
   ADFUtils.showPopup("CreatePendingChanges");
}

public void onContinueCreate(ActionEvent actionEvent) {
   onBeforeCreate(actionEvent);
   onCreate(actionEvent);
   onAfterCreate(actionEvent);
}
```

6. Similar methods were added for consistent record deletion behaviour. In this case, we have added functionality to show a delete confirmation popup.

How it works

To provide consistent functionality at the JSF page actions level, we have implemented the `commit()`, `rollback()`, `create()` and `remove()` methods. Derived backing beans should handle these actions by simply delegating to this base class via calls to `super.commit()`, `super.rollback()` and so on. The base class `commit()` implementation first calls the helper `ADFUtils.hasChanges()` to determine whether there are transaction changes. If there are, then the `onBeforeCommit()` is called to allow derived backing beans to perform any before commit handling. Commit processing continues by calling `onCommit()`. Again, derived backing beans can override this method to provide specialized commit processing. The base class implementation of `onCommit()` calls the helper `ADFUtils.execOperation()` to execute the `Operations.COMMIT` bound operation. The commit processing finishes by calling the `onAfterCommit()`. Derived backing beans can override this method to provide any after commit processing. The default base class implementation displays a **Changes were committed successfully** message on the screen.

The generic functionality for a new record creation is implemented in the `create()` method. Derived backing beans should delegate to this method for default record creation processing by calling `super.create()`. In `create()` we first check to see if we have any changes to the existing transaction. If we do, we will inform the user by displaying a message dialog. We do this in the `onCreatePendingChanges()` method. The default implementation of this method displays the `CreatePendingChanges` confirmation popup. The derived backing bean can override this method to handle this event in a different manner. If the user chooses to go ahead with the record creation, the `onContinueCreate()` is called. This method calls `onBeforeCreate()` to handle pre-create functionality. The default implementation commits the current record by calling `ADFUtils.execOperation(Operations.COMMIT)`.

Record creation continues with calling `onCreate()`. The default implementation of this method creates and inserts the new record by calling `ADFUtils.execOperation(Operations.INSERT)`. Finally `onAfterCreate()` is called to handle any creation post processing.

The generic rollback and record deletion functionality is similar. For the default delete processing, a popup is displayed asking the user to confirm whether the record should be deleted or not. The record is deleted only after the user's confirmation.

There's more

Note that this framework uses a number of popups in order to confirm certain user choices. In order to avoid adding these popups to all JSF pages and provide reusable popups for all of your JSF pages, these popups should be added once to your JSF page template. In order to support this generic functionality, additional plumbing code will need to be added to the actions framework. We will talk at length about it in the Using page templates for popup reuse recipe in chapter 7.

See also

Using page templates for popup reuse, chapter 7.

Breaking up the application in multiple workspaces, chapter 1.

2

Dealing with Basics: Entity Objects

In this chapter, we will cover:

- Using a custom property to populate a sequence attribute
- Overriding doDML() to populate an attribute with a gapless sequence
- Creating and applying property sets
- Using getPostedAttribute() to determine the posted attribute's value
- Overriding remove() to delete associated children entities
- Overriding remove() to delete a parent entity in an association
- Using a method validator based on a View Object accessor
- Using Groovy expressions to resolve validation error message tokens
- Using doDML() to enforce a detail record for a new master record

Introduction

Entity objects are the basic building blocks in the chain of Business Components. They represent a single row of data and they encapsulate the business model, data, rules and persistence behavior. Usually they map to database objects, most commonly to database tables and views. Entity object definitions are stored in XML metadata files. These files are maintained automatically by JDeveloper and the ADF framework and they should not be edited by hand. The default Entity object implementation is provided by the ADF framework class `oracle.jbo.server.EnityImpl`. For large scale projects you want to create your own custom Entity framework class as we have demonstrated in *Setting up BC base classes* in chapter 1.

Likewise, it is not uncommon in large scale projects that custom implementations are provided for the Entity object methods `doDML()`, `create()` and `remove()`. The recipes in this chapter demonstrate, among others, some of the custom functionality that can be implemented in these methods. Furthermore other topics from generic programming using Custom Properties and Property Sets, custom validators, Entity Associations, populating sequence attributes, and more, are covered throughout the chapter.

Using a custom property to populate a sequence attribute

In this recipe we will go over a generic programming technique that you can use to assign database sequence values to specific **Entity Object** attributes. Generic functionality is achieved by using **Custom Properties**. Custom Properties allow you to define custom metadata that can be accessed by the **ADF Business Components** at runtime.

Getting ready

We will add this generic functionality to the custom Entity framework class. This class was created back in the *Setting up BC base classes* recipe in chapter 1. The custom framework classes in this case reside in the shared components workspace. This workspace was created in the *Breaking up the application in multiple workspaces* recipe. You will need to create a database connection to the `HR` schema if you are planning to run the recipe's test case. You can do this either by creating the database connection in the **Resource Palette** and drag and dropping it to the **Application Resources** > **Connections,** or by creating it directly in the **Application Resources** > **Connections**.

How to do it

1. Start by opening the shared components workspace in JDeveloper. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom Entity framework class in the shared BC project and open it in the editor.
3. Click on the **Override Methods...** icon on the toolbar (the green left arrow) to bring up the **Override Methods** dialog.
4. From the list of methods that are presented, select the **create()** method and click **OK**. JDeveloper will go ahead and insert a `create()` method to the body of your custom Entity class.

5. Add the following code to the `create()` immediately after the call to `super.create()`:

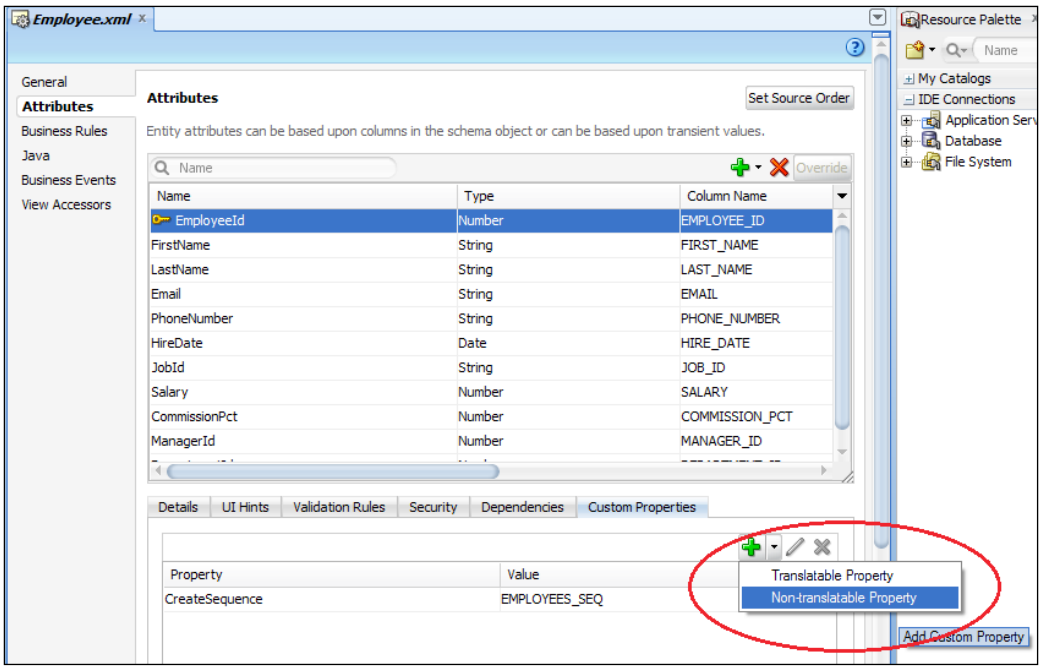
```
// iterate all entity attributes
for (AttributeDef atrbDef : this.getEntityDef().getAttributeDefs()) {
  // check for a custom property called CREATESEQ_PROPERTY
  String sequenceName =
    (String)atrbDef.getProperty(CREATESEQ_PROPERTY);
  if (sequenceName != null) {
    // create the sequence based on the custom property sequence name
    SequenceImpl sequence =
      new SequenceImpl(sequenceName, this.getDBTransaction());
    // populate the attribute with the next sequence number
    this.populateAttributeAsChanged(atrbDef.getIndex(),
                                    sequence.getSequenceNumber());
  }
}
```

How it works

In the code listed above we have overridden the `create()` method for the custom Entity framework class. This method is called by the ADF framework each time a new Entity object is constructed. We call `super.create()` to allow the framework processing, and then we retrieve the Entity's attribute definitions by calling `getEntityDef().getAttributeDefs()`. We then iterate over them, calling `getProperty()` for each attribute definition. `getProperty()` accepts the name of a custom property defined for the specific attribute. In our case the custom property is called `CreateSequence` and it is indicated by the constant definition `CREATESEQ_PROPERTY`. It represents the name of the database sequence used to assign values to the particular attribute. Then we instantiate a `SequenceImpl` object using the database sequence name retrieved from the custom property. Finally, the attribute is populated with the value returned from the sequence – via the `getSequenceNumber()` call – by calling `populateAttributeAsChanged()`. This method will populate the attribute without marking the entity as changed. Since all of the Entity objects are derived from the custom Entity framework class, all object creations will go through this `create()` implementation.

There's more

So how do you use this technique to populate your sequence attributes? First you must deploy the shared components workspace into an **ADF Library JAR** and add the library to the project where it will be used. Then, you will need to add the `CreateSequence` custom property to the specific attributes of your Entity objects that must be populated by a database sequence. To add a custom property to an Entity object attribute, select the specific attribute in the entity **Attributes** tab and click on the arrow next to the **Add Custom Property** icon (the green plus sign) in the **Custom Properties** tab below. From the context menu select **Non-translatable Property.**



Click on the **Property** field and enter **CreateSequence**. For the **Value** enter the database sequence name that will be used to assign values to the specific attribute. For the **Employee** Entity object example above, we will use the **EMPLOYEES_SEQ** database sequence to assign values to the **EmployeeId** attribute.

Note that for testing purposes, we have created in the `HREmployees` workspace an `Employee` Entity object and added the `CreateSequence` custom property to its `EmployeeId` attribute. To test the recipe, you can run the `EmployeeAppModule` **Application Module**.

See also

Breaking up the application in multiple workspaces, chapter 1

Setting up *BC base classes, chapter 1*

Overriding doDML() to populate an attribute with a gapless sequence, chapter 2

Creating and applying property sets, chapter 2

Overriding doDML() to populate an attribute with a gapless sequence

In this recipe we will go over a generic programming technique that you can use to assign gapless database sequence values. A gapless sequence will produce values with no gaps in between them. The difference between this technique and the one presented in the *Using a custom property to populate a sequence attribute* recipe, is that the sequence values are assigned during the transaction commit cycle instead of the component creation.

Getting ready

We will add this generic functionality to the custom **Entity** framework class that we created in the *Setting up BC base classes* recipe in chapter 1. The custom framework classes in this case reside in the shared components workspace. You will need access to the `HR` database schema to run the recipe's test case.

How to do it...

1. Start by opening the shared components workspace in JDeveloper. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom Entity framework class in the shared BC project and open it in the editor.
3. Click on the **Override Methods...** icon on the toolbar (the green left arrow ) to bring up the **Override Methods** dialog.
4. From the list of methods that are presented select the **doDML()** method and click **OK**. JDeveloper will go ahead and insert a `doDML()` method to the body of your custom Entity class.
5. Add the following code to the `doDML()` before the call to `super.doDML()`:

```
// check for insert operation
```

```
if (DML_INSERT == operation) {
   // iterate all entity attributes
   for (AttributeDef atrbDef :
       this.getEntityDef().getAttributeDefs()) {
      // check for a custom property called COMMITSEQ_PROPERTY
      String sequenceName =
         (String)atrbDef.getProperty(COMMITSEQ_PROPERTY);
      if (sequenceName != null) {
         // create the sequence based on the custom property sequence name
         SequenceImpl sequence =
            new SequenceImpl(sequenceName, this.getDBTransaction());
         // populate the attribute with the next sequence number
         this.populateAttributeAsChanged(atrbDef.getIndex(),
                                         sequence.getSequenceNumber());
      }
   }
}
```

How it works

If you examine the code presented in this recipe you will see that it looks similar to the code presented in the *Using a custom property to populate a sequence attribute* recipe. The difference is that this code executes during the transaction commit phase. During this phase, the ADF framework calls the Entity's `doDML()` method. In our overridden `doDML()` we first check for a `DML_INSERT` operation flag. This would be the case when inserting a new record to the database. We then iterate the Entity's attribute definitions looking for a custom property identified by the constant `COMMITSEQ_PROPERTY`. Based on the property's value we create a sequence object and get the next sequence value by calling `getSequenceNumber()`. Finally, we assign the sequence value to the specific attribute by calling `populateAttributeAsChanged()`. Assigning a sequence value during the commit phase does not allow the user to change the sequence value. This will produce gapless sequence values. Of course to guarantee that there are no final gaps to the sequence values, no deletes should be allowed. Since all of the Entity objects are derived from the custom Entity framework class, all object commits will go through this `doDML()` implementation.

To use this technique, first you will need to re-deploy the shared components project. Then add the `CommitSequence` custom property as needed to the specific attributes of your Entity objects. We explained how to do this in the *Using a custom property to populate a sequence attribute* recipe.

There's more

`doDML()` is called by the ADF framework during a transaction commit operation. It is called for every Entity object in the transaction's pending changes list. For an Entity-based View object, this means that it will be called for every row in the row set. The method accepts an operation flag that indicates the DML operation. It can be any of the `DML_INSERT`, `DML_UPDATE` or `DML_DELETE` to indicate an insert, update or delete operation on the specific Entity. Data is posted to the database once `super.doDML()` is called. This means that any exceptions thrown before calling `super.doDML()` will result in no posted data. Once the data is posted to the database, queries or stored procedures that rely upon the posted data should be coded in the overridden Application Module `beforeCommit()` method. `beforeCommit()` is also available at the Entity object level. In this case it is called by the framework for each Entity in the transaction's pending changes list.

Note that for testing purposes, we have created in the `HRDepartments` workspace a `Department` Entity object and added the `CommitSequence` custom property to its `DepartmentId` attribute. The value of the `CommitSequence` property was set to `DEPARTMENTS_SEQ`, the database sequence that is used to assign values to the `DepartmentId` attribute. To test the recipe, run the `DepartmentAppModule` **Application Module** on the ADF Model Tester.

See also

Breaking up the application in multiple workspaces, chapter 1

Setting up *BC base classes*, chapter 1

Using a custom property to populate a sequence attribute, chapter 2

Creating and applying property sets, chapter 2

Creating and applying property sets

In the *Using a custom property to populate a sequence attribute* and *Overriding doDML() to populate an attribute with a gapless sequence* recipes, we introduced custom properties for generic **ADF Business Components** programming. In this recipe we will present a technique to organize your custom properties in reusable **Property Sets**.

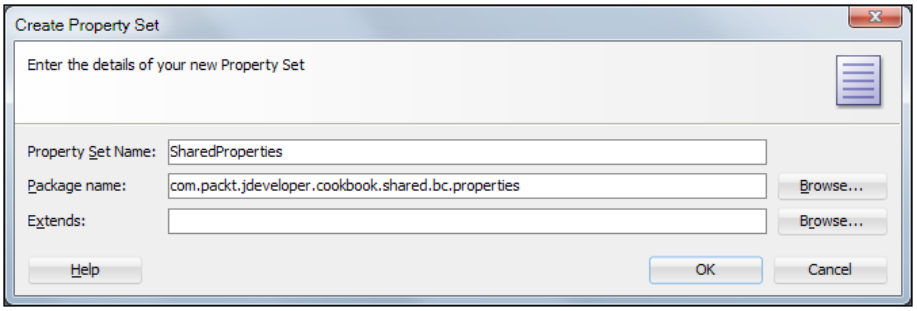
Getting ready

We will create a property set in the shared components workspace. I suggest that you go over the *Using a custom property to populate a sequence attribute* and *Overriding doDML()*

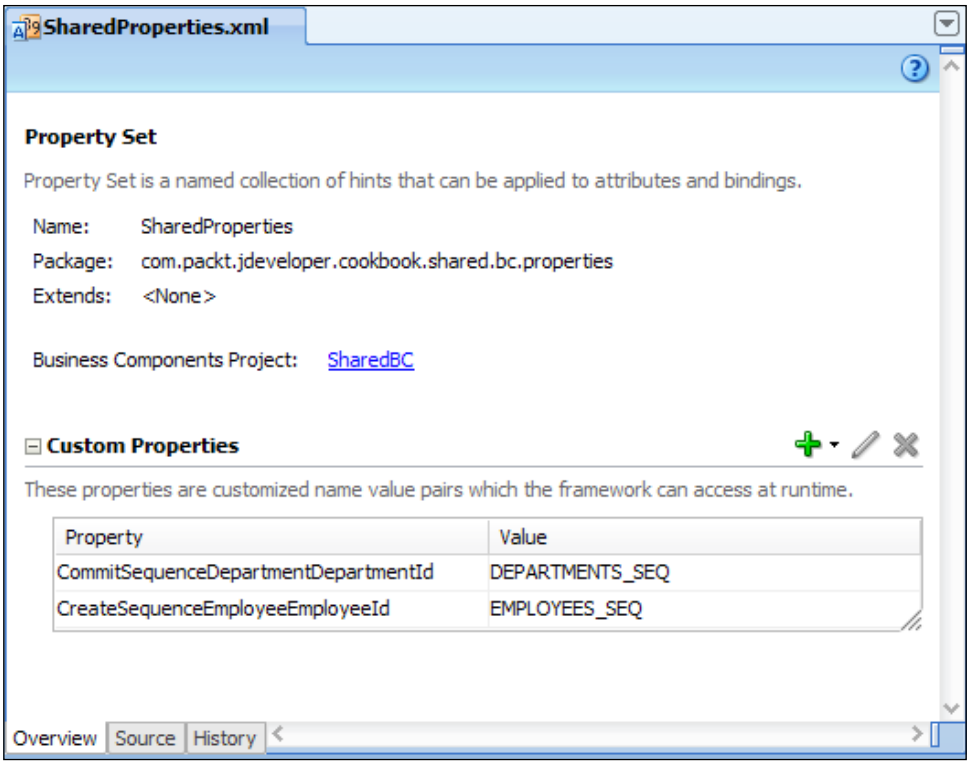
to populate an attribute with a gapless sequence recipes in this chapter before continuing with this recipe. To run the recipe's test cases, you will need access to the `HR` schema in the database.

How to do it

1. Start by opening the shared components workspace. If needed, follow the steps in the referenced recipe to create it.
2. Right-click on the shared BC project and select **New...**.
3. On the **New Gallery** dialog select **ADF Business Components** under the **Business Tier** node and **Property Set** from the **Items** on the right.
4. Click **OK** to proceed. This will open the **Create Property Set** dialog.



5. Enter the property set name and package in the appropriate fields. For this recipe we will call it **SharedProperties** and use the **com.packt.jdeveloper.cookbook.shared.bc.properties** package. Click **OK** to continue.
6. JDeveloper will create and open the `SharedProperties` property set.
7. To add a custom property to the property set click on the **Add Custom Property** button (the green plus sign icon).
8. Go ahead and add two non-translatable properties called **CommitSequenceDepartmentDepartmentId** and **CreateSequenceEmployeeEmployeeId**. Set their values to **DEPARTMENTS_SEQ** and **EMPLOYEES_SEQ** respectively. Your property set should look like this:



9. Next you need to change the `create()` method in the custom Entity framework class so that the custom property is now constructed like this:

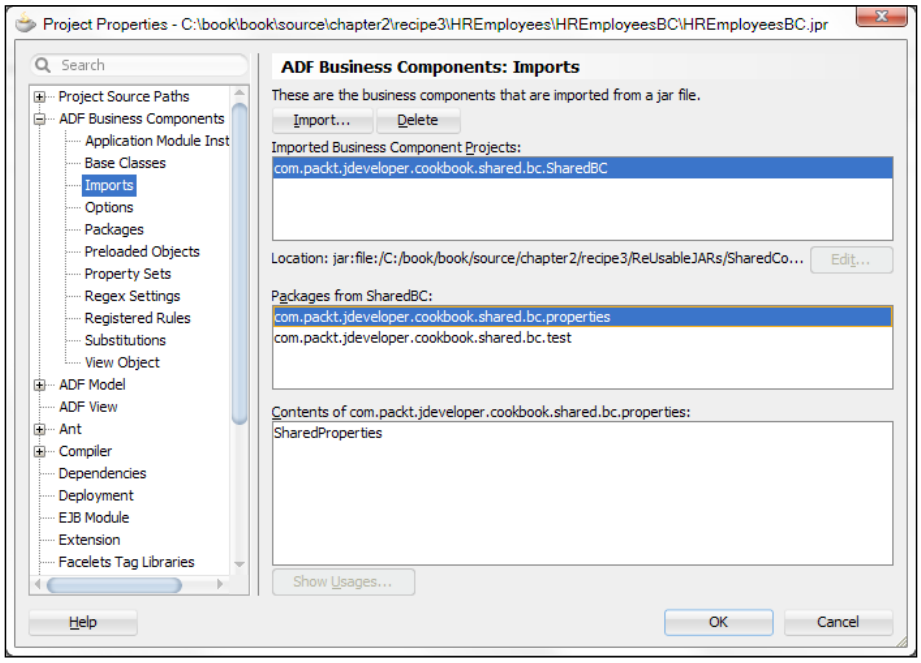
```
// construct the custom property name from the entity name and attribute
String propertyName = CREATESEQ_PROPERTY +
        getEntityDef().getName() + atrbDef.getName();

// check for a custom property called CREATESEQ_PROPERTY
String sequenceName =
        (String)atrbDef.getProperty(propertyName);
```

Similarly change the `doDML()` method in the custom Entity framework class so that the custom property is also constructed as shown below:

```
// construct the custom property name from the entity name and attribute
String propertyName = COMMITSEQ_PROPERTY +
        getEntityDef().getName() + atrbDef.getName();

// check for a custom property called COMMITSEQ_PROPERTY
String sequenceName =
```

```
(String)atrbDef.getProperty(propertyName);
```

10. Redeploy the shared components workspace into an **ADF Library JAR**.
11. Open the `HREmployees` workspace and double-click on the `Employees` Business Components project to bring up the **Project Properties** dialog.
12. Select **Imports** under the **ADF Business Components** node and click on the **Import...** button on the right.
13. On the **Import Business Components XML File** dialog browse for the shared components ADF Library JAR file in the `ReUsableJARs` directory. Select it and click **Open**.
14. You should see the imported BC project under the **Imported Business Component Projects** along with the imported packages and package contents. Click **OK** to continue with importing the business components.



15. Double-click on the **Employee** Entity object and go to the **Attributes** tab.
16. Click on the **Details** tab and from the **Property Set** choice list select the imported property set.
17. Repeat steps 12-17 for the `HRDepartments` workspace and apply the property set to the `DepartmentId` attribute of the `Department` Entity object.

How it works

Property sets are a way to gather all of your custom properties together into logical collections. Instead of applying each custom property separately to a BC object or to any of its attributes, custom properties that are defined in these collections can be applied at once on them. Property sets can be applied to Entity objects and their attributes, View objects and their attributes, and Application Modules. You access custom properties programmatically as indicated above by calling `AttributeDef.getProperty()` for properties applied to attributes, `EntityDefImpl.getProperty()` for properties applied to Entity objects, `ViewDefImpl.getProperty()` for properties applied to View objects, and so on.

How you organize your custom properties into property sets is up to you. In this recipe for example, we used a single property set called `SharedProperties`, which we defined in the shared components ADF library. In this way we kept all custom properties used by the application in a single container. For this to work we had to devise a way to differentiate among them. The algorithm that we used was to combine the property name with the BC object name and the attribute name that the property applies to. So we came up with properties called `CommitSequenceDepartmentDepartmentId` and `CreateSequenceEmployeeEmployeeId`.

Finally, note how we imported the property set from the shared components workspace into the relevant BC projects using the **Import Business Components** facility of the BC **Project Properties** dialog.

There's more

To test the recipe, you can run the `EmployeeAppModule` and `DepartmentAppModule` Application Modules in the `HREmployees` and `HRDepartments` workspaces respectively.

Note that you can override any of the properties defined in a property set by explicitly adding the same property to the Business Component object or to any of its attributes.

Also note that property sets can be applied onto Entity objects, View objects and Application Modules by clicking on the **Edit property set selection** button (the pen icon) on the Business Component object definition **General** tab. On the same tab you can add custom properties to the Business Component object by clicking on the **Add Custom Property** button (the green plus sign icon).

See also

Breaking up the application in multiple workspaces, chapter 1

Setting up *BC base classes*, chapter 1

Using a custom property to populate a sequence attribute, chapter 2

Overriding doDML() to populate an attribute with a gapless sequence in this, chapter 2

Using getPostedAttribute() to determine the posted attribute's value

There are times when you need to get the original database value of an **Entity** object attribute. This would be the case for instance when you want to compare the attribute's current value to the original database value. In this recipe we will illustrate how to do this by utilizing the `getPostedAttribute()` method.

Getting ready

We will be working on the shared components workspace. We will add a helper method to the custom Entity framework class.

How to do it

1. Start by opening the shared components workspace. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom Entity framework class and open it into the source editor.
3. Add the following code to the custom Entity framework class:

```
/**
 * Check if attribute's value differs from its posted value
 * @param attrIdx the attribute index
 * @return
 */
public boolean isAttrValueChanged(int attrIdx) {
   // get the attribute's posted value
   Object postedValue = getPostedAttribute(attrIdx);
   // get the attribute's current value
   Object newValue = getAttributeInternal(attrIdx);
   // return true is attribute value differs from its posted value
   return isAttributeChanged(attrIdx) &&
      ((postedValue == null && newValue != null) ||
       (postedValue != null && newValue == null) ||
       (postedValue != null && newValue != null &&
```

```
            !newValue.equals(postedValue)));
    }
```

How it works

We added a helper method called `isAttrValueChanged()` to the our custom Entity framework class. This method accepts the attribute's index. The attribute index is generated and maintained by JDeveloper itself. The method first calls `getPostedAttribute()` specifying the attribute index to retrieve the attribute value that was posted to the database. This is the attribute's database value. Then it calls `getAttributeInternal()` using the same attribute index to determine the current attribute value. The two values are then compared. The method `isAttributeChanged()` returns `true` if the attribute value was changed in the current transaction.

Here is an example of calling `isAttrValueChanged()` from an Entity implementation class to determine whether the current value of the employee's last name differs from the value that was posted to the database:

```
super.isAttrValueChanged(this.LASTNAME);
```

See also

Breaking up the application in multiple workspaces, chapter 1

Setting up *BC base classes*, chapter 1

Overriding remove() to delete associated children entities

There are times when during the deletion of a parent **Entity** you want to delete all of the child Entity rows in an Entity **Association** relation. In this recipe we will see how to accomplish this task.

Getting ready

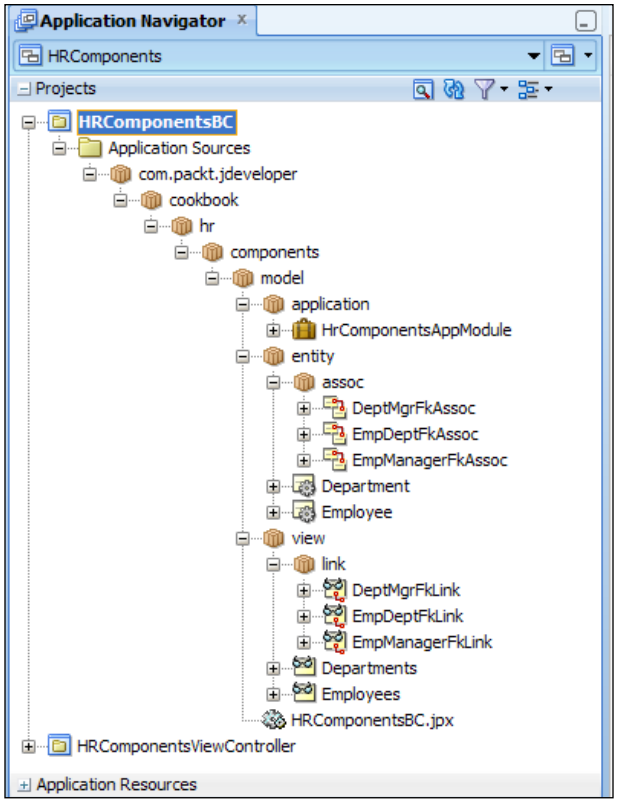
You will need access to the `HR` database schema.

How to do it

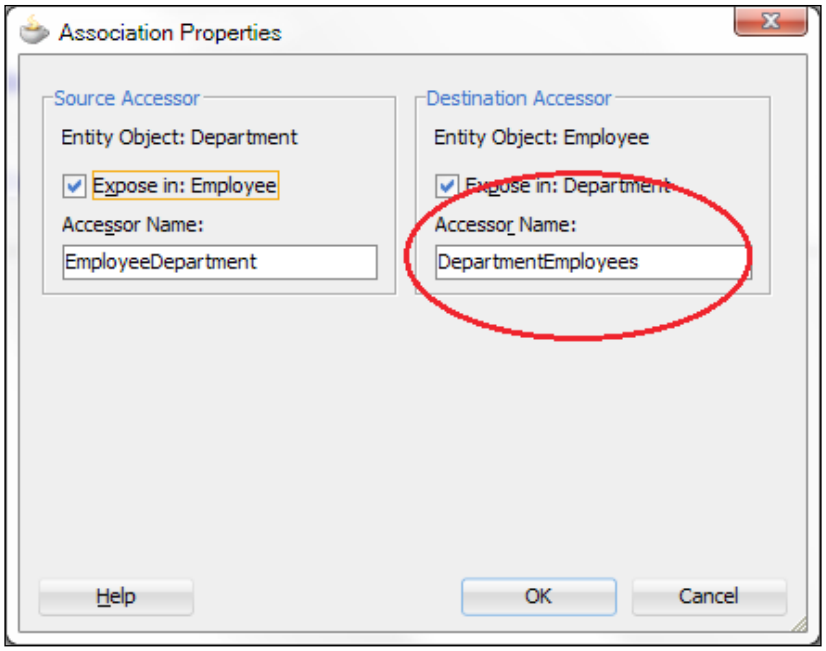
1. Start by creating a new **Fusion Web Application (ADF)** workspace called

`HRComponents`.

2. Create a **Database Connection** for the `HR` schema in the **Application Resource** section of the **Application Navigator**.
3. Use the **Business Components from Tables** selection on the **New Gallery** dialog to create Business Components objects for the `DEPARTMENTS` and `EMPLOYEES` tables. The components in the **Application Navigator** should look similar to this:



4. Double-click on the **EmpDeptFkAssoc** Association on the **Application Navigator** to open the Association definition and click on the **Relationship** tab.
5. Click on the **Edit accessors** button (the pen icon) in the **Accessors** section to bring up the **Association Properties** dialog.
6. Change the **Accessor Name** in the **Destination Accessor** section to **DepartmentEmployees** and click **OK** to continue.



7. Double-click on the **Department** Entity object in the Application Navigator to open its definition and go to the **Java** tab.
8. Click on the **Edit Java Options** button (the pen icon on the top right of the tab) to bring up the **Select Java Options** dialog.
9. On the **Select Java Options** dialog select **Generate Entity Object Class**.
10. Ensure that both the **Accessors** and **Remove Method** check boxes are selected. Click **OK** to continue.
11. Repeat steps 7-10 to create a Java implementation class for the `Employee` Entity object. You do not have to click on the **Remove Method** check box in this case.
12. Open the `DepartmentImpl` Java implementation class for the `Department` Entity object in the JDeveloper Java editor and locate the `remove()` method.
13. Add the following code before the call to `super.remove()`:

```
// get the department employeess accessor
RowIterator departmentEmployees = this.getDepartmentEmployees();
// iterate over all department employees
while (departmentEmployees.hasNext()) {
  // get the department employee
  EmployeeImpl departmentEmployee =
    (EmployeeImpl)departmentEmployees.next();
```

```
        // remove employee
        departmentEmployee.remove();
    }
```

How it works

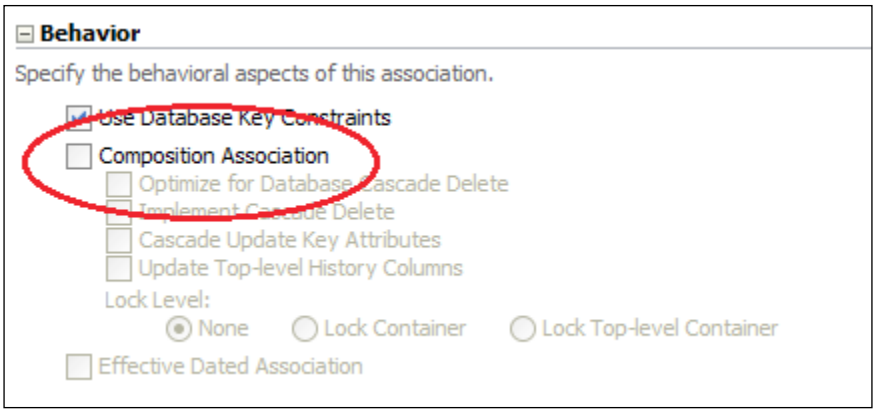
During the creation of the `Department` and `Employee` Entity objects, JDeveloper automatically creates the Entity Associations based on the foreign key constraints that exist among the `DEPARTMENTS` and `EMPLOYEES` database tables. The specific Association that relates a department to its employees was automatically created and it was called `EmpDeptFkAssoc`.

Jdeveloper also exposes the Association via **Accessors** to both the source and destination Entity objects. In step 6 we changed the Accessor names in order to give them more meaningful names. We called the Association accessor that returns the department employees `DepartmentEmployees`. Using Java this Accessor is available in the `DepartmentImpl` class by calling `getDepartmentEmployees()`. This method returns an `oracle.jbo.RowIterator` object that can be iterated over.

Now let's take a closer look at the code added to the `remove()` method. This method is called by the ADF framework each time we delete a record. In it, first we access the current department's employees by calling `getDepartmentEmployees()`. Then we iterate over the department employees checking each time that we have an Employee. This is done by calling `hasNext()` on the employees `RowIterator`. Then for each employee, we get the `Employee` Entity object by calling `next()` and call `remove()` on it to delete it. The call to `super.remove()` finally deletes the `Department` Entity itself. The net result is to delete all employees associated with the specific department before deleting the department itself.

There's more

A specific type of Association called **Composition** Association can be enabled in those cases where an object composition behavior is observed. These are the cases where the composed child Entity cannot exist on its own without the associated "parent" Entity. In these cases, there are special provisions by the ADF framework and JDeveloper itself to fine-tune the delete behavior of children Entities when the parent Entity is removed. These options are available in the Association editor **Relationship** tab under the **Behavior** section.



Overriding remove() to delete a parent entity in an association

In this recipe we will present a technique that you can use in cases that you want to delete the parent **Entity** in an **Association** when the last child Entity is deleted. An example of such case would be to delete a department when the last department employee is deleted.

Getting ready

You will need access to the `HR` schema in your database.

How to do it

1. Start by creating a new **Fusion Web Application (ADF)** workspace called `HRComponents`.
2. Create a **Database Connection** for the `HR` schema in the **Application Resource** section of the **Application Navigator**.
3. Use the **Business Components from Tables** selection on the **New Gallery** dialog to create **Business Components** objects for the `DEPARTMENTS` and `EMPLOYEES` tables.
4. Double-click on the **EmpDeptFkAssoc** Association on the Application Navigator to bring up the Association editor and click on the **Relationship** tab.
5. Click on the **Edit accessors** button (the pen icon) in the **Accessors** section to bring up the **Association Properties** dialog.
6. Change the **Accessor Name** in the **Source Accessor** section to

EmployeeDepartment and click **OK** to continue.

7. Generate custom Java implementation classes for both the `Employee` and `Department` Entity objects.
8. Open the `EmployeeImpl` custom Java implementation class for the `Employee` Entity and locate the `remove()` method.
9. Replace the call to `super.remove()` with this code:

```
// get the associated department
DepartmentImpl department = this.getEmployeeDepartment();
// get for number of employees in the department
int numberOfEmployees =
   department.getDepartmentEmployees().getRowCount();
// check for last employee in the department
if (numberOfEmployees == 1) {
   // delete the last employee
   super.remove();
   // delete the department as well
   department.remove();
}
else {
  // just delete the employee
  super.remove();
}
```

How it works

If you followed the *Overriding remove() to delete associated children entities* recipe in this chapter, then steps 1 through 8 should look familiar to you. These are the basic steps to create the `HRComponents` workspace along with the Business Components associated with the `EMPLOYEES` and `DEPARTMENTS` tables in the `HR` schema. These steps also create the custom Java implementation classes for the `Employee` and `Department` Entity objects and setup the `EmpDeptFkAssoc` Association.

The code in `remove()` first gets the `Department` Entity row by calling the accessor `getEmployeeDepartment()` method. Remember, this was the name of accessor - `EmployeeDepartment` - that we setup in step 6. `getEmployeeDepartment()` returns the custom `DepartmentImpl` that we setup in step 7. In order to determine the number of employees in the associated `Department` we first get the `Employee RowIterator` by calling `getDepartmentEmployees()` on it, and then `getRowCount()` on the `RowIterator`. All that is done in the following statement:

```
int numberOfEmployees = department.getDepartmentEmployees().
getRowCount();
```

Remember that we setup the name of the `DepartmentEmployees` accessor in step 6. Next we checked for the number of employees in the associated department, and if there was only one employee – the one we are about to delete – we first deleted it by calling `super.remove()`. Then we deleted the department itself by calling `department.remove()`. If more than one employees were found for the specific department, we just delete the employee by calling `super.remove()`. This was done in the `else` part of the `if` statement.

See also

Overriding remove() to delete associated children entities, chapter 2

Using a method validator based on a View accessor

In this recipe we will show how to validate an **Entity** object against a **View Accessor** using a custom Entity method validator. The use case that we will cover - based on the `HR` schema – will not allow the user to enter more than a specified number of employees per department.

Getting ready

We will be using the `HRComponents` workspace that we created in the previous recipes in this chapter so that we don't repeat these steps again. You will need access to the `HR` database schema.

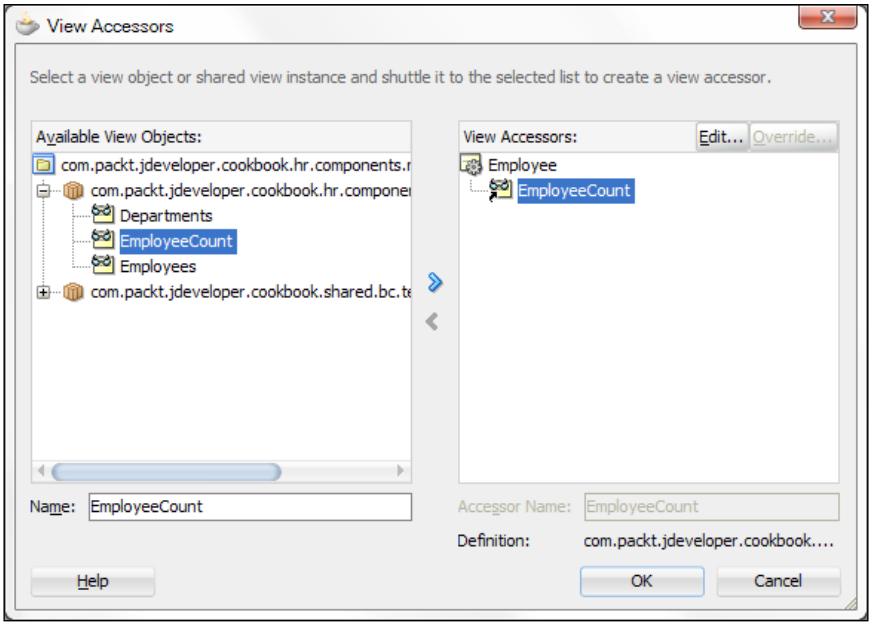
How to do it

1. Right-click on the `com.packt.jdeveloper.cookbook.hr.components.model.view` package of the **HRComponentsBC Business Components** project of the `HRComponents` workspace and select **New View Object...**.
2. Use the **Create View Object** wizard to create a **SQL query** View object called **EmployeeCount** based on the following query:

   ```
   SELECT COUNT(*) AS EMPLOYEE_COUNT FROM EMPLOYEES WHERE
   DEPARTMENT_ID = :DepartmentId
   ```

3. While on the **Create View Object** wizard also do the following:
4. Create a **Bind Variable** called **DepartmentId** of type **Number**.
5. On the **Attribute Settings** page ensure that you select **Key Attribute** for the **EmployeeCount** attribute.

6. On the **Java** page make sure that both the **Generate View Row Class** and **Include accessors** check boxes are checked.
7. Do not add the View object to an **Application Module**.
8. Now, double-click on the **Employee** Entity object to open its definition and go to the **View Accessors** page.
9. Click on the **Create new view accessors** button (the green plus sign icon) to bring up the **View Accessors** dialog.
10. On the **View Accessors** dialog locate the **EmployeeCount** View object and click the **Add instance** button – the blue right arrow button.



11. On the Entity object definition **Business Rules** tab select the **Employee** Entity and click on the **Create new validator** button (the green plus sign icon).
12. On the **Add Validation Rule** dialog select **Method** for the **Rule Type** and enter **validateDepartmentEmployeeCount** for the **Method Name**.
13. Click on the **Failure Handling** tab and in the **Message Text** enter the message "*Department has reached maximum employee limit.*" Click **OK**.
14. Open the `EmployeeImpl` custom implementation Java class, locate the `validateDepartmentEmployeeCount()` method and add the following code to it before the `return true` statement:

```
// get the EmployeeCount view accessor
RowSet employeeCount = this.getEmployeeCount();
```

```
// setup the DepartmentId bind variable
employeeCount.setNamedWhereClauseParam("DepartmentId",
                        this.getDepartmentId());
// run the View Object query
employeeCount.executeQuery();
// check results
if (employeeCount.hasNext()) {
   // get the EmployeeCount row
   EmployeeCountRowImpl employeeCountRow =
      (EmployeeCountRowImpl)employeeCount.next();
   // get the deparment employee count
   Number departmentEmployees = employeeCountRow.getEmployeeCount();
   if (departmentEmployees.compareTo(MAX_DEPARTMENT_EMPLOYEES)>0) {
      return false;
   }
}
```

How it works

We have created a separate query-based View object called `EmployeeCount` for validation purposes. If you look closely at the `EmployeeCount` query you will see that it determines the number of employees in a department. Which department is determined by the bind variable `DepartmentId` used in the `WHERE` clause of the query.

We then added the `EmployeeCount` View object as a View Accessor to the `Employee` object. We called the accessor instance `EmployeeCount` as well. Once you have generated a custom Java implementation class for the `Employee` Entity object, the `EmployeeCount` View Accessor is available by calling `getEmployeeCount()`.

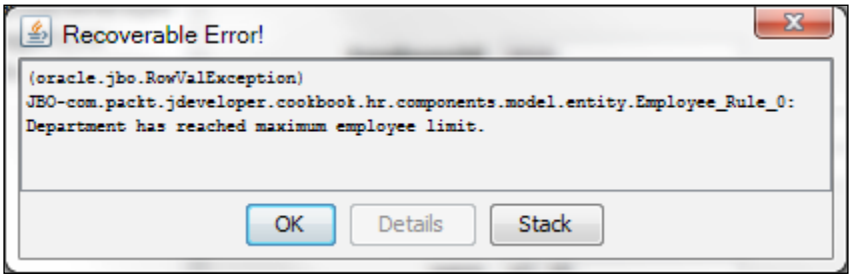
We proceeded by adding a Method Validator to the Entity object. We called the method to use for the validator `validateDepartmentEmployeeCount`. JDeveloper created this method for us in the Entity custom implementation Java class.

The code that we added to the `validateDepartmentEmployeeCount()` method first gets the `EmployeeCount` accessor, and calls `setNamedWhereClauseParam()` on it to set the value of the `DepartmentId` bind variable to the value of the department identifier from the current `Employee`. This value is accessible via the `getDepartmentId()` call. We then executed the `EmployeeCount` View object query by calling its `executeQuery()` method. We checked for the results of the query by calling `hasNext()` on the View object. If the query yielded results, we get the next result row by calling `next()`.

We have casted the `oracle.job.Row` returned by `next()` to an `EmployeeCountRowImpl` so we can call directly its `getEmployeeCount()` accessor. This returns the number of employees for the specific department. We then compared it to a predefined maximum number of employees per department identified by the constant `MAX_DEPARTMENT_EMPLOYEES`.

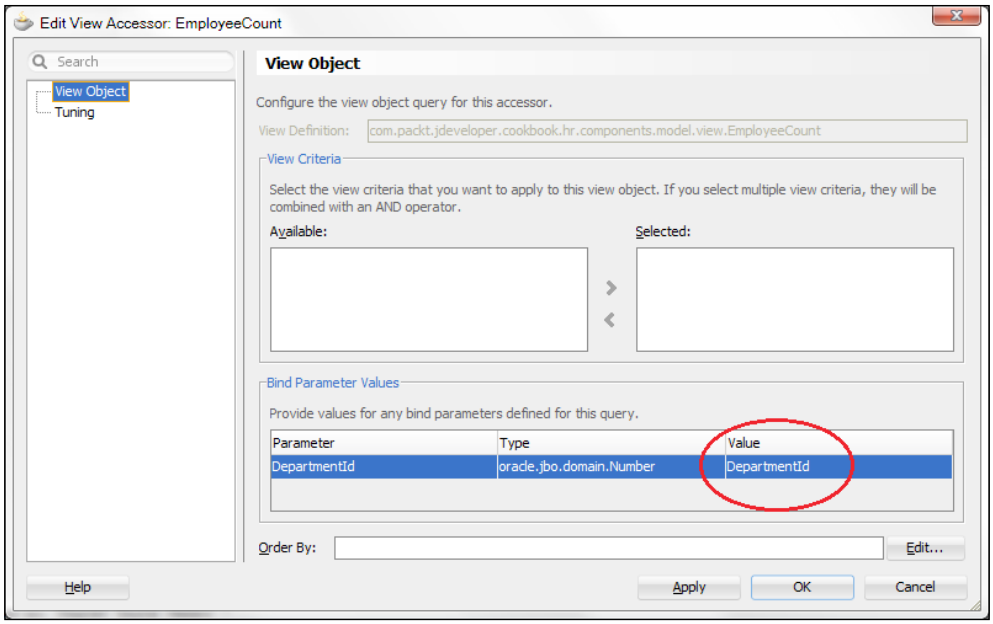
The method validator returns a `false` to indicate that the validation will fail. Otherwise it returns `true`.

Observe what happens when you run the **Application Module** with the **Business Components Browser**. When you try to add a new employee to a department that has more than a predefined number of employees (identified by the constant `MAX_DEPARTMENT_EMPLOYEES`), a validation message is raised. This is the message that we defined for our Method Validator.



There's more

Note that in the code above we called `setNamedWhereClauseParam()` on the `EmployeeCount` View object to set the value of the `DepartmentId` bind variable to the current employee's department id. This could have been done declaratively as well using the **Edit View Accessor** dialog, which is available on the **View Accessors** page of the `Employee` Entity definition page by clicking on the **Edit selected View Accessor** button (the pen icon). On the **Edit View Accessor** dialog, locate the **DepartmentId** bind variable in the **Bind Parameter Values** section and on the **Value** field enter **DepartmentId**. This will set the value of the `DepartmentId` bind variable to the value of the `DepartmentId` attribute of the `Employee` Entity object.



See also

Overriding remove() to delete associated children entities, chapter 2

Using Groovy expressions to resolve validation error message tokens

In this recipe we will expand on the *Using a custom validator based on a View Object accessor* recipe to demonstrate how to use validation message parameter values based on **Groovy expressions**. Moreover we will show how to retrieve the parameter values from a specific parameter bundle.

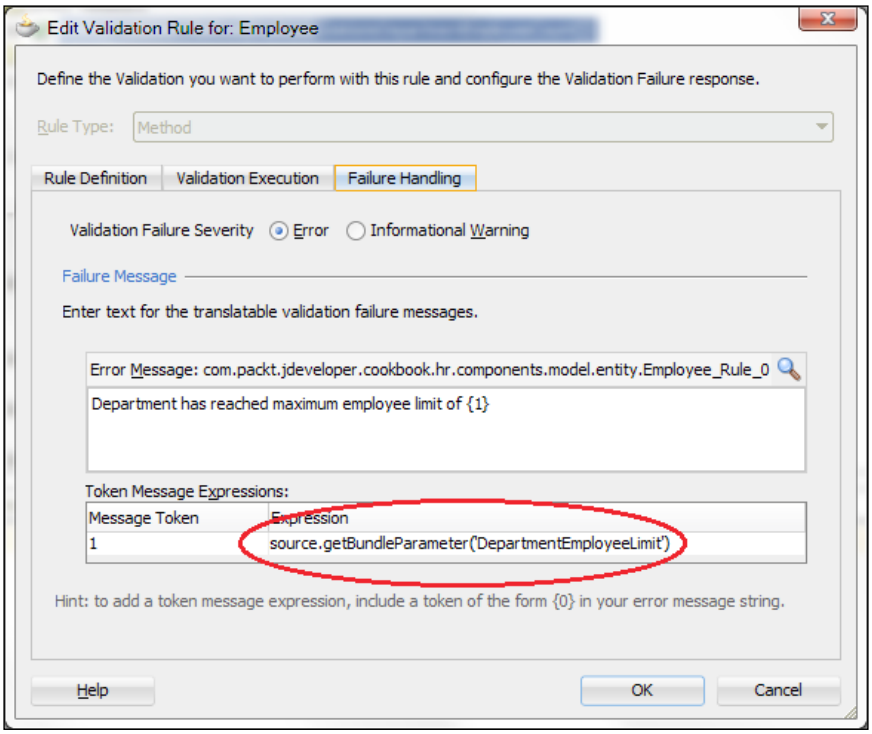
Groovy is a dynamic language that runs inside the Java Virtual Machine. In the context of the ADF Business Components it can be used to provide declarative expressions that are interpreted at runtime. Groovy expressions can be used in validation rules, validation messages and parameters, attribute initializations, bind variable initializations, and more.

Getting ready

This recipe builds on the *Using a custom validator based on a View Object accessor* recipe. It also relies on recipes *Breaking up the application in multiple workspaces* and *Setting up BC base classes* presented in chapter 1.

How to do it

1. In the **Application Navigator** double-click on the **Employee Entity** object definition and go to its **Business Rules** tab.
2. Double-click on the **validateDepartmentEmpoyeeCount Method Validator** to bring up the **Edit Validation Rule** dialog and go to the **Failure Handling** tab.
3. Change the **Error Message** to *Department has reached maximum employee limit of {1}*.
4. For the **Message Token 1 Expression** in the **Token Message Expressions** section enter the following expression `source.getBundleParameter('DepartmentEmployeeLimit')`



5. Now, open the shared components workspace and locate the `ExtEntityImpl` **Entity** framework extension class. Add the `getBundleParameter()` method below to it:

```
public String getBundleParameter(String parameterKey) {
   // use BundleUtils to load the parameter
   return BundleUtils.loadParameter(parameterKey);
}
```

6. Locate the `BundleUtils` helper class in the `com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages` packageand add the `loadParameter()` method below:

```
public static String loadParameter(final String parameterKey) {
   // get access to the error message parameters bundle
   final ResourceBundle parametersBundle =
      ResourceBundle.getBundle(PARAMETERS_BUNDLE, Locale.getDefault());
   // get and return the the parameter value
   return parametersBundle.getString(PARAMETER_PREFIX + parameterKey);
}
```

Finally, locate the `ErrorParams.properties` property file and add the following text to it:

```
parameter.DepartmentEmployeeLimit=2
```

How it works

For this recipe first we added a parameter to the Method Validator message. The parameter is indicated by adding parameter placeholders to the message using braces `{}`. The parameter name is indicated by the value within the braces. In our case we defined a parameter called `1` by entering `{1}`. We then had to supply the parameter value. Instead of hard-coding the parameter value we used the following Groovy expression:

```
source.getBundleParameter('DepartmentEmployeeLimit').
```

The `source` prefix allows us to reference an Entity object method from the validator. In this case the method is called `getBundleParameter()`. This method accepts a parameter key which is used to load the actual parameter value from the parameters bundle. In this case we have used the `DepartmentEmployeeLimit` parameter key.

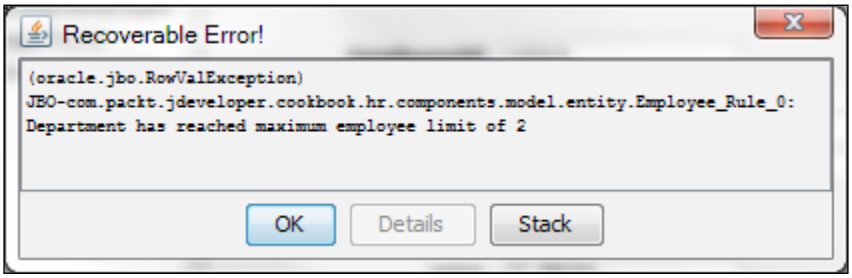
Then we implemented the `getBundleParameter()` method. We implemented this method in the base Entity custom framework class so that it is available to all Entity objects. If you look at the code in `getBundleParameter()` you will see that it loads and returns the parameter value using the helper `BundleUtils.loadParameter()`.

We introduced the helper class `BundleUtils` while we worked on the *Using a generic View Controller actions framework* recipe in chapter 1.

The `BundleUtils.loadParameter()` method pre-pends the parameter with the prefix `parameter.`.

Finally, we defined the `parameter.DepartmentEmployeeLimit` parameter in the `ErrorParams.properties` parameters bundle. Refer to the *Using a custom exception class* recipe in chapter 1 for further information on this bundle.

When we test the **Application Module** using the **Business Components Browser**, when the validation is raised the error message looks like this:



As you can see the message parameter placeholder `{1}` which was defined originally in the message, has been substituted with the actual parameter value (the number 2).

See also

Breaking up the application in multiple workspaces, chapter 1

Setting up BC base classes, chapter 1

Using a custom exception class, chapter 1

Using a generic View Controller actions framework, chapter 1

Using doDML() to enforce a detail record for a new master record

In this recipe we will consider a simple technique that we can use to enforce having detailed records when inserting a new master record in an **Entity Association** relationship. The use case demonstrates how to enforce creating at least one employee at the time when a new department is created.

Getting ready

We will use the `HR` database schema and the `HRComponents` workspace that we have created in previous recipes in this chapter.

How to do it

Open the `DepartmentImpl` custom Entity implementation class and override the `doDML()` method using the **Override Methods** dialog.

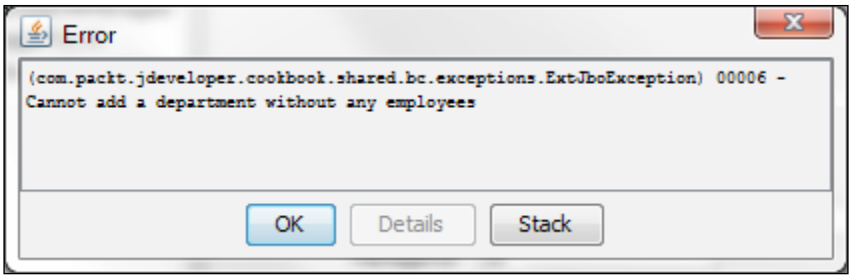
Add the following code to the `doDML()` method before the call to `super.doDML()`:

```
        // check for insert
        if (DML_INSERT == operation) {
            // get the department employees accessor
            RowIterator departmentEmployees = this.
getDepartmentEmployees();
            // check for any employees
            if (!departmentEmployees.hasNext()) {
                // avoid inserting the department if there are no
employees for it
                throw new ExtJboException("00006");
            }
        }
```

How it works

In the overridden `doDML()`, we only check for insert operations. This is indicated by comparing the DML `operation` flag which is passed as a parameter to `doDML()` to the `DML_INSERT` flag. Then we get the department employees from the `DepartmentEmployees` accessor by calling `getDepartmentEmployees()`. The `DepartmentEmployees` accessor was set up during the creation of the `HRComponents` workspace earlier in this chapter. We check whether the `RowIterator` returned has any rows by calling `hasNext()` on it. If this is not the case, i.e. there are no employees associated with the specific department that we are about to insert, we alert the user by throwing an `ExtJboException` exception. The `ExtJboException` exception is part of the shared components workspace and it was developed in the *Using a custom exception class* recipe back in chapter 1.

When testing the **Application Module** with the **Business Components Browser** we get the following error message when we try to insert a new department without any associated employees:



There's more

Note that in the case that the exception is thrown, this will result in data being posted to the database.

See also

Using a custom exception class, chapter 1

Overriding remove() to delete associated children entities, chapter 1

3

A Different Point of View: View Object Techniques

In this chapter, we will cover:

- Iterating a View object using a secondary rowset iterator
- Setting default values for View Row attributes
- Controlling the updatability of View object attributes programmatically
- Setting a View object attribute's Queryable property
- Using a transient attribute to indicate a new View object row
- Conditionally inserting new rows at the end of the rowset
- Using `findAndSetCurrentRowByKey()` to set the View object currency
- Restoring the current row after a transaction rollback
- Dynamically changing the View object's query `WHERE` clause
- Removing a row from a rowset without deleting it from the database

Introduction

View objects are an essential part of the **ADF Business Components**. They work in conjunction with **Entity** objects, for **Entity-based View** objects, to support querying the database, retrieving data from the database and building rowsets of data. The underlying Entities enable an updatable data model that supports the addition, deletion and modification of data. They also support the enforcement of business rules and the permanent storage of the data to the database.

In cases where an updatable data model is not required, the framework supports a **read-only** View object, one that is not based on Entity objects but on a SQL query supplied by the developer. *Read-only* View objects should be used in cases where `UNION` and `GROUP BY` clauses appear in the View object queries. In other cases, even though an updatable data model is not required, the recommended practice is to base the View objects on Entities and allow the JDeveloper framework-supporting wizards to build the SQL query automatically instead.

This chapter presents several techniques covering a wide area of expertise related to View objects.

Iterating a View object using a secondary rowset iterator

There are times when you need to iterate through a View object rowset programmatically. In this recipe we will see how to do this using a secondary rowset iterator. The following scenario will be considered: we will iterate over the `Employees` rowset and for each employee that belongs to the *Sale*s department, we will increase the employee's commission by a certain percentage.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

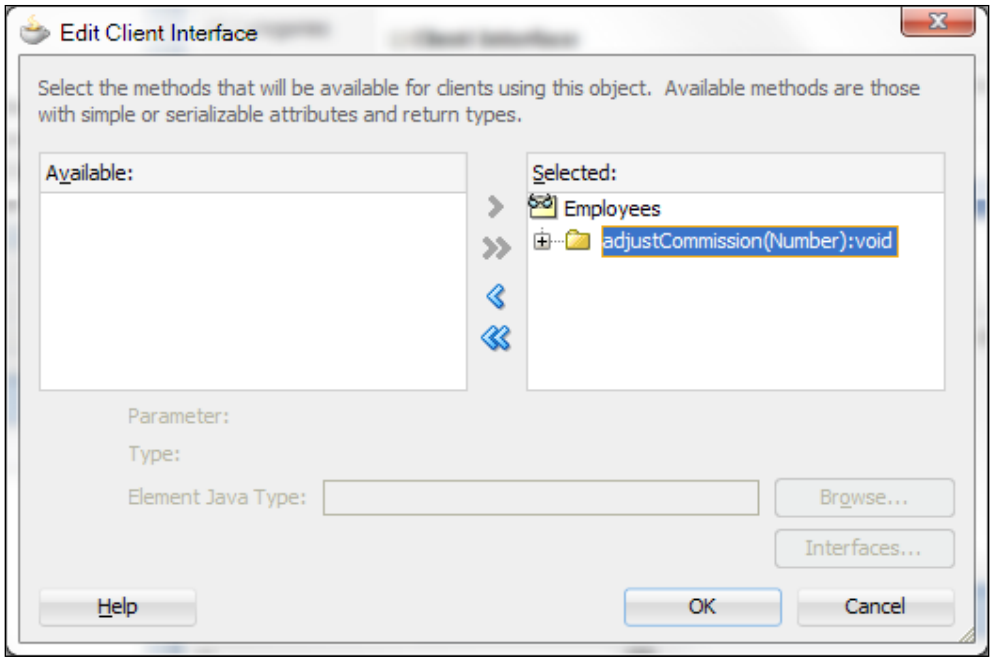
How to do it...

1. Open the **Employees** View object definition and go to the **Java** page.
2. Click on the **Edit java options** button (the pen icon) to open the **Select Java Options** dialog.

3. Click on the **Generate View Object Class** and **Generate View Row Class** check boxes. Ensure that the **Include accessors** check box is also selected.
4. Click **OK** to proceed with the creation of the custom implementation classes.
5. Add the following helper method to the `EmployeesImpl.java`:

```
public void adjustCommission(Number commissionPctAdjustment) {
  // check for valid commission adjustment
  if (commissionPctAdjustment != null) {
    // create an employee secondary rowset iterator
    rowsetIterator employees = this.createrowsetIterator(null);
    // reset the iterator
    employees.reset();
    // iterate the employees
    while (employees.hasNext()) {
      // get the employee
      EmployeesRowImpl employee = (EmployeesRowImpl)employees.next();
      // check for employee belonging to the sales department
      if (employee.getDepartmentId() != null &&
        SALES_DEPARTMENT_ID ==
        employee.getDepartmentId().intValue()) {
        // calculate adjusted commission
        Number commissionPct = employee.getCommissionPct();
        Number adjustedCommissionPct =
          (commissionPct != null) ? commissionPct.
add(commissionPctAdjustment) :
          commissionPctAdjustment;
        // set the employee's new commission
        employee.setCommissionPct(adjustedCommissionPct);
      }
    }

    // done with the rowset iterator
    employees.closerowsetIterator();
  }
}
```

6. On **the Employees Java** page click on the **Edit view object client interface** button (the pen icon).
7. On the **Edit Client Interface** dialog, shuttle the `adjustCommission()` method to the **Selected** list and click **OK**.



8. Open the **HRComponentsAppModule Application Module** definition and go to the **Java** page.
9. Click on the **Edit java options** button.
10. On the **Select Java Options** dialog click on the **Generate Application Module Class** check box. Then click **OK** to close the dialog.
11. Open the `HrComponentsAppModuleImpl` class and add the following method:

```
public void adjustCommission(Number commissionPctAdjustment) {
  // execute the Employees View object query to create a rowset
  this.getEmployees().executeQuery();
  // adjust the employees commission
  this.getEmployees().adjustCommission(commissionPctAdjustment);
}
```

12. Return to the Application Module definition **Java** page and use the **Edit application module client interface** button to add the `adjustCommission()` method to the Application Module's client interface.

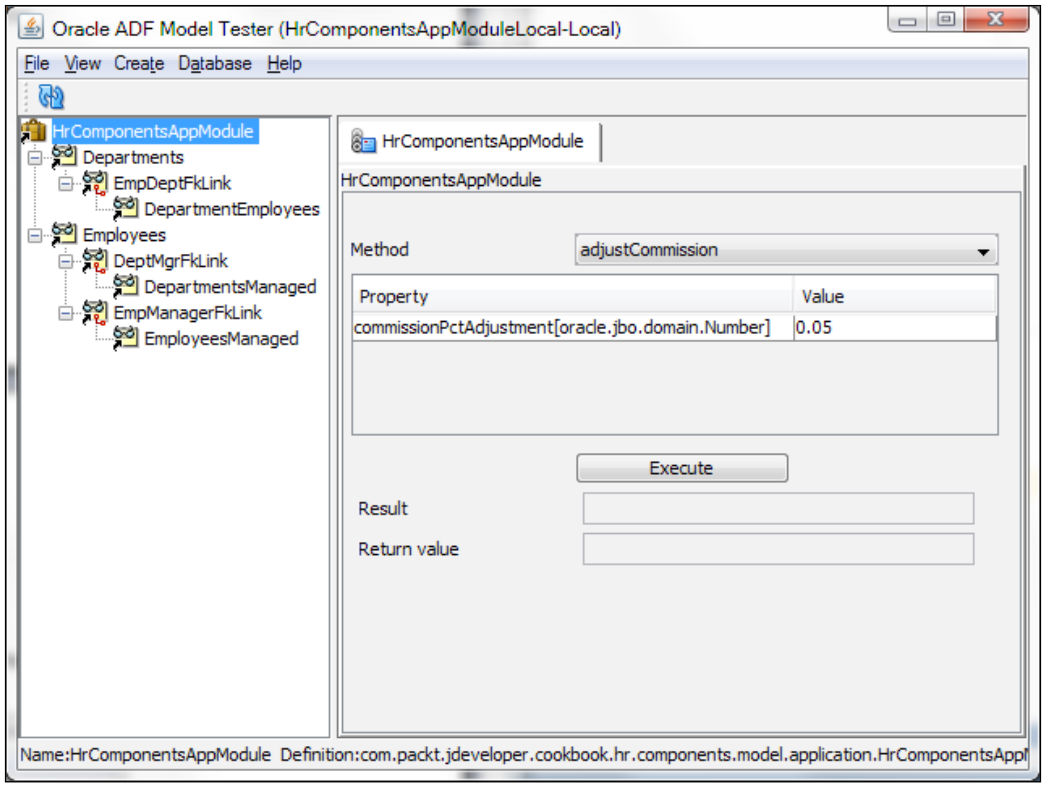
How it works...

We created a View object custom Java implementation class for the `Employees` View object and we added a method called `adjustCommission()`. The method was then exposed to the View object's client interface so that it can be accessible and called using the `Employees` interface.

The `adjustCommission()` method adjusts the commission for all employees belonging to the *Sales* department. The method accepts the commission adjustment percentage as an argument. We called the `createrowsetIterator()` method to create a secondary iterator, which we then used to iterate over the `Employees` rowset. This is the recommended practice to perform programmatic iteration over a rowset. The reason is that the View object instance that is being iterated may be bound to UI components and that iterating it directly will interfere with the UI. In this case, you will see the current row changing by itself.

We then called the `reset()` method to initialize the rowset iterator. This places the iterator to the slot before the first row in the rowset. We iterated the rowset by checking whether a next row exists. This is done by calling `hasNext()` on the iterator. If a next row exists, we retrieve it by calling `next()`. `next()` returns an `oracle.jbo.Row`. We cast the default `Row` object that is returned to an `EmployeesRowImpl` so we can use the custom setter and getter methods to manipulate the `Employee` row.

For testing purposes we created a custom Application Module implementation class and added a method called `adjustCommission()` to it. We exposed this method to the Application Module client interface so that we can call it from the **ADF Model Tester**. Inside the `adjustCommission()` we execute the `Employees` View object query by calling `executeQuery()` on it. We get the `Employees` View object instance via the `getEmployees()` getter method. Finally, we call the `adjustCommission()` method we implemented in `EmployeesImpl` to adjust the employees' commission.



There's more...

In order to be able to iterate a View object rowset using a secondary iterator, the View object **Access Mode** in the **General > Tuning** section must set to `Scrollable`. Any other access mode setting will result in a **JBO-25083: Cannot create a secondary iterator on row set {0} because the access mode is forward-only or range-paging** error when attempting to create a secondary iterator. To iterate View objects configured with Range Paging, use the range paging View object API methods. Specifically, call `getEstimatedRangePageCount()` to determine the number of pages and for each page call `scrollToRangePage()`. Then determine the range page size by calling `getRangeSize()` and iterate through the page calling `getRowAtRangeIndex()`.

Pitfalls when iterating over large rowsets

Before iterating a View object rowset, consider that iterating the rowset may result in fetching a large number of records from the database to the middle layer. In this case other alternatives should be considered, such as running the iteration asynchronously on a separate work manager, for instance. In certain cases, for instance when iterating in order to compute a total amount, consider using any of the following:

- **Groovy** expressions such as `object.getRowSet().sum('SomeAttribute')`
- **Analytic functions**, such as `COUNT(args) OVER ([PARTITION BY <…>] …)`, in the View object's SQL query instead.

See also

Overriding remove() to delete associated children entities, chapter 2.

Setting default values for View Row attributes

In this recipe we will see how to set default values for View object attributes. There are a number of places where you can do this, namely:

- In the overridden `create()` method of the View object row implementation class
- Declaratively using a Groovy expression
- In the attribute getter method

The following use case will be considered: for a newly created employee, we will set the employee's hire date to the current date.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

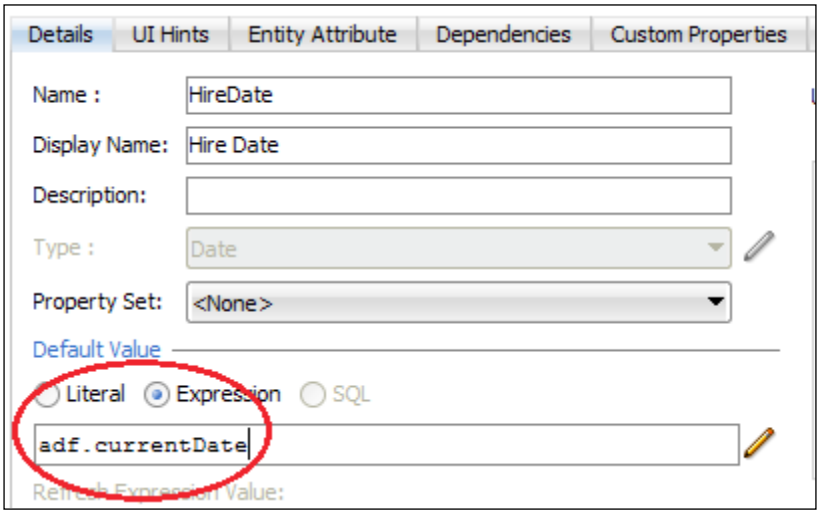
How to do it...

1. To set the attribute's default value in the overridden `create ()` method, create a **View Row** Java implementation class for the `Employees` View object.

2. Open the `EmployeesRowImpl.java` custom View Row Java implementation class and override the `create()` method using the **Override Methods...** button (the green left arrow on the editor toolbar).
3. To set the default employee's hire date to today's date, add the following code to `create()` immediately after the call to `super.create()`:

```
// set the default hire date to today
this.setHireDate((Date)Date.getCurrentDate());
```

5. To set the attribute's default value using a Groovy expression, open the `Employees` View object definition and go to the **Attributes** page.
6. Select the attribute that you want to initialize – **HireDate** in this case.
7. Select the **Details** tab.
8. In the **Default Value** section select **Expression** and enter the following Groovy expression: `adf.currentDate`



9. Another way to return a default value for a View object attribute is in the attribute getter method. To return a default employee hire date, locate the View object attribute getter in the View object row implementation class. In this example this is the `getHireDate()` method in `EmployeesRowImpl.java`.
10. Replace the existing code in `getHireDate()` with the following:

```
// get the HireDate attribute value
Date hireDate = (Date)getAttributeInternal(HIREDATE);
// check for null and return today's date if needed
return (hireDate == null) ? (Date)Date.getCurrentDate() : hireDate;
```

How it works...

This recipe presents three different techniques to set default values to View object attributes. The first technique (steps 1-3) overrides the View row `create()` method. This method is called by the ADF Business Components framework when a View object row is being created. In the code sample above we first call the parent `ViewRowImpl create()` to allow the framework processing. Then we initialize the attribute by calling its setter method - `setHireDate ()` in this case - supplying `Date.getCurrentDate()` for the attribute value.

The second technique (steps 4-7) initializes the View object attribute declaratively using a Groovy expression. The Groovy expression used to initialize the `HireDate` attribute is `adf.currentDate`. Note that we changed the attribute's **Value Type** field to **Expression** so that it can be interpreted as an expression instead of a literal value. This expression when evaluated at runtime by the framework retrieves the current date.

Finally the last technique (steps 8-9) uses the attribute getter – `getHireDate()` for this example - to return a default value. Using this technique we don't actually set the attribute value; instead we return a default value, which can be subsequently applied to the attribute. Also notice that this is done only if the attribute does not already have a value (the check for `null`).

There's more...

A common use case related to this topic is setting an attribute's value based on the value of another related attribute. Consider for instance the use case where the employee's commission should be set to a certain default value if the employee is part of the *Sales* department. Also consider the case where the employee's commission should be cleared if the employee is not part of the *Sales* department. In addition to accomplishing this task with Groovy as stated earlier, it can also be implemented in the employee's `DepartmentId` setter, i.e. in the `setDepartmentId()` method as it is shown below:

```
public void setDepartmentId(Number value) {
// set the department identifier
   setAttributeInternal(DEPARTMENTID, value);
   // set employee's commission based on employee's department
   try {
    // check for Sales department
      if (value != null && SALES_DEPARTMENT_ID == value.intValue()) {
          // if the commission has not been set yet
         if (this.getCommissionPct() == null) {
          // set commission to default
            this.setCommissionPct(new Number(DEFAULT_COMMISSION));
         }
      } else {
```

```
            // clear commission for non Sales department
            this.setCommissionPct(null);
          }
        } catch (SQLException e) {
        // log the exception
          LOGGER.severe(e);
        }
    }
```

Specifying default values at the Entity level

Note that default values can be supplied at the Entity object level as well. In this case, all View objects based on the particular Entity object will inherit the specific behavior. You can provide variations for this behavior by implementing the techniques outlined in this recipe for specific View objects.

See also

Overriding remove() to delete associated children entities, chapter 2.

Controlling the updatability of View object attributes programmatically

In ADF there are a number of ways to control whether a View object attribute can be updated or not. It can certainly be done declaratively in the **Attributes** tab via the **Updatable** combo, or on the front end **ViewController** layer by setting the `disabled` or `readOnly` attributes of a **JSF** page component. Programmatically it can be done either on a **Backing Bean,** or if you are utilizing ADF Business Components on a custom View object row implementation class. This recipe demonstrates the latter case. For our example we will implement the following use case: we will disable updating any of the `Department` attributes for specific departments having more than a specified number of employees.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Create View row implementation classes for the `Department` and `Employee` View objects. Ensure that in both cases you have selected **Include accessors** on the **Java Options** dialog.
2. Open the `DepartmentsRowImpl` class in the Java editor.
3. Use the **Override Methods...** button to override the `isAttributeUpdateable()` method.
4. Replace the call to `super.isAttributeUpdateable(i)` with the following code:

```
// get the number of employees for the specific department
int departmentEmployeeCount =
        this.getEmployees().getRowCount();
// set all attributes to non-updatable if the department
// has more than a specified number of employees
return (departmentEmployeeCount > MAX_EMPLOYEES_UPDATE)
        ? false
        : super.isAttributeUpdateable(prameterIdx);
```

How it works...

The `isAttributeUpdateable()` method is called by the framework in order to determine whether a specific attribute is updatable or not. The framework supplies the attribute in question to the `isAttributeUpdateable()` method as an attribute index parameter. Inside the method we add the necessary code to conditionally enable or disable the specific attribute. We enable or disable the attribute by returning a `boolean` indicator: a `true` return value indicates that the attribute can be updated.

There's more

Because the `isAttributeUpdateable()` method could potentially be called several times for each of the View object attributes (when bound to page components for instance), avoid writing code in it that will hinder the performance of the application. For instance, avoid calling database procedures or executing expensive queries in it.

Controlling attribute updatability at the Entity level

Note that we can conditionally control attribute updatability at the Entity object level as well by overriding the `EntityImpl isAttributeUpdateable()` method. In this case all View objects based on the particular Entity will exhibit the same attribute updatability behavior. You can provide different behavior in this case by overriding `isAttributeUpdateable()` for specific View objects.

See also

Overriding remove() to delete associated children entities, chapter 2.

Setting a View object attribute's Queryable property programmatically

The `Queryable` property when set for a View object attribute indicates that the specific attribute can appear on the View object's `WHERE` clause. This has the effect of making the attribute available in all search forms and allows the user to search for it. Declaratively you can control whether an attribute is queryable or not by checking or un-checking the **Queryable** check box in the View object **Attributes** > **Details** tab. But how do you accomplish this task programmatically and for specific conditions?

This recipe will show how to determine the `Queryable` status of an attribute and change it if needed based on a particular condition.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it...

1. Open the `ExtViewObjectImpl` View object custom framework class in the Java editor.
2. Add the following method to it:

```
protected void setQueriable(int attribute, boolean condition) {
  // get the attribute definition
  AttributeDef def = getAttributeDef(attribute);
  // set/unset only if needed
  if (def != null && def.isQueriable() != condition) {
    // set/unset queriable
    ViewAttributeDefImpl attributeDef =
```

```
                    (ViewAttributeDefImpl)def;
            attributeDef.setQueriable(condition);
        }
    }
```

How it works...

We have added the `setQueriable()` method to the `ExtViewObjectImpl` View object custom framework class. This makes the method available to all View objects. The method accepts the specific attribute index (`attribute`) and a boolean indicator whether to set or unset the `Queryable` flag (`condition`) for the specific attribute.

In `setQueriable()` we first call `getAttributeDef()` to retrieve the `oracle.jbo.AttributeDef` attribute definition. Then we call `isQueriable()` on the attribute definition to retrieve the `Queryable` condition. If the attribute's current `Queryable` condition differs from the one we have passed to `setQueriable()`, we call `setQueriable()` on the attribute definition to set the new value.

There's more...

Note that you can control the `Queryable` attribute at the Entity object level as well. In this case all View objects based on the specific Entity object will inherit this behavior. This behavior can be overridden declaratively or programmatically for the View object as long as the new value is more restrictive than the inherited value.

See also

Breaking up the application in multiple workspaces, chapter 1.

Setting up BC base classes, chapter 1.

Using a transient attribute to indicate a new View object row

For **Entity-based** View objects there is a simple technique you can use to determine whether a particular row has a **New** status. The status of a row is New when the row is first created. The row remains in the New state until it is successfully committed to the database. It then goes to an **Unmodified** state. Knowledge of the status of the row can be used to set up enable/disable conditions on the front end user interface.

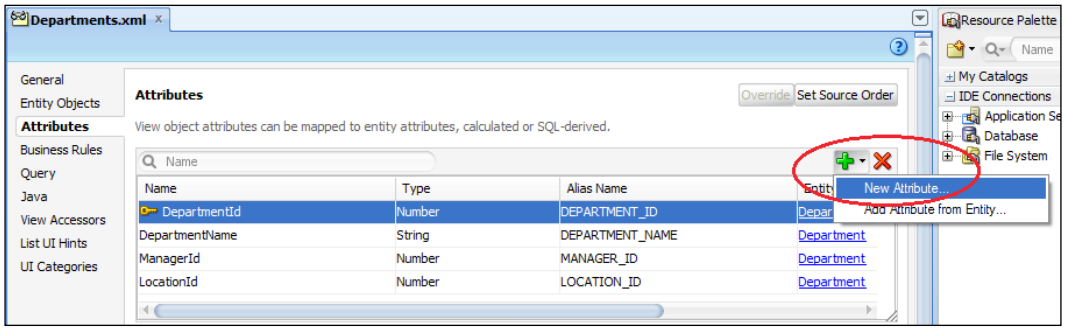
In this recipe we will see how to utilize a transient View object attribute to indicate the New status of the View object row.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `Departments` View object definition.
2. Go to the **Attributes** tab and click on the **Create new attribute** button (the green plus sign icon).
3. Select **New Attribute...** from the context menu.



4. On the **New View Object Attribute** dialog enter `IsNewRow` and click **OK**.
5. By default the new attribute is of type `String`, so change it to a `Boolean` using the **Type** choice list in the **Details** tab.
6. If you don't already have a custom View Row implementation class created, use the **Java** tab to create one. Make sure that you have selected the **Include accessors** check box.
7. Open the `DepartmentsRowImpl.java` View Row implementation class and locate the `getIsNewRow()` method.
8. Replace the code inside the `getIsNewRow()` method with the following:

```
// return true if the row status is New
return Row.STATUS_NEW ==
            this.getDepartment().getEntityState();
```

How it works...

First we create a new `Boolean` transient attribute called `IsNewRow`. This attribute will be used to indicate whether the status of the View object row is New or not. A transient attribute does not correspond to a database table column and it can be used as placeholder for intermediate data. Then we generate a custom View Row implementation class. On the transient attribute getter, `getIsNewRow()` in this case, we get access to the Entity object. For this recipe the `Department` entity is returned by calling the `getDepartment()` getter. We get the Entity row state by calling `getEntityState()` on the `Department` Entity object and we compare it to the constant `Row.STATUS_NEW`.

Once the `IsNewRow` attribute is bound to a JSF page, it can be used in **Expression Language (EL)** expressions. For instance the EL expression below indicates a certain disabled condition based on the row status not being *New*:

```
disabled="#{bindings.IsNewRow.inputValue ne true}"
```

There's more

The following table summarizes the Entity object Row States.

Entity State	*Description*	*State Transition*
New	*Indicates a new Entity.*	▶ *When a new Entity is first created.* ▶ *When* `setAttribute()` is called on an *Initialized* Entity.
Initialized	*Indicates that a new Entity is initialized and thus it is removed from the transaction's pending changes list.*	▶ *When* `setNewRowState()` is explicitly called on a New Entity.
Unmodified	*Indicates an unmodified Entity.*	▶ *When the Entity is retrieved from the database.* ▶ *After successfully committing a* New of Modified Entity.
Modified	*Indicates the state of a modified Entity.*	▶ *When* `setattribute()` is called on an Unmodified Entity.
Deleted	*Indicates a deleted Entity.*	▶ *When* `remove()` is called on an Unmodified or Modified Entity.

Dead	*Indicates a dead Entity.*	▶ *When* `remove()` is called on a New or Initialized Entity. ▶ After successfully committing a Deleted Entity.

See also

Overriding remove() to delete associated children entities, chapter 2.

Conditionally inserting new rows at the end of the rowset

When you insert a new row on a rowset, by default the new row is inserted at the current slot in the rowset. There are times however that you want to override this default behavior for the application that you are developing.

In this recipe we will see how to conditionally insert new rows at the end of the rowset by implementing generic programming functionality at the base View object framework implementation class.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it

1. Open the `ExtViewObjectImpl.java` custom View object framework class in the Java editor.
2. Override the `insertRow()` method.
3. Replace the call to `super.insertRow()` in the generated `insertRow()` method with the following code:

```
// check for overriden behavior based on custom property
if ("true".equalsIgnoreCase((String)this.getProperty(
```

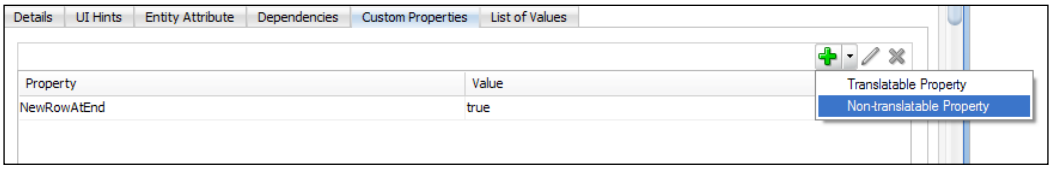
```
                                NEW_ROW_AT_END))) {
    // get the last row in the rowset
      Row lastRow = this.last();
      if (lastRow != null) {
              // get index of last row
        int lastRowIdx = this.getRangeIndexOf(lastRow);
        // insert row after the last row
        this.insertRowAtRangeIndex(lastRowIdx, row);
        // set inserted row as the current row
        this.setCurrentRow(row);
      } else {
        super.insertRow(row);
      }
    } else {
      // default behavior: insert at current rowset slot
      super.insertRow(row);
    }
```

How it works

We have overridden the ADF Business Components framework `insertRow()` method in order to implement custom row insertion behavior. Moreover, we conditionally override the default framework behavior based on the existence of the **Custom Property** `NewRowAtEnd` identified by the constant `NEW_ROW_AT_END`. So if this custom property is defined for specific View objects we do the following: determine the index of the last row in the rowset by calling `getRangeIndexOf()` and then call `insertRowAtRangeIndex()` to insert the new row at the specific last row index. Finally we set the rowset currency to the row just inserted.

If the `NewRowAtEnd` custom property is not defined in the View object, then the row is inserted by default at the current slot in the rowset.

To add a custom property to a View object use the drop down menu next to the **Add Custom Property** button (the green plus sign icon) and select **Non-translatable Property**. The Add Custom Property button is located in the **Attributes** > **Custom Properties** tab.



There's more

Note that If you have configured **Range Paging** access mode for the View object, calling `last()` will produce a **JBO-25084: Cannot call last() on row set {0} because the access mode uses range-paging** error. In this case, call `getEstimatedRangePageCount()` to determine the number of pages and `setRangeStart()` to set the range to the last page instead.

Inserting new rows at the beginning of the rowset

The use case presented in this recipe can be easily adapted to insert a row at the beginning of the rowset instead. In this case, you will need to call `this.first()` to get the first row instead. The functionality of getting the row index and inserting the row at the specified index should work as is.

See also

Breaking up the application in multiple workspaces, chapter 1.

Setting up BC base classes, chapter 1.

Using findAndSetCurrentRowByKey() to set the View object currency

You can set the currency on a View object by calling its `findAndSetCurrentRowByKey()` method. The method accepts two arguments: a `Key` object that is used to locate the row in the View object, and an integer indicating the range position of the row (for View objects configured with Range Paging access mode).

This recipe demonstrates how to set the View object row currency by implementing a helper method called `refreshView()`. In it we first save the View object currency, re-query the View object and finally restore its currency to the original row before the re-query. This has the effect of refreshing the View object while keeping the current row.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it

Open the `ExtViewObjectImpl.java` View object framework extension class into the Java editor.

Override the `create()` method.

Add the following code after the call to `super.create()`:

```
// allow read-only View objects to use findByKey() methods
this.setManageRowsByKey(true);
```

While at the `ExtViewObjectImpl.java` add the following `refreshView()` method:

```
public void refreshView() {
    Key curRowKey = null;
    int rangePosOfCurRow = -1;
    int rangeStart = -1;

    // get and save the current row
    Row currentRow = getCurrentRow();
    // do this only if we have a current row
    if (currentRow != null) {
        // get the row information
        curRowKey = currentRow.getKey();
        rangePosOfCurRow = getRangeIndexOf(currentRow);
        rangeStart = getRangeStart();
    }

    // execute the View object query
    executeQuery();

    // if we have a current row, restore it
    if (currentRow != null) {
        setRangeStart(rangeStart);
        findAndSetCurrentRowByKey(curRowKey, rangePosOfCurRow);
    }
}
```

How it works

First we override the `create()` method in the View object framework extension class. We do this so we can call the `setManageRowsByKey()` method. This method will allow us to use framework find methods utilizing Key objects on read-only View objects as well. By default this is not the case for read-only View objects.

Then we implement the `refreshView()` method. We made `refreshView()` `public` so that it could be called explicitly for all View objects. In it we first determine the current row in the rowset by calling `getCurrentRow()`. This method returns an `oracle.jbo.Row` object indicating the current row, or `null` if there is no current row in the rowset. If there is a current row, we get all the necessary information in order to be able to restore it after we re-query the View object. This information include the current row's Key (`getKey()`), the index of the current row in range (`getRangeIndexOf()`) and the row's range (`getRangeStart()`).

Once the current row information is saved, we re-execute the View object's query by calling `executeQuery()`.

Finally, we restore the current row by setting the range and the row within the range by calling `setRangeStart()` and `findAndSetCurrentRowByKey()` respectively.

There's more

Range Paging is an optimization technique that can be utilized by View objects returning large result sets. The effect of using it is that it limits the result set to a specific number of rows, as determined by the **Range Size** option setting. So instead of retrieving hundreds or even thousands of rows over the network and caching them into the middle layer memory, only the ranges of rows utilized are retrieved.

The methods used in this recipe to retrieve the row information will work regardless of using Range Paging or not.

See also

Breaking up the application in multiple workspaces, chapter 1.

Setting up BC base classes, chapter 1.

Restoring the current row after a transaction rollback, chapter 2.

Restoring the current row after a transaction rollback

On a transaction rollback, the default behavior of the ADF framework is to set the current row to the first row in the **rowset**. This is certainly not the behavior you expect to see, when you rollback during editing a record. It is more intuitive that the current record remains even after the rollback operation.

This recipe shows how to accomplish the task of restoring the current row after a transaction rollback.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtApplicationModuleImpl` and `ExtViewObjectImpl` custom framework classes that were developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it

1. Open the `ExtApplicationModuleImpl.java` Application Module framework extension class into the Java editor.
2. Click on the **Override Methods...** button (the green left arrow button) and choose to override the `prepareSession(Session)` method.
3. Add the following code after the call to `super.prepareSession()`:

```
// do not clear the cache after a rollback
getDBTransaction().setClearCacheOnRollback(false);
```

4. Open the `ExtViewObjectImpl.java` View object framework extension class into the Java editor.
5. Override the `create()` method.
6. Add the following code after the call to `super.create()`:

```
// allow read-only View objects to use findByKey() methods
this.setManageRowsByKey(true);
```

7. Override the `beforeRollback()` method.
8. Add the following code before the call to `super.beforeRollback()`:

```
// check for query execution
if (isExecuted()) {
  // get the current row
  ViewRowImpl currentRow = (ViewRowImpl)getCurrentRow();
  if (currentRow != null) {
    // save the current row's key
    currentRowKeyBeforeRollback = currentRow.getKey();
    // save range start
    rangeStartBeforeRollback = getRangeStart();
    // get index of current row in range
    rangePosOfCurrentRowBeforeRollback =
```

```
                    getRangeIndexOf(currentRow);
        }
    }
```

9. Override the `afterRollback()` method and add the following code after the call to `super.afterRollback()`:

```
// check for current row key to restore
if (currentRowKeyBeforeRollback != null) {
    // execute View object's query
    executeQuery();
    // set start range
    setRangeStart(rangeStartBeforeRollback);
    // set current row in range
    findAndSetCurrentRowByKey(
            currentRowKeyBeforeRollback,
            rangePosOfCurrentRowBeforeRollback);
}

// reset
currentRowKeyBeforeRollback = null;
```

How it works

We override the Application Module `prepareSesssion()` method so we can call `setClearCacheOnRollback()` on the `Transaction` object. `prepareSession()` is called by the ADF Business Components framework the first time that an Application Module is accessed. The call to `setClearCacheOnRollback()` in `prepareSession()` tells the framework whether the Entity cache will be cleared when a rollback operation occurs. Since the framework by default clears the cache, we call `setClearCacheOnRollback()` with a `false` argument to prevent this from happening. We need to avoid clearing the cache because we will be getting information about the current row during the rollback operation.

Next we override the `create()` method in the View object framework extension class. We do this so we can call the `setManageRowsByKey()` method. This method will allow us to use framework find methods utilizing Key objects on read only View objects. By default this is not the case for read-only View objects.

Then we override the `beforeRollback()` and `afterRollback()` methods in the View object framework extension class. As theirs names indicate, they are called by the framework before and after the actual rollback operation. Let's take a look at the code in `beforeRollback()` first. In it we first get the current row by calling `getCurrentRow()`.

Then for the current row we determine the row's key, range and position of the current row within the range. This will work independently of whether range paging is used for the View object or not. Range paging should be enabled for optimization purposes for View objects that may return large rowsets. We save these values into corresponding member variables. We will be using them in `afterRollback()` to restore the current row after the rollback operation.

> Notice that we do all that after checking whether the View object query has been executed (`isExecuted()`). We do this because `beforeRollback()` may be called multiple times by the framework and we need to ensure that we retrieve the current row information only if the View object's rowset has been updated, which is the case after the query has been executed.

In `afterRollback()` we use the information obtained about the current row in `beforeRollback()` to restore the rowset currency to it. We do this by first executing the View object query – the call to `executeQuery()` – and then calling `setRangeStart()` and `findAndSetCurrentRowByKey()` to restore the range page and the row within the range to the values obtained for the current row earlier. We do all that only if we have a current row key to restore to – the check for `currentRowKeyBeforeRollback` not being `null`.

There's more

Note that for this technique to work properly on the front-end ViewController, you will need to call `setExecuteOnRollback()` on the `oracle.adf.model.binding.DCBindingContainer` object before executing the `Rollback` operation binding. By calling `setExecuteOnRollback()` on the binding container we prevent the View objects associated with the page's bindings to be executed after the rollback operation. This means that in order to call `setExecuteOnRollback()` you can't just execute the Rollback action binding directly. Rather you will have to associate the Rollback button with a Backing Bean action method, like the one shown below for instance:

```
public void rollback() {
    // get the binding container
    DCBindingContainer bindings =
                    ADFUtils.getDCBindingContainer();
    // prevent View objects from executing after rollback
    bindings.setExecuteOnRollback(false);
    // execute rollback operation
    ADFUtils.findOperation("Rollback").execute();
}
```

For testing purposes, a `ViewController` project page called `recipe3_8.jspx` has been provided that demonstrates this technique. To run it, just right-click on the **recipe3_8.jspx** page and select **Run** from the context menu.

See also

Breaking up the application in multiple workspaces, chapter 1.

Setting up BC base classes, chapter 1.

Using findAndSetCurrentRowByKey() to set the View object currency, chapter 2.

Overriding prepareSession() to do session-specific initializations, chapter 5.

Dynamically changing the View object's query WHERE clause

During the execution of the **View** object's query, the ADF Business Components framework calls a series of methods to accomplish this task. You can intervene during this process by overloading any the methods called by the framework in order to change the query about to be executed by the View object. You can also explicitly call methods in the public View object interface to accomplish this task prior to the View object's query execution. Depending on what exactly you need to change in the View object's query, the framework allows you to do the following:

- Change the query's `SELECT` clause by overriding `buildSelectClause()` or calling `setFromClause()`
- Change the query's `FROM` clause by overriding `buildFromClause()` or calling `setFromClause()`
- Change the query's `WHERE` clause via the `buildWhereClause()`, `setWhereClause()`, `addWhereClause()`, `setWhereClauseParams()` and other methods
- Change the query's `ORDER BY` clause via the `buildOrderByClause()`, `setOrderByClause()` and `addOrderByClause()` methods.

Even the complete query can also be changed by overriding the `buildQuery()` method or directly calling `setQuery()`.

This recipe shows how to override `buildWhereClause()` to alter the View object's `WHERE` clause. The use case implemented in this recipe is to limit the result set of the Employee View object by a pre-defined number of rows indicated by a custom property.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it

1. Open the `ExtViewObjectImpl.java` View object framework extension class into the Java editor.
2. Override the `buildWhereClause()` method.
3. Replace the code inside the `buildWhereClause()` with the following:

```
// framework processing
boolean appended = super.buildWhereClause(sqlBuffer,
                                noUserParams);

// check for a row count limit
String rowCountLimit =
        (String)this.getProperty(ROW_COUNT_LIMIT);

// if a row count limit exists, limit the query
if (rowCountLimit != null) {
// check to see if a WHERE clause was appended;
// if not, we will append it
if (!appended) {
        // append WHERE clause
    sqlBuffer.append(" WHERE ");
    // indicate that a where clause was added
    appended = true;
}
// a WHERE clause was appended by the framework;
// just amend it
else {
        sqlBuffer.append(" AND ");
}

// add ROWNUM limit based on the pre-defined
// custom property
sqlBuffer.append("(ROWNUM <= " + rowCountLimit + ")");
```

```
}

// a true/false indicator whether a WHERE clause was appended
// is returned to the framework
return appended;
```

How it works

We override `buildWhereClause()` in order to alter the View object's query `WHERE` clause. Specifically, we limit the result set produce by the View object's query. We do this only if the custom property called `RowCountLimit` (indicated by the constant `ROW_COUNT_LIMIT`) is defined by a View object. The value of the `RowCountLimit` indicates the number of rows that the View object's result set should be limited to.

First we call `super.buildWhereClause()` to allow the framework processing. This call will return a boolean indicator of whether a `WHERE` clause was appended to the query. This is indicated by the boolean `appended` local variable. Then we check for the existence of the `RowCountLimit` custom property. If it is defined by the specific View object, we alter or we add the `WHERE` clause depending on whether one was added or not by the framework. We make sure that we set the `appended` flag to `true` if we actually have appended the `WHERE` clause. Finally, the `appended` flag is returned back to the framework.

There's more

The use case implemented in this recipe shows one of the possible ways of limiting the View object result set. You can explore additional techniques in chapter 12.

See also

Breaking up the application in multiple workspaces, chapter 1.

Setting up BC base classes, chapter 1.

Removing a row from a rowset without deleting it from the database

There are times when you want to remove a row from the View object's query collection (the query resultset) without actually removing it from the database. The query collection - `oracle.jbo.server.QueryCollection` - gets populated each time the View object is executed - when the View object's associated query is run, and represents the query result.

While the `Row.remove()` method will remove a row from the query collection, it will also remove the underlying Entity row for an Entity-based View object, and post a deletion to the database. If your programming task requires that the row is removed from the query collection only, i.e. removing a table row in the UI without actually posting a delete to the database, use the Row method `removeFromCollection()` instead.

Note however that each time the View is re-executed the Row will show up, since it is not actually deleted from the database.

This recipe will demonstrate how to use `removeFromCollection()` by implementing a helper method in the Application Module to remove rows from the `Employees` collection.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it

1. Open the `HrComponentsAppModuleImpl.java` Application Module custom implementation class into the Java editor.
2. Add the following `removeEmployeeFromCollection()` method to it:

```
public void removeEmployeeFromCollection() {
  // get the current employee
  EmployeesRowImpl employee = (EmployeesRowImpl)(this.getEmployees().
getCurrentRow());
  // remove employee from collection
  if (employee != null) {
    employee.removeFromCollection();
  }
}
```

3. Expose the `removeEmployeeFromCollection()` method to the Application Module's client interface using the **Edit application module client interface** button (the pen icon) in the **Client Interface** section of the Application Module's **Java** page.

How it works

We implemented `removeEmployeeFromCollection()` and exposed it to the Application Module's client interface. By doing so, we will be able to call this method using the **Oracle ADF Model Tester** for testing purposes. In it we first get the current employee by calling `getCurrentRow()` on the `Employees` View object instance. We retrieve the `Employees` View object instance by calling the `getEmployees()` getter method. Then we call `removeFromCollection()` on the current employee View row to remove it from the `Employees` View object rowset. This has the effect of removing the employee from the rowset without removing from the database. A subsequent re-query of the `Employees` View object will retrieve those `Employee` rows that were removed earlier.

There's more

Note that there is a quick and easy way to remove all rows from the View object's rowset by calling the View object `executeEmptyrowset()` method. This method re-executes the View object's query ensuring that the query will return no rows, however it achieves this in an efficient programmatic way without actually sending the query to the database for execution. Calling `executeEmptyrowset()` marks the query's `isExecuted` flag to `true`, which means that it will not be re-executed upon referencing a View object attribute.

See also

Overriding remove() to delete associated children entities, chapter 2.

4

Important Contributors: List of Values, Bind Variables, View Criteria

In this chapter, we will cover:

- Setting up multiple LOVs using a switcher attribute
- Setting up cascading LOVs
- Creating static LOVs
- Overriding `bindParametersForCollection()` to set a View object bind variable
- Creating View Criteria programmatically
- Clearing the values of bind variables associated with the View Criteria
- Searching case-insensitively using View Criteria

Introduction

List of Values (LOVs), **bind variables** and **View Criteria** are essential elements related to **View** objects. They allow further refinements to the View object's query (bind variables and view criteria) and make the development of the front end user interface easier when dealing with list controls (LOVs) and Query-by-Example (view criteria) components.

Many of the user interface aspects that deal with list controls and Query-by-Example components can be pre-defined to a set of default values via the **UI Hints** sections and pages in JDeveloper, thus providing a standard UI behavior. For LOVs for instance, the default UI list component, the attributes to be displayed, whether "No Selection" items will be included in the list, and others can be pre-defined for the UI list component. These defaults can be overridden as needed for specific LOVs.

Bind variables and View Criteria, when used in conjunction or separately, allow you to dynamically alter the View object query based on certain conditions. Furthermore, using bind variables as placeholders in the query, allows the database to effectively reuse the same parsed query for multiple executions without the need to re-parse it.

In this chapter we will examine how the **ADF Business Components** framework API programmatically and JDeveloper declaratively provide support for these components.

Setting up multiple LOVs using a switcher attribute

List of Values (LOV) when enabled for View object attributes, greatly simplify the effort involved in utilizing list controls in the front end user interface. LOV-enabling a View object attribute is a straightforward task, done declaratively in JDeveloper. Moreover, the ADF Business Components framework allows you to define multiple LOVs for the same attribute. In this case, in order to differentiate among the LOVs, a separate attribute called List of Values **Switcher** is used. The differentiation is usually done based on some data value.

This recipe shows how to enable multiple LOVs for a View object attribute and how to use an LOV Switcher to switch among the LOVs. The scenario that we will consider is the following: depending on the employee's job we will associate a different LOV to an `Employees` transient attribute.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

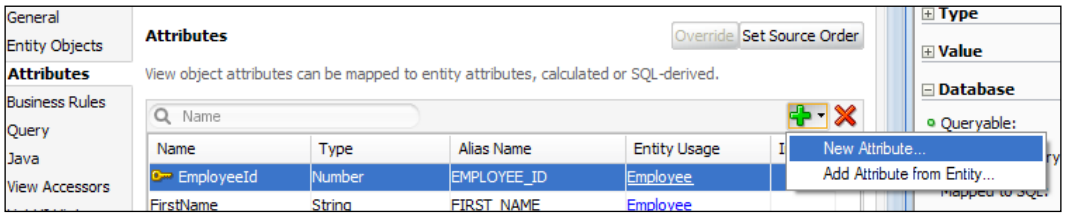
How to do it...

1. Create a new **read-only** View object called `DepartmentsLov` by right-clicking on a package of the `HRComponents` Business Components project in the **Application Navigator** and selecting **New View Object...**.
2. Base the View object on this SQL query:

   ```
   SELECT DEPARTMENT_ID, DEPARTMENT_NAME FROM DEPARTMENTS
   ```
3. In addition, set the `DepartmentId` attribute as the **Key Attribute**. Do not add the `DepartmentsLov` View object to the **Application Module**.
4. Repeat the steps above to create another read-only View object called `JobsLov` based on this SQL query:

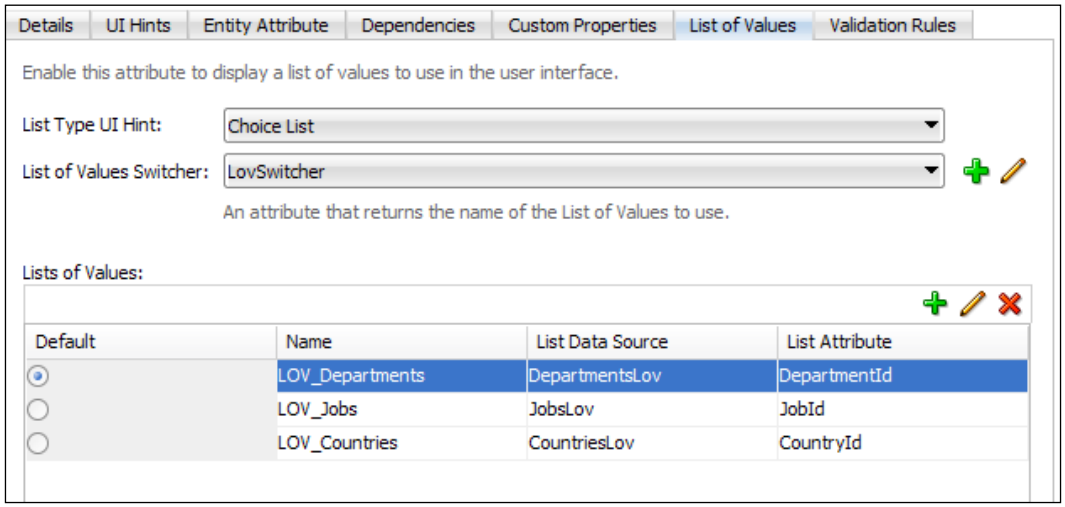
   ```
   SELECT JOB_ID, JOB_TITLE FROM JOBS
   ```
5. In this case, set the `JobId` attribute as the key attribute.
6. Create yet another read-only View object called `CountriesLov` based on the SQL query

   ```
   SELECT COUNTRY_ID, COUNTRY_NAME FROM COUNTRIES
   ```
7. Define `CountryId` as the key attribute.
8. Now, open the `Employees` View object definition and go to the **Attributes** tab. Create a new attribute called `LovAttrib` by selecting **New Attribute...** from the context menu.



9. In the **Details** tab, change the attribute **Updatable** value to **Always**.
10. Switch to the **List of Values** tab and click on the **Add list of values** button (the green plus sing icon).
11. On the **Create List of Values** dialog, enter `LOV_Departments` for the **List of Values Name**.
12. Click on the **Create new view accessor** button (the green plus sing icon) to create a new **List Data Source**. This will bring up the **View Accessors** dialog.
13. On the **View Accessors** dialog, locate the **DepartmentsLov**, shuttle it to the **View Accessors** list on the right and click **OK**.

14. Select **DepartmentName** for the **List Attribute**. In the **List Return Values** section, the `DepartmentName` view accessor attribute should be associated with the `LovAttrib`. Click **OK** when done.
15. Repeat the steps above to add another LOV, called `LOV_Jobs`. Add the **JobsLov** as a view accessor, as you did in the previous steps, and select it as the **List Data Source**. Use the **JobTitle** attribute as the **List Attribute**.
16. Add one more LOV called `LOV_Countries` by repeating the steps above. Add **CountriesLov** as a view accessor and select it as the **List Data Source**. For the **List Attribute** use the **CountryName** attribute.
17. While at the **List of Values** tab, click on the **Create new attribute** button (the green plus sign icon) next to the **List of Values Switcher** field to create a switcher attribute. Call the attribute **LovSwitcher**. Now, the **List of Values** tab should look like this:



18. Lastly, select the **LovSwitcher** attribute and go to the **Details** tab. Click on the **Expression** radio button in the **Default Value** section, and then on the **Edit value** button (the pen icon). Enter the following expression:

```
if(JobId == 'SA_REP'){
   return 'LOV_Countries'
} else if(JobId == 'ST_CLERK'){
   return 'LOV_Jobs'
} else if(JobId == 'ST_MAN'){
   return 'LOV_Departments'
} else {
   return null;
}
```

How it works...

In steps 1 through 4 we created three read-only View objects, one for each LOV. We then created a new transient attribute, called `LovAttrib`, for the `Employees` View object (step 5). This is the attribute that we will use to add the three LOVs. We added the LOVs by switching to the **List of Values** tab. In steps 6 through 13 we added the appropriate View objects as accessors to the `Employees` View object and associated the accessors with the LOV **List Data Source**. The List Data Source indicates the view accessor that provides the list data at runtime. In each case we also specified a view accessor attribute as the **List Attribute**. This is the view accessor attribute that supplies the data value to the `LovAttrib` attribute. You can specify additional view accessor attributes to supply data for other `Employees` View object attributes in the **List Return Values** section of the **Create/Edit List of Values** dialog. In step 14 we created a new transient attribute, called `LovSwitcher`, to act as the LOV Switcher. Finally in step 15 we supplied the default value to the LOV switcher `LovSwitcher` attribute in the form of a **Groovy** expression. In the Groovy expression we examine the value of the `JobId` attribute and based on its value we assign (by returning the LOV name) the appropriate LOV to the `LovSwitcher` attribute. Since the `LovSwicher` attribute is used as an LOV Switcher, the result is that the appropriate LOV is associated with the `LovAttrib` attribute.

There's more...

For Entity-based View objects, you can LOV-enable an attribute using a list data source that is based on an Entity object view accessor. This way a single Entity-based view accessor is used for all View objects referencing the Entity object and is applied to each instance of the LOV. Note however that while this will work fine on create and edit forms it will not work for search forms. In this case, the LOV must be based on a view accessor defined at View object level. For a use case where two LOVs are defined on an attribute one based on an Entity object accessor and another on a View object accessor you can use a Switcher attribute that differentiates among the two LOVs based on this expression:

```
adf.isCriteriaRow ?  "LOV_ViewObject_accessor" : "LOV_EntityObject_
accessor"
```

See also

Overriding remove() to delete associated children entities, chapter 2.

Setting up cascading LOVs

Cascading LOVs refer to two or more LOVs where the possible values of one LOV depend on specific attribute values defined in another LOV. These controlling attributes are used in order to filter the resultset produced by the controlled LOVs. The filtering is usually accomplished by adding named view criteria, based on bind variables, to the controlled LOV list data source, the view accessor. It can also be done by directly modifying the controlled LOV view accessor query adding the controlling attributes as bind variable placeholders in its query.

For this recipe we will create two LOVs, one for the `DEPARTMENTS` table and another for the `EMPLOYEES` table so that when a department is selected only the employees of that particular department are shown.

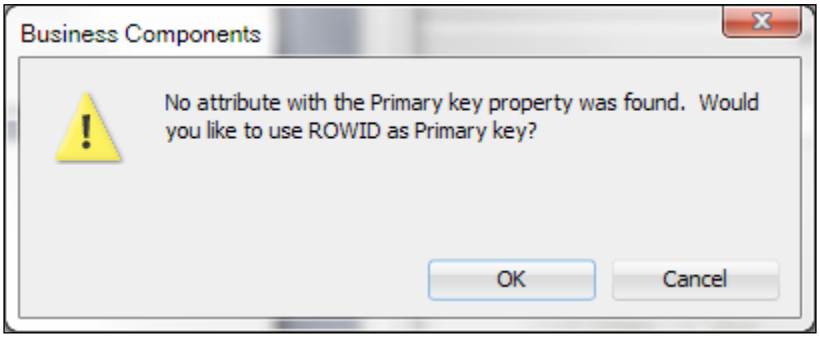
Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema. You will also need an additional table added to the HR schema called `CASCADING_LOVS`. You can create it by running the following SQL command:

```
CREATE TABLE CASCADING_LOVS (EMPLOYEE_ID NUMBER(6), DEPARTMENT_ID
NUMBER(4));
```

How to do it...

1. Create a new Entity object based on the `CASCADING_LOVS` table.
2. Since the `CASCADING_LOVS` table does not define a primary key, the **Create Entity Object** wizard will ask you if you want to create an attribute with a primary key property based on the `ROWID`. Select **OK**.



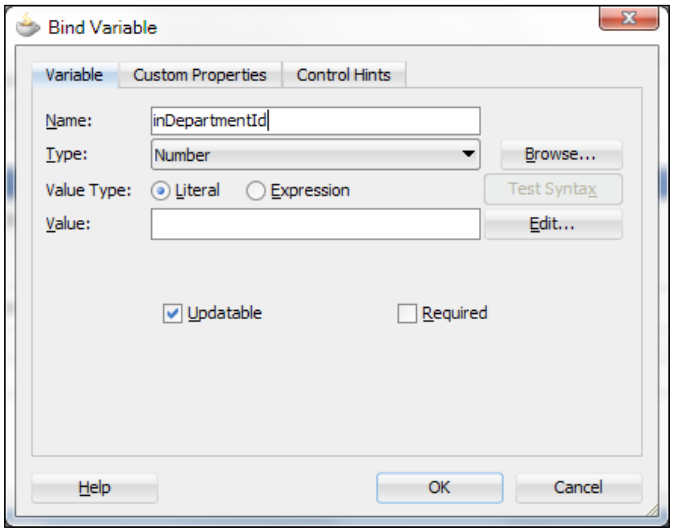
3. Create a View object based on the `CascadingLovs` Entity object.
4. Create a read-only View object called `DepartmentsLov` based on the following query:

   ```
   SELECT DEPARTMENT_ID, DEPARTMENT_NAME FROM DEPARTMENTS
   ```

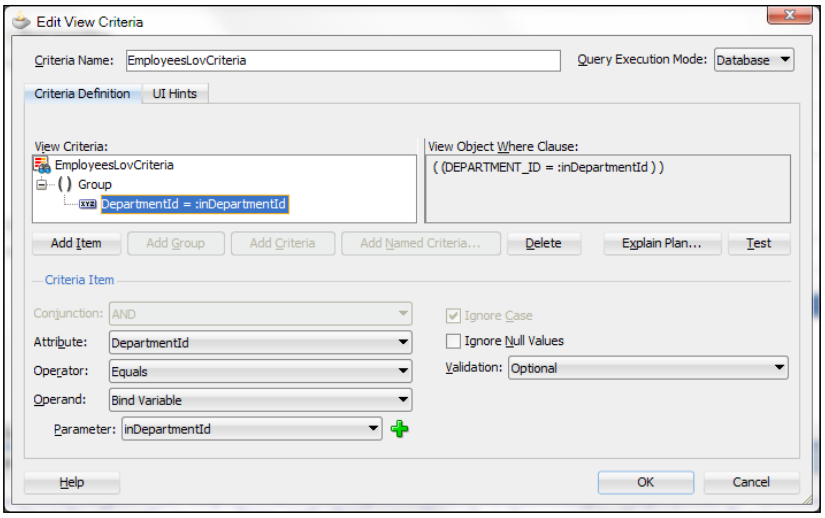
5. Create another read-only View object called `EmployeesLov` based on the following query:

   ```
   SELECT DEPARTMENT_ID, EMPLOYEE_ID, FIRST_NAME, LAST_NAME
     FROM EMPLOYEES
   ```

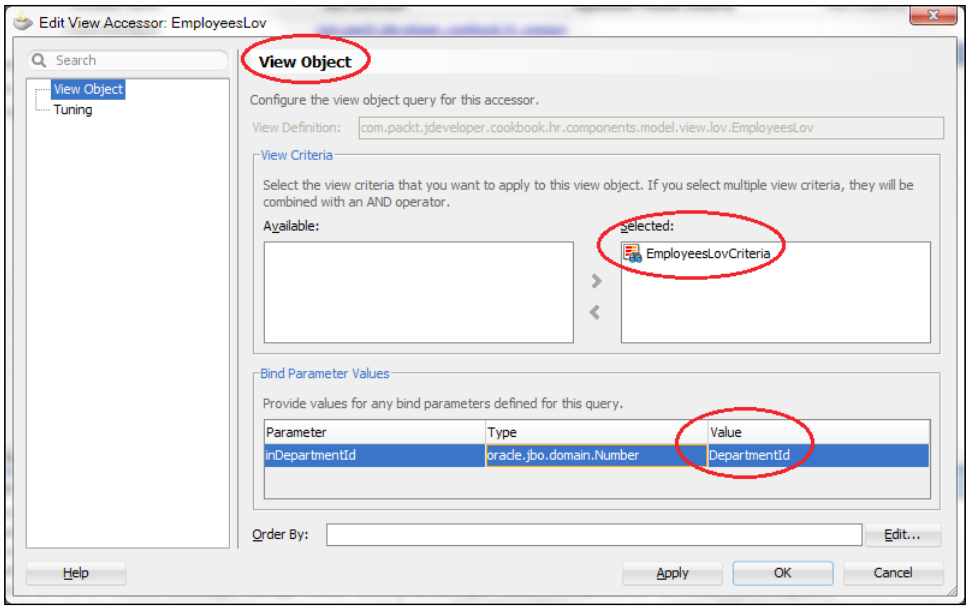
6. In the **Query** section, use the **Create new bind variable** button (the green plus sign icon) to add a bind variable to the `EmployeesLov`. Call the bind variable `inDepartmentId` of **Type** `Number` and ensure that the **Required** check box is unchecked.



7. While in the **Query** section, click on the **Create new view criteria** button (the green plus sign icon) to add view criteria to the `EmployeesLov` View object. Click the **Add Item** button and select **DepartmentId** for the **Attribute**. For the **Operator** select **Equals** and for the **Operand** select **Bind Variable**. Ensure that you select the **inDepartmentId** bind variable that you created in the step above from the **Parameter** combo. Make sure the **Ignore Null Values** checkbox is unselected and that the **Validation** selection is **Optional**.



8. Back to the `CascadingLovs` View object, go to the **Attributes** section and select the **DepartmentId** attribute.
9. Click on the **List of Values** tab and then on the **Add list of values** button (the green plus sign icon).
10. On the **Create List of Values** dialog click on the **Create new view accessor** button (the green plus sign icon) next to the **List Data Source** combo and add `DepartmentsLov` view accessor.
11. Select **DepartmentId** for the **List Attribute**.
12. While on the **Create List of Values** dialog, click on the **UI Hints** tab and in the **Display Attributes** section shuttle the **DepartmentName** attribute from the **Available** list to the **Selected** list. Click **OK**.
13. Repeat steps 8 through 12 to add an LOV for the `EmployeeId` attribute. Use the `EmployeesLov` as the list data source and `EmployeeId` as the list attribute. For the display attributes in the **UI Hints** tab use the `FirstName` and `LastName` attributes.
14. In the **View Accessors** section select the **EmployeesLov** view accessor (do not click on the **View Definition** link) and click on the **Edit selected View Accessor** button (the pen icon).
15. In the **Edit View Accessor** dialog, select the **View Object** section and shuttle the **EmployeesLovCriteria** from the **Available** list to the Selected list. Also, for the **inDepartmentId** parameter in the **Bind Parameter Values** section enter **DepartmentId** in the **Value** field and click **OK**.



16. Double-click on the **HrComponentsAppModule** Application Module in the **Application Navigator** to open the application module definition.
17. Go to the **Data Model** section, select the **CascadingLovs** View object and shuttle it from the **Available View Objects** list to the **Data Model**.

How it works...

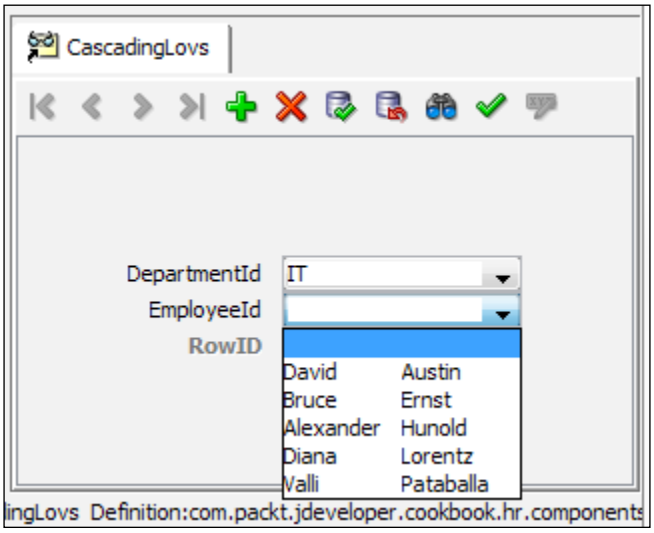
To demonstrate this recipe we created a new table in the `HR` schema called `CASCADING_LOVS`. This table has two columns an `EMPLOYEE_ID` and a `DEPARTMENT_ID`. The table does not have a primary key constraint, so we will be able to freely add records to it. Based on this table, we created an Entity object called `CascadingLovs` (step 1). Since we did not indicated a primary key for the `CASCADING_LOVS` table, the framework asked us to indicate a primary key attribute (step 2). We did so by creating a key attribute called `RowID` based on the row's `ROWID`. Then we proceeded to create a View object called `CascadingLovs` based on the `CascadingLovs` Entity object (step 3).

In order to setup LOVs for the `DepartmentId` and `EmployeeId` attributes, we had to create the LOV accessor View objects, namely the `DepartmentsLov` and the `EmployeesLov` View objects (steps 4 and 5). We also added named view criteria to the `EmployeesLov` (steps 6 and 7) based on the `inDepartmentId` bind variable. This way we will be able to control the resultset produced by the `EmployeesLov` based on the department id value.

In steps 8 through 13, we proceeded by LOV-enabling the `DepartmentId` and `EmployeeId` attributes.

The important *glue* work was done in steps 14 and 15. In these steps we edited the `EmployeesLov` view accessor and we declaratively applied the `EmployeesLovCriteria` on the accessor. We also provided a value for the `inDepartmentId` bind variable using the expression `DepartmentId`, which indicates the value of the `DepartmentId` attribute at runtime. By doing so, we have set the controlling variable's value, the `inDepartmentId` bind variable, using the value provided by the `DepartmentsLov` data source, i.e. the `DepartmentId`.

Finally, in steps 16 and 17, we added the `CascadingLovs` View object to the Application Module's data model, so that we may be able to test it using the ADF Model Tester. When running the ADF Model Tester, notice how the employees list is controlled by the selected department.



There's more...

For the cascading LOVs to work properly on the front end user interface, you need to make sure that the `autoSubmit` property is set to `true` for the controlling LOV UI component. This will ensure that, upon selection, the controlling attribute's value is submitted to the server. The UI component's `autoSubmit` property can also be set to a default value by setting the attribute's **Auto Submit** property at the Business Component level. This can be done in the View object's **Attributes > UI Hints** tab.

Also, note the behavior of the controlled LOV based on the View criteria **Ignore Null Values** setting. When this checkbox is selected, null values for the criteria item will be ignored and the resultset will not be filtered yielding all possible employee rows. In this case, the `EmployeesLov` View object's `WHERE` clause is amended by adding `OR (:inDepartmentId is null)` to the query. If the **Ignore Null Values** checkbox is not selected, then null values for the criteria item are not ignored, yielding no employees rows.

See also

Overriding remove() to delete associated children entities, chapter 2.

Creating static LOVs

A static LOV is produced by basing its list data source view accessor on a static View object, a View object that is that uses a static list as its data source. A static list is a list of constant data that you either enter manually or import from a text file using JDeveloper. A static list could also be produced by basing the View object on a query that generates static data.

In this recipe we will create a static View object called `ColorLov` and use it as an LOV list data source to LOV-enable a transient attribute of the `Employees` View object.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Create a new View object using the **Create View Object** wizard.
2. In the **Name** page, enter **ColorsLov** for the name of the View object and select **Static list** for the **Data Source**.
3. In the **Attributes** page, click on the **New...** button. Create an attribute called **ColorDesc**.
4. In the **Attribute Settings** page, select **Key Attribute** for the `ColorDesc` attribute.
5. In the **Static List** page, use the **Add Row** button (the green plus sign icon) to add the following data: `Black`, `Blue`, `Green`, `Red`, `White` and `Yellow`.
6. Click Finish to complete the creation of the `ColorsLov` View object.

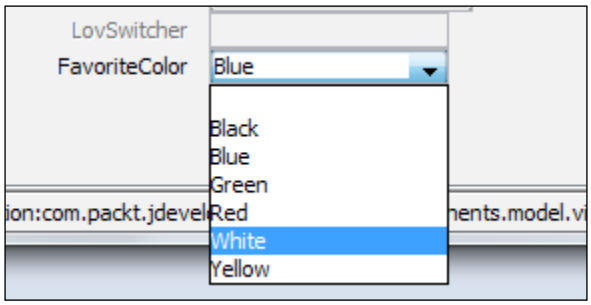
7. Add a new transient attribute to the `Employees` View object called `FavoriteColor`.
8. In the **Attributes > Details** tab for the `FavoriteColor` attribute, ensure that the **Updatable** property is set to **Always**.
9. Click on the **List of Values** tab and add an LOV called `LOV_FavoriteColor`. For the LOV **List Data Source** use the **Create new view accessor** button (the green plus sign icon) and select the **ColorsLov** static View object.
10. For the LOV List Attribute, select the **ColorDesc** attribute.

How it works...

In steps 1 through 6 we went through the process of creating a View object, called `ColorsLov`, which uses a static list as its data source. We have indicated that the View object has one attribute called `ColorDesc` (step 3) and we have indicated that attribute as a key attribute (step 4). Notice in step 5 how the Create View object wizard allows you to manually enter the static data. In the same **Static List** page, the wizard allows you to import data from a file in a comma-separated-values (CSV) format.

In order to test the static LOV, we added a transient variable called `FavoriteColor` to the `Employees` View object (step 7-8) and we LOV-enabled the attribute using the `ColorsLov` as the list data source view accessor (steps 9-10).

When we test the Application Module with the ADF Model Tester the `FavoriteColor` attribute is indeed populated by the static values we have entered for the `ColorsLov` View object.



There's more...

Notice that the static data that is entered for the static View object is saved on a resource bundle. This allows you to localize the data as needed.

Also note that in some cases where localization of static data is not needed, a read-only View object that is based on a query producing static data can simulate a static View object. For instance consider the read-only View object that is based on the following query:

```
SELECT 'Black' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Blue' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Green' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Red' AS COLOR_DESC FROM DUAL
UNION
SELECT 'White' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Yellow' AS COLOR_DESC FROM DUAL;
```

It can be used as a list data source for the `FavoriteColor` LOV producing the same results.

See also

Overriding remove() to delete associated children entities, chapter 2.

Overriding bindParametersForCollection() to set a View object bind variable

There are times when you need to programmatically set the value of a bind variable used in the View object query. One way to accomplish this task is by overriding the View object method `bindParametersForCollection()` and explicitly specify the value for the particular bind variable.

This recipe will show how to provide a default value for a bind variable used in the View object query if a value has not already been specified for it.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. Additional functionality will be added to the `ExtViewObjectImpl` and ExtApplicationModuleImpl custom framework classes that were developed in the *Setting up BC base classes* recipe in chapter 1.

This recipe is also using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

Moreover, we will modify `EmployeeCount` View object, which was introduced in the *Using a method validator based on a View accessor* recipe in chapter 2.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl` View object framework extension class and add the following method to it:

```
protected void setBindVariableValue(
                Object[] bindVariables,
                String name, Object value) {
  // iterate all bind variables
  for (Object bindVariable : bindVariables) {
    // check for the specific bind variable name
    if (((Object[])bindVariable)[0].toString().equals(name)) {
      // set the bind variable's new value
      ((Object[])bindVariable)[1] = value;
      return;
    }
  }
}
```

3. Open the `ExtApplicationModuleImpl` Application Module framework extension class and add the following method to it:

```
public Object getCustomData(String key) {
  // base class returns no custom data
  return null;
}
```

4. Deploy the shared components workspace into an ADF Library JAR.
5. Open the `HRComponents` workspace.
6. Open the `HrComponentsAppModuleImpl` custom Application Module implementation class and override the `getCustomData()` method. In it, replace the return `super.getCustomData()` with the following:

```
return DEFAULT_DEPARTMENT_ID_KEY.equals(key)
            ? DEFAULT_DEPARTMENT_ID
            : null;
```

7. Open the `EmployeeCount` View object, go to the **Java** page and create a custom View object class.
8. Open the `EmployeeCountImpl` custom View object implementation class and override the `bindParametersForCollection()` method. Add the following code to it before the call to `super.bindParametersForCollection()`:

```
// if bind variable value has not been provided,
// provide a default setting
if (this.getDepartmentId() == null) {
   // get default department id
   Number departmentId =
    ((Number)((ExtApplicationModuleImpl)
    this.getApplicationModule()).getCustomData(
                DEFAULT_DEPARTMENT_ID_KEY));
   // set bind variable right on the query to
   // default variable
   super.setBindVariableValue(object, "DepartmentId",
                departmentId.toString());
   // set bind variable on View object as well,
   //to be available for this time forward
   this.setDepartmentId(departmentId);
}
```

9. Finally, add the `EmployeeCount` View object to the `HrComponentsAppModule` Application Module data model.

How it works...

In recipe *Using a method validator based on a View accessor* in chapter 2, we created a View object called `EmployeeCount`, which we used in order to get the employee count for specific departments. The View object was based on the following query:

```
SELECT COUNT(*) AS EMPLOYEE_COUNT
      FROM EMPLOYEES
      WHERE DEPARTMENT_ID = :DepartmentId
```

The `EmployeeCount` View object was added as a view accessor to the Employee Entity object and it was used in a method validator. In the method validator, a value was supplied programmatically for the `DepartmentId` bind variable prior to executing the `EmployeeCount` query.

In this recipe we have used the same `EmployeeCount` View object to demonstrate how to supply a default value for the `DepartmentId` bind variable. We did this by creating a custom View object implementation class (step 7) and then by overriding its `bindParametersForCollection()` method (step 8). This method is called by the ADF Business Components framework to allow you to set values for the View object query's bind variables. When the framework calls `bindParametersForCollection()`, it supplies among the other parameters an `Object[]` which contains the query's bind variables (the `bindVariables` parameter). We set a default value of the `DepartmentId` bind variable by calling `super.setBindVariableValue()`. This is the helper method that we added to the View object framework extension class in step 2. In the `setBindVariableValue()` method we iterate over the query's bind variables until we find the one we are looking for and once we find it, we set its new value.

Note that in `bindParametersForCollection()` we have called `getApplicationModule()` to get the Application Module for this `EmployeeCount` View object instance (we have added the `EmployeeCount` View object to the `HrComponentsAppModule` data model in step 9). `getApplicationModule()` returns an `oracle.jbo.ApplicationModule` interface, which we have casted it to an `ExtApplicationModuleImpl`. As a recommended practice, you should not be accessing specific Application Modules from within your View objects. In this case we relaxed the rule a bit by casting the `oracle.jbo.ApplicationModule` interface returned by the `getApplicationModule()` method to our Application Module framework extension class. We have then called `getCustomData()`, which we overrode in step 6, to get the default `DepartmentId` value. It is this default value (stored in variable `departmentId`) that we supply when calling `super.setBindVariableValue()`.

There's more...

Although the `ViewObjectImpl executeQueryForCollection()` method can be used to set the View object's query bind variables values, do not use this method because the framework will never invoke it when `getEstimatedRowCount()` is called to identify the resultset's row count. If you do, `getEstimatedRowCount()` will not produce the correct row count if you are altering the query by supplying values to the query's bind variables.

Also note that with the 11.1.1.5.0 (PS4) release, a new `ViewObjectImpl` method called `prepareRowSetForQuery()` was introduced that can be used to set the query's bind parameters values. The code below illustrates how to use it to set a value for the `DepartmentId` bind variable in this recipe:

```
public void prepareRowSetForQuery(ViewRowSetImpl vrsImpl) {
    // get default departmentId value as before
    Number departmentId =
    vrsImpl.ensureVariableManager().setVariableValue(
```

```
                        "DepartmentId", departmentId);
        super.prepareRowSetForQuery(vrsImpl);
    }
```

Both `bindParametersForCollection()` and `prepareRowSetForQuery()` are valid choices for setting the View object's query bind variables values. If for some reason both of them are overridden, note that the framework will first call `prepareRowSetForQuery()` and then `bindParametersForCollection()`.

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using a method validator based on a View accessor, chapter 2.

Creating View Criteria programmatically

View Criteria augment the View object's `WHERE` clause by appending additional query conditions to it. They work in conjunction with the `af:query` ADF Faces UI component to provide Query-by-Example support to the front end user interface. View criteria can be created declaratively in JDeveloper in the **Query** section of the View object definition by clicking on the **Create new view criteria** button (the green plus sign icon) in the **View Criteria** section. Programmatically, the ADF Business Components API supports the manipulation of View Criteria among others via the `ViewCriteria`, `ViewCriteriaRow` and `ViewCriteriaItem` classes and through a number of methods implemented in the `ViewObjectImpl` class.

In this recipe we will see how to create View Criteria programmatically. The use case will be to dynamically amend the `Employees` View object query by adding to it View Criteria. The values that we will use for the view criteria items will be obtained from the resultset of yet another View object.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1. For testing purposes, we will be using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the `searchUsingAdditionalCriteria()` method below:

```
public void searchUsingAdditionalCriteria(
         ViewObject providerViewObject,
         String[] attribNames) {
    // create the criteria
    ViewCriteria vc = this.createViewCriteria();
    // set the view criteria name
    vc.setName("searchUsingAdditionalCriteria");
    // AND with previous criteria
    vc.setConjunction(ViewCriteriaComponent.VC_CONJ_AND);

    // get criteria item data from the provider
    // View object
    RowSetIterator it =
            providerViewObject.createRowSetIterator(null);
    it.reset();
    while (it.hasNext()) {
        Row providerRow = it.next();
        // add a criteria item for each attribute
        for (String attribName : attribNames) {
            try {
                // create the criteria item
                ViewCriteriaRow vcRow =
                              vc.createViewCriteriaRow();
                // set the criteria item value
                vcRow.setAttribute(attribName,
                    providerRow.getAttribute(attribName));
                // add criteria item to the view criteria
                vc.insertRow(vcRow);
            } catch (JboException e) {
                LOGGER.severe(e);
            }
        }
    }

    // done with iterating provider View object
    it.closeRowSetIterator();

    // apply the criteria to this View object
```

```
        this.applyViewCriteria(vc);

        // execute the View object's query
        this.executeQuery();
    }
```

3. For logging purposes, add an `ADFLogger LOGGER` to the same class as it is shown below:

```
private static ADFLogger LOGGER = ADFLogger.createADFLogger(
                                    ExtViewObjectImpl.class);
```

4. Deploy the shared components projects into an **ADF Library JAR**.
5. For testing purposes, open the `HRComponents` workspace and add the following `searchEmployeesUsingAdditionalCriteria()` method to the `HrComponentsAppModuleImpl` custom Application Module implementation class:

```
public void searchEmployeesUsingAdditionalCriteria() {
  // invoke searchUsingAdditionalCriteria() to create
  // resultset based on View criteria item
  // data obtained from another View object's rowset
((ExtViewObjectImpl)this.getEmployees())
.searchUsingAdditionalCriteria(this.getCascadingLovs(),
                    new String[] { "EmployeeId" });
  }
```

How it works...

In step 1, we created a method called `searchUsingAdditionalCriteria()` in the shared components workspace `ExtViewObjectImpl` View object framework extension class, to allow View objects to alter their queries by dynamically creating and applying view criteria. The data for the view criteria items are provided by another View object. The method accepts the View object that will provide the view criteria item data values (`providerViewObject`) and an array of attribute names (`attribNames`) that is used to create the view criteria items and also retrieve the data from the provider View object. The lines below show how this method is called from the `searchEmployeesUsingAdditionalCriteria()` method that we added to the application module implementation class in step 5:

```
((ExtViewObjectImpl)this.getEmployees())
.searchUsingAdditionalCriteria(this.getCascadingLovs(),
            new String[] { "EmployeeId" });
```

As you can see, we have used the View object returned by `this.getCascadingLovs()` in order to obtain the view criteria item values. A single criteria item based on the employee id was also used.

In the `searchUsingAdditionalCriteria()`, we first called `createViewCriteria()` to create view criteria for the View object. This method returns an `oracle.jbo.ViewCriteria` object representing the view criteria. This object can be used subsequently to add criteria items onto it. Then we called `setConjunction()` on the view criteria to set the conjunction operator (`OR, AND, UNION, NOT`). The conjunction operator is used to combine multiple criteria when nested view criteria are used by the View object. This could be the case if the View object has defined additional view criteria. We have used an AND conjunction in this example (the `ViewCriteriaComponent.VC_CONJ_AND` constant), although this can easily be changed by passing the conjunction as another parameter to `searchUsingAdditionalCriteria()`.

In order to retrieve the view criteria item data, we iterated the provider View object, and for each row of data we called `createViewCriteriaRow()` on the view criteria to create the criteria row. This method returns an `oracle.jbo.ViewCriteriaRow` object representing a criteria row. We added the view criteria row data by calling `setAttribute()` on the newly created criteria row, and we added the criteria row to the view criteria by calling `insertRow()` on the view criteria, passing the criteria row as an argument.

Once, all criteria items have been setup, we called `applyViewCriteria()` on the View object, specifying the newly created view criteria, to set the View object's view criteria, and called `executeQuery()` to execute the View object's query based on the applied view criteria. The resultset produced matches the applied criteria.

There's more...

Note what happens when the framework executes the View object query after applying the view criteria programmatically. Adding two criteria rows for example, will append the following to the query's `WHERE` clause:

```
( ( (Employee.EMPLOYEE_ID = :vc_temp_1 ) )  OR ( (Employee.EMPLOYEE_ID
= :vc_temp_2 ) ) )
```

As you can see, the framework amends the query using temporary bind variables (`vc_temp_1`, `vc_temp_2`, and so on) for each criteria row.

Also note the following:

- Calling `applyViewCriteria()` on the View object, erases any previously applied criteria. In order to preserve any previously applied criteria, the framework provides another version of `applyViewCriteria()` that accepts an extra `bAppend` boolean parameter. Based on the value of `bAppend`, the newly applied criteria can be appended to existing criteria, if any. Moreover, to apply multiple criteria at once, the framework provides the `setApplyViewCriteriaNames()` method. This method accepts a `java.lang.String` array of the criteria names to apply and by default `AND`s the criteria applied.

- The usage of `ViewCriteriaRow setAttribute()` method to set the view criteria item value. The way it was used in this recipe, it has setup an equality operation for the criterion, i.e. `EmployeeId = someValue`. In order to specify a different operation for the criterion item, you must specify the operation as part of the setAttribute() method call. For example, `vcRow.setAttribute("EmployeeId","< 150"),vcRow.setAttribute("EmployeeId","IN (100,200,201)")` and so on.
- Finally, note that you can setup the view criteria item via the `setOperator()` and `setValue()` methods supplied by the `ViewCriteriaItem` class. You will need to call `ensureCriteriaItem()` on the criteria row in order to get access to a `ViewCriteriaItem`. Here is an example:

```
// get the criteria item from the criteria row
ViewCriteriaItem criteriaItem =
                 vcRow.ensureCriteriaItem("EmployeeId");
// set the criteria item operator
criteriaItem.setOperator("<");
// set the criteria item value
criteriaItem.getValues().get(0).setValue(new Integer(150));
```

See also

Setting up *BC base classes, chapter 1.*

Overriding remove() to delete associated children entities, chapter 2.

Clearing the values of bind variables associated with the View Criteria

This recipe shows you how to clear the values associated with bind variables used as operands by the view criteria items for specific View object View Criteria. It implements a method called `clearCriteriaVariableValues()` in the View object framework extension class which becomes available for all View objects to call. Bind variables are associated as operands for criteria items during the process of creating the View object's in the **Create View Criteria** dialog. Also, as we have seen in the *Creating View Criteria programmatically* recipe, bind variables are generated automatically by the framework when programmatically creating view criteria.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the `clearCriteriaValues()` method below:

```
public void clearCriteriaVariableValues(
                          String[] criteriaNames) {
    // iterate all view criteria names
    for (String criteriaName : criteriaNames) {
        // get the view criteria
        ViewCriteria vc =
            this.getViewCriteria(criteriaName);
        if (vc != null) {
            VariableValueManager vvm =
                   vc.ensureVariableManager();
            Variable[] variables = vvm.getVariables();
            for (Variable var: variables) {
                vvm.setVariableValue(var, null);
            }
        }
    }
}
```

How it works...

The `clearCriteriaVariableValues()` is added to the `ExtViewObjectImpl` View object framework extension class, thus making it available for all View objects to call it. The method accepts a `java.lang.String` array of view criteria names (`criteriaNames`) and iterates over them getting for each the associated view criteria. It then calls `ensureVariableManager()` on the view criteria to retrieve the bind variables manager, an `oracle.jbo.VariableValueManager` interface which is implemented by the framework to manage named variables.

The bind variables are retrieved by calling `getVariables()` on the variable manager. This method returns an array of objects implementing the `oracle.jbo.Variable` interface, the actual bind variables. Finally, we iterate over the bind variables used by the view criteria, setting their values to null by calling `setVariableValue()` for each one of them.

There's more...

Note that the technique used in the recipe does not remove the view criteria associated with the view object, it simply resets the values of the bind variables associated with the view criteria. In order to completely remove the view criteria associated with a particular view object, call the `ViewObjectImpl` method `removeViewCriteria()`. This method first un-applies the specific view criteria and then completely removes them from the view object. If you want to un-apply the view criteria without removing them from the View object, use the `removeApplyViewCriteriaName()` method. Furthermore, you can also clear all the View object view criteria in effect by calling `applyViewCriteria()` on the view object and specifying `null` for the view criteria name. Finally, to clear any view criteria in effect, you can also delete all the view criteria rows from it using the `remove()` method.

See also

Setting up *BC base classes, chapter 1.*

Searching case-insensitively using View Criteria

This recipe shows you a technique that you can use to handle case-insensitive (or case-sensitive for that matter) searching for strings, when using View Criteria for a View object. The framework provides various methods, such as `setUpperColumns()` and `isUpperColumns()` for instance, at various View Criteria levels (`ViewCriteria`, `ViewCriteriaRow` and `ViewCriteriaItem`) that can be used to construct generic helper methods to handle case searching.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the `setViewCriteriaCaseInsensitive()` method below:

```
public void setViewCriteriaCaseInsensitive(
                    boolean bCaseInsensitive) {
  // get all View Crietria managed by this View object
  ViewCriteria[] vcList = getAllViewCriterias();
  if (vcList != null) {
    // iterate over all view criteria
    for (ViewCriteria vc : vcList) {
      // set case-insensitive or case-sensitive as
      // indicated by the bCaseInsensitive parameter
      if (vc.isUpperColumns() != bCaseInsensitive)
        vc.setUpperColumns(bCaseInsensitive);
    }
  }
}
```

How it works...

We have added the `setViewCriteriaCaseInsensitive()` method in the `ExtViewObjectImpl` View object framework extension class to allow all View objects to call it in order to enable or disable case-insensitive search based on view criteria managed by the specific View object. The boolean parameter `bCaseInsensitive` indicates whether case-insensitive search is to be enabled for the view criteria.

The method gets access to all view criteria managed by the specific View object by calling `getAllViewCriterias()`. This framework method returns an `oracle.jbo.ViewCriteria` array containing all view criteria (both applied and unapplied) that are managed by the View object. It then iterates over them, checking in each iteration whether the current case-insensitive setting, obtained by calling `isUpperColumns()`, differs from the desired setting indicated by `bCaseInsensitive`. If this is the case, case-insensitivity is set (or reset) by calling `setUpperColumns()` for the specific view criteria.

When you enable case-insensitive search for the view criteria, the framework, when it adjusts the View object a query based on the view criteria, calls the `UPPER()` database function in the `WHERE` clause for those criteria items where case-insensitive search has been enabled. This behavior can be seen when you declaratively define view criteria using the **Create View Criteria** dialog. Notice how the **View Object Where Clause** is altered as you check and uncheck the `Ignore Case` checkbox. This behavior is achieved programmatically as explained in this recipe by calling `setUpperColumns()`.

There's more...

As mentioned earlier, the framework allows you to control case-insensitivity search at various levels. In this recipe, we have seen how to affect case searching for the view criteria as a whole, by utilizing the `setUpperColumns()` method defined for the `ViewCriteria` object. Individual criteria rows and items can be set separately by calling `setUpperColumns()` for specific `VewCriteriaRow` and `ViewCriteriaItem` objects respectively.

See also

Setting up *BC base classes, chapter 1.*

5

Putting Them All Together: Application Modules

In this chapter, we will cover:

- Creating and using generic extension interfaces
- Exposing a custom method as a Web service
- Accessing a service interface method from another Application Module
- A passivation/activation framework for custom session-specific data
- Displaying Application Module pool statistics
- Using a shared Application Module for static lookup data
- Using a custom database transaction

Introduction

An Application Module in the ADF Business components framework represents a basic transactional unit that implements specific business use cases. It encompasses a data model that is comprised of a hierarchy of View object and possibly other Application Module instances and a number of custom methods that together implement the specific business use cases. It allows through the corresponding Application Module **Data Control** and the **ADF Model Layer (ADFm)** for the creation of **bindings** at the ViewController project layer. 3

Moreover, it allows for the creation of custom functionality that can be exposed to its client interface and subsequently bound as method bindings. These bindings declaratively "bind" user interface components to back-end data and services thus making the creation of complex user interfaces easier.

Custom Application Module methods can easily be exposed as Web Services through the Application Module service interface.

Creating and using generic extension interfaces

Back in chapter 1, in the *Setting up BC base classes* recipe, we introduced a number of framework extension classes for various Business Components. We did this so that we can provide common implementation functionality for all derived Business Components throughout the application. In this recipe we will go over how to expose any of that common functionality as an extension interface. By doing so, this generic interface becomes available to all derived Business Components, which in turn they can expose it to their own client interface and make it available to the ViewController layer through the bindings layer.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

This recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

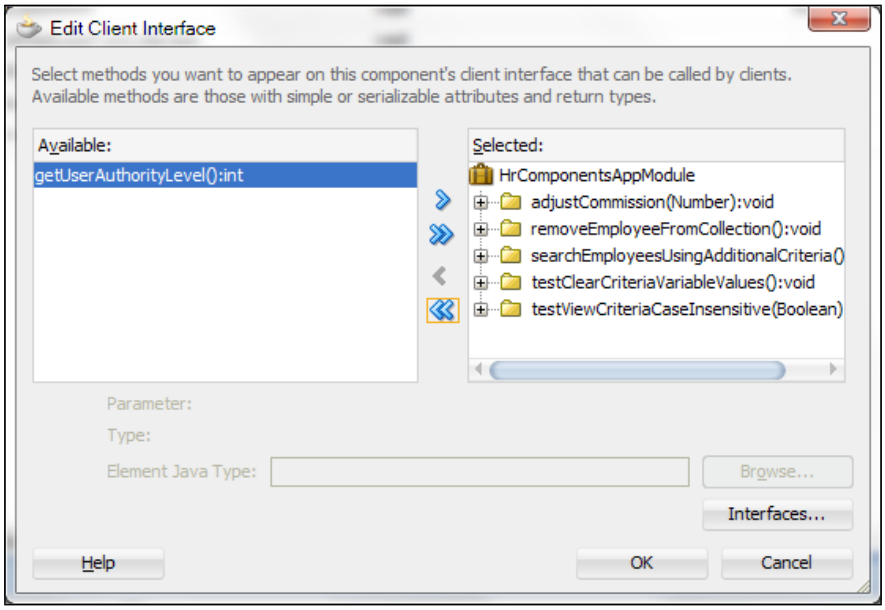
1. Open the shared components workspace in JDeveloper.
2. Create an interface called `ExtApplicationModule` as it shown below:

```
public interface ExtApplicationModule {
  // return some user authority level, based on
  // the user's name
  public int getUserAuthorityLevel();
}
```

3. Locate and open the custom Application Module framework extension class `ExtApplicationModuleImpl`. Modify it so that it implements the `HrComponentsAppModule` interface.
4. Then, add the following method to it:

```
public int getUserAuthorityLevel() {
    // return some user authority level, based on
    // the user's name
    return ("anonymous".equalsIgnoreCase(
        ADFContext.getCurrent().getSecurityContext()
                .getUserPrincipal().getName()))
            ? AUTHORITY_LEVEL_MINIMAL
            : AUTHORITY_LEVEL_NORMAL;
}
```

5. Rebuild the shared components workspace and deploy it as an ADF Library JAR.
6. Now open the `HRComponents` workspace.
7. Locate and open the `HrComponentsAppModule` Application Module definition.
8. Go to the **Java** section and click on the **Edit application module client interface** button (the pen icon in the **Client Interface** section).
9. On the **Edit Client Interface** dialog, shuttle the **getUserAuthorityLevel()** interface from the **Available** to the **Selected** list.



How it works...

In steps 1 and 2 we opened the shared components workspace and we created an interface called `HrComponentsAppModule`. This interface contains a single method called `getUserAuthorityLevel()`.

Then, we updated the Application Module framework extension class `HrComponentsAppModuleImpl` so that it implements the `HrComponentsAppModule` interface (step 3). We also implemented the method `getUserAuthorityLevel()` required by the interface (step 4). For the sake of this recipe, this method returns a user authority level based on the authenticated user's name. We retrieve the authenticated user's name by calling `getUserPrincipal().getName()` on the `SecurityContext`, which we retrieve from the current `ADFContext` (`ADFContext.getCurrent().getSecurityContext()`). If security is not enabled for the ADF application, the user's name defaults to `anonymous`. In this example, for anonymous users we return `AUTHORITY_LEVEL_MINIMAL` and for all others `AUTHORITY_LEVEL_NORMAL`. We rebuilt and redeployed the shared components workspace in step 5.

In steps 6 through 9 we opened the `HRComponents` workspace and we added the `getUserAuthorityLevel()` method to the `HrComponentsAppModuleImpl` client interface. By doing this, we have exposed the `getUserAuthorityLevel ()` generic extension interface to a derived Application Module, while keeping its implementation in the base framework extension class `ExtApplicationModuleImpl`.

There's more...

Note that the steps that were followed in this recipe to expose an Application Module framework extension class method to a derived class' client interface can be followed for other Business Components framework extension classes as well.

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Exposing a custom method as a web service

Service-enabling an Application Module allows you among others to expose custom Application Module methods as web services. This is one way for service consumers to consume the service-enabled Application Module. The other possibilities are accessing the Application Module by another Application Module and accessing it through a Service Component Architecture (SCA) composite. Service-enabling an Application Module allows access to the same Application Module both through web service clients and interactive web user interfaces.

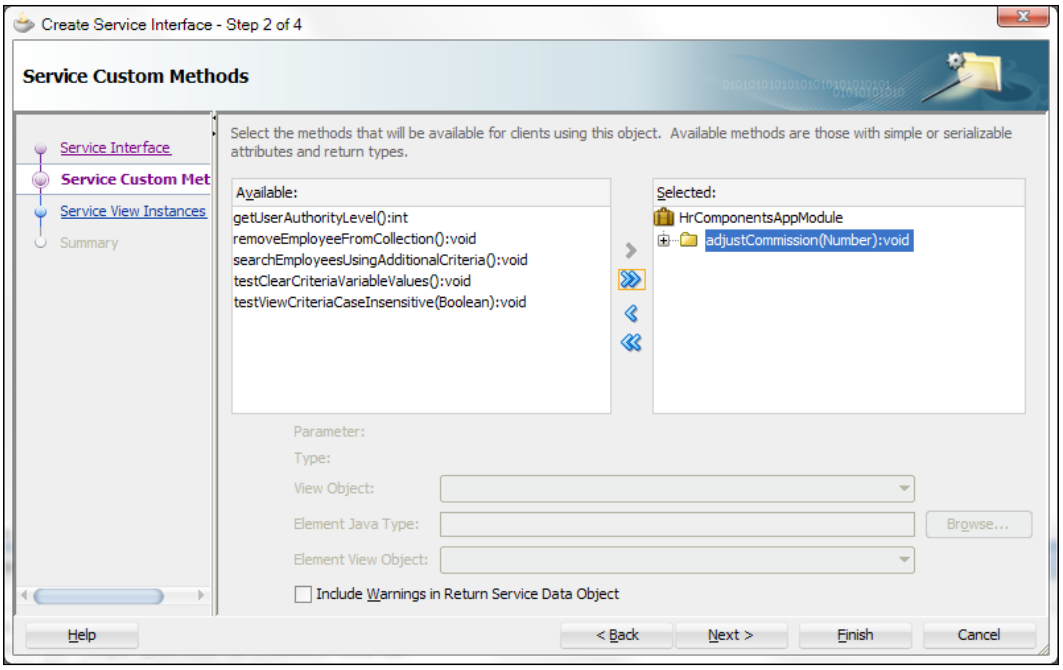
Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

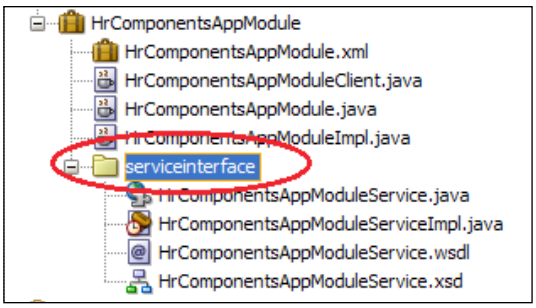
Furthermore, for this recipe we will expose the `adjustCommission()` Application Module method that was developed back in chapter 3 for the *Iterating a View object using a secondary rowset iterator* recipe as a web service.

How to do it...

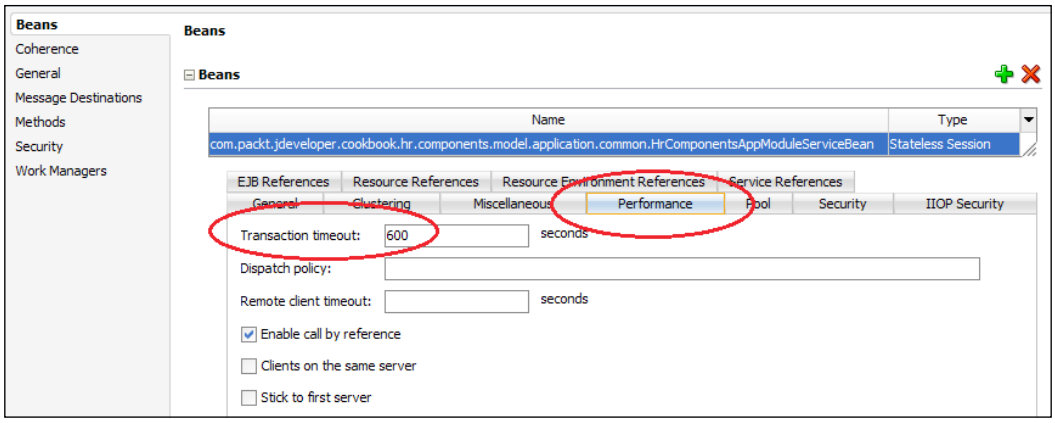
1. Open the `HRComponents` project in JDeveloper.
2. Double-click on the `HRComponentsAppModule` Application Module in the **Application Navigator** to open its definition.
3. Go to the **Service Interface** section and click on the **Enable support for Service Interface** button (the green plus sign icon in the **Service Interface** section). This will start the **Create Service Interface** wizard.
4. In the **Service Interface** page accept the defaults and click **Next**.
5. In the **Service Custom Methods** page locate the **adjustCommission()** method and shuttle it from the **Available** list to the **Selected** list. Click **Finish**.



6. Observe that the **adjustCommission()** method is shown in the **Service Interface Custom Methods** section of the Application Module's **Service Interface** and that the service interface files were generated in the **serviceinterface** package under the Application Module and are shown in the Application Navigator.



7. Double-click on the **weblogic-ejb-jar.xml** file under the **META-INF** package in the **Application Navigator** to open it.
8. In the **Beans** section, select the **com.packt.jdeveloper.cookbook.hr.components.model.application.common.HrComponentsAppModuleServiceBean** bean and click on the **Performance** tab. For the **Transaction timeout** field, enter **600**.



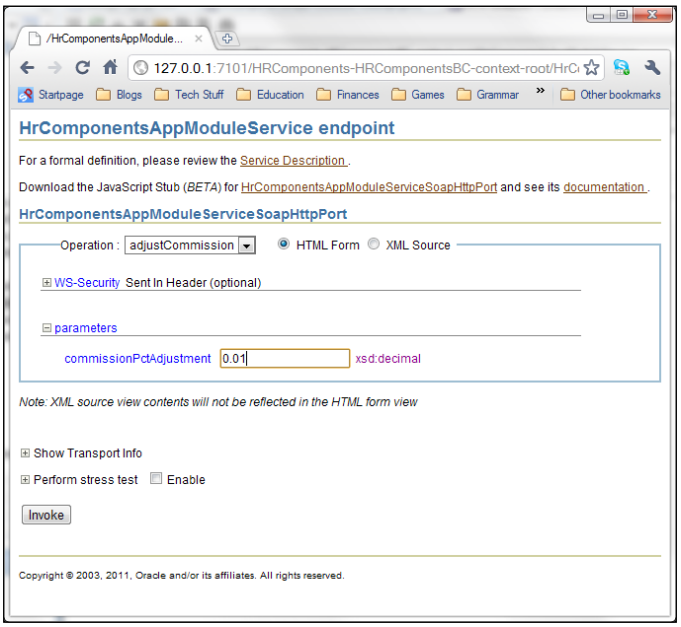
How it works...

In steps 1 through 6, we have exposed the `adjustCommission()` custom Application Module method to the Application Modules service interface. This is a custom method that adjusts all the Sales department employees' commissions by the percentage specified. As a result of exposing the `adjustCommission()` method to the Application Module service interface, JDeveloper generates the following files:

- `HrComponentsAppModuleService.java` – Defines the service interface. Uses metadata annotations to indicate how the interface is exposed as a web service. It is located in the `common/serviceinterface` package.
- `HrComponentsAppModuleServiceImpl.java` – The service implementation class. An EJB 3.0 stateless session bean that implements the `HrComponentsAppModuleService` interface. It extends the `oracle.jbo.server.svc.ServiceImpl` class, the ADF Business Components generic service engine. It is located in the `server/serviceinterface` package.
- `HrComponentsAppModuleService.xsd` – The service schema file describing the input and output parameters of the service. It is located in the `common/serviceinterface` package.
- `HrComponentsAppModuleService.wsdl` – The Web Service Definition Language (WSDL) file, describing the web service. It is located in the `common/serviceinterface` package.
- `ejb-jar.xml` – The EJB deployment descriptor. It is located in the `src/META-INF` directory.
- `weblogic-ejb-jar.xml` – The WebLogic-specific EJB deployment descriptor. It is located in the `src/META-INF` directory.

In steps 7 and 8, we have adjusted the service Java Transaction API (JTA) transaction timeout (defaults to 30 seconds) to 600 seconds (10 minutes). This will avoid any exceptions related to transaction timeouts when invoking the service.

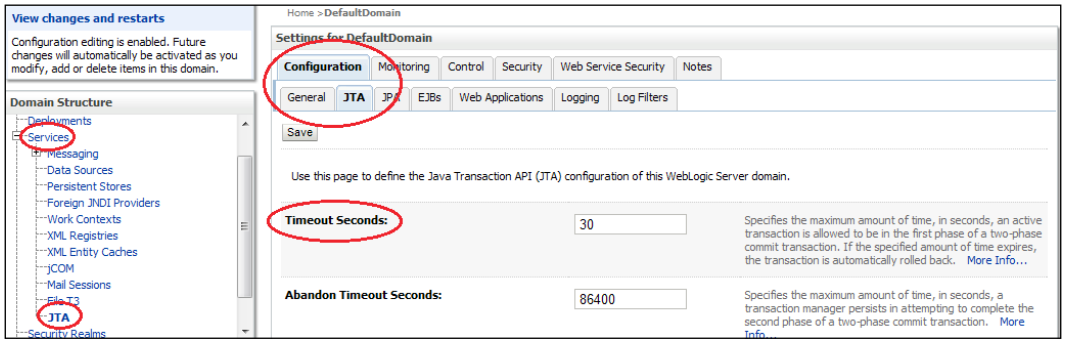
To test the service using the JDeveloper integrated WebLogic application server, right-click on the **HrComponentsAppModuleServiceImpl.java** service implementation file in the **Application Navigator** and select **Run** or **Debug** from the context menu. This will build and deploy the `HrComponentsAppModuleService` web service into the integrated WebLogic server. Once the deployment process is completed successfully, copy the target service URL from the **Log** window and paste it into your browser's address field. This will bring up the service's endpoint page.



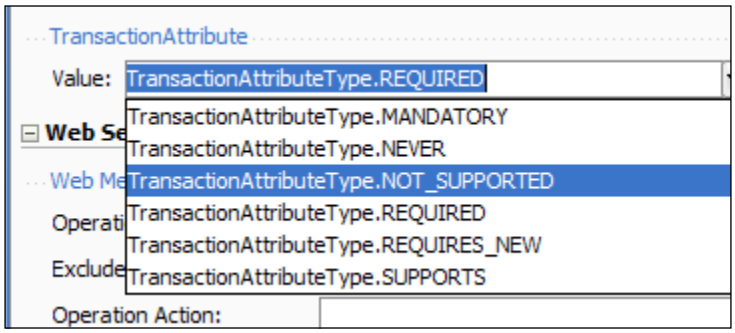
On it, select the **adjustCommission** method from the **Operation** drop down, specify the **commissionPctAdjustment** parameter amount and click on the **Invoke** button to execute the web service. Observe how the employees' commissions are adjusted in the `EMPLOYEES` table in the `HR` schema.

There's more...

In steps 6 and 7 we adjusted the JTA transaction timeout at the `HrComponentsAppModuleService` service level. This affects all methods exposed in the service interface. The JTA transaction timeout can also be adjusted globally for the server using the WebLogic Administration Console by selecting **Services > JTA** from the **Domain Structure** tree and adjusting the **Timeout Seconds** field in the **Configuration > JTA** tab.



The JTA transaction timeout can also be adjusted on a per method basis by specifying **TransactionAttributeType.NOT_SUPPORTED** for the **TransactionAttribute Value** field in the **Property Inspector** for the specific service method.



This will add a `@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)` annotation to the specific method. You will then need to handle the transaction commit and/or rollback in the service method yourself, by manually calling `getTransaction().commit()` or `getTransaction().rollback()` respectively.

See also

Overriding remove() to delete associated children entities, chapter 2.

Iterating a View object using a secondary rowset iterator, chapter 3

Accessing a service interface method from another Application Module

In the recipe Exposing a custom method as a Web service in this chapter, we went through the steps required to service-enable an Application Module and expose a custom Application Module method as a web service. We will continue in this recipe by explaining how to invoke the custom Application Module method, exposed as a web service, from another Application Module.

Getting ready

This recipe will call the `adjustCommission()` custom Application Module method that was exposed as a web service in the Exposing a custom method as a Web service recipe in this chapter . It requires that the web service is deployed in WebLogic and that it is accessible.

The recipe also requires that both the shared components workspace and the `HRComponents` workspace are deployed as ADF Library JARs and that are added to the workspace used by this specific recipe. Moreover, a database connection to the `HR` schema is required.

How to do it...

1. Ensure that you have build and deployed as ADF Library JARs both the `SharedComponents` and `HRComponents` workspaces.
2. Create a File System connection in the Resource Palette to the directory path where the `SharedComponents.jar` and `HRComponents.jar` ADF Library JARs are located. In the book source code, they are located in the `chapter5/recipe3/ReUsableJARs` directory.
3. Create a new **Fusion Web Application (ADF)** called `HRComponentsCaller` using the **Create Fusion Web Application (ADF)** wizard.

4. Create a new Application Module called `HRComponentsCallerAppModule` using the **Create Application Module** wizard. In the **Java** page, check on the **Generate Application Module Class** checkbox to generate a custom Application Module implementation class. JDeveloper will ask you for a database connection during this step, so make sure that a new database connection to the `HR` schema is created.
5. Expand the **File System > ReUsableJARs** connection in the **Resource Palette** and add both the `SharedComponents` and `HRComponents` libraries to the project. You do this by right-click on the jar file and selecting **Add to Project...** from the context menu.
6. Bring up the Business Components **Project Properties** dialog and go to the **Libraries and Classpath** section. Click on the **Add Library...** button and add the **BC4J Service Client** and **JAX-WS Client** extensions.
7. Double-click on the **HRComponentsCallerAppModuleImpl.java** custom Application Module implementation file in the **Application Navigator** to open it in the Java editor.

   Add the following method to it:

```
public void adjustCommission(
          BigDecimal commissionPctAdjustment) {
  // get the service proxy
  HrComponentsAppModuleService service =
    (HrComponentsAppModuleService)ServiceFactory
          .getServiceProxy(
           HrComponentsAppModuleService.NAME);
  // call the adjustCommission() service
  service.adjustCommission(commissionPctAdjustment);
}
```

8. Expose `adjustCommission()` to the `HRComponentsCallerAppModule` client interface.
9. Finally, in order to be able to test the `HRComponentsCallerAppModule` Application Module with the ADF Model Tester, locate the **connections.xml** file in the **Application Resources** section of the **Application Navigator** under the **Descriptors > ADF META-INF** node, and add the following configuration to it:

```
<Reference name="{/com/packt/jdeveloper/cookbook/hr/components/model/
application/common/}HrComponentsAppModuleService"
        className="oracle.jbo.client.svc.Service" xmlns="">
  <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
  <RefAddresses>
    <StringRefAddr addrType="serviceInterfaceName">
      <Contents>com.packt.jdeveloper.cookbook.hr.components.model.
application.common.serviceinterface.HrComponentsAppModuleService</Contents>
```

```
            </StringRefAddr>
            <StringRefAddr addrType="serviceEndpointProvider">
              <Contents>ADFBC</Contents>
            </StringRefAddr>
            <StringRefAddr addrType="jndiName">
              <Contents>HrComponentsAppModuleServiceBean#com.packt.
jdeveloper.cookbook.hr.components.model.application.common.serviceinterface.
HrComponentsAppModuleService</Contents>
            </StringRefAddr>
            <StringRefAddr addrType="serviceSchemaName">
              <Contents>HrComponentsAppModuleService.xsd</Contents>
            </StringRefAddr>
            <StringRefAddr addrType="serviceSchemaLocation">
              <Contents>com/packt/jdeveloper/cookbook/hr/components/model/
application/common/serviceinterface/</Contents>
            </StringRefAddr>
            <StringRefAddr addrType="jndiFactoryInitial">
              <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
            </StringRefAddr>
            <StringRefAddr addrType="jndiProviderURL">
              <Contents>t3://localhost:7101</Contents>
            </StringRefAddr>
          </RefAddresses>
        </Reference>
```

How it works...

In steps 1 and 2 we made sure that both the shared components and HR components ADF Library JARs were deployed and that a file system connection was created in order that both of these libraries are added to a newly created project (in step 5). Then, in steps 3 and 4, we created a new Fusion Web Application based on ADF, and an Application Module called `HRComponentsCallerAppModule`. It is from this Application Module that we intend to call the `adjustCommission()` custom Application Module method, exposed as a web service by the `HrComponentsAppModule` service-enabled Application Module in the `HRComponents` library JAR. For this reason in step 4 we have generated a custom Application Module implementation class. We proceeded by adding the necessary libraries to the new project in steps 5-6. Specifically, the following libraries were added:

- `SharedComponents.jar` – The shared components library. Provides among others a number of framework extension classes. It is needed in this case because the Application Module should have been derived from the generic framework extension class `ExtApplicationModuleImpl`.

- `HRComponents.jar` – The HR components library. It is in this library that we have both implemented the service-enabled `HrComponentsAppModule` Application Module, the custom Application Module method `adjustCommission()` exposed as a web service and the actual service implementation as an EJB 3.0 stateless session bean (`HrComponentsAppModuleServiceImpl.java`). It is required at compile-time, since we will be using the `HrComponentsAppModuleService` service interface (in step 8).
- `BC4J Service Client` – Provides access to both `adfbcsvc-client.jar` and `adfbcsvc-share.jar` libraries that provide access the `ServiceFactory` and `ServiceException` classes respectively. `ServiceFactory` is used explicitly in step 8, to get access to the web service proxy via the `getServiceProxy()` method. `ServiceException` is required at runtime.
- `JAX-WS Client` – The Java API for XML-Based Web Services (JAX-WS) web service client programming model to support the Dynamic Proxy client API. It is required in order to invoke the web service based on the service endpoint interface.

In steps 7 through 9, we created a custom Application Module method called `adjustCommission()`, in which we wrote the necessary glue code to call our web service. In it we first retrieved the web service proxy, as a `HrComponentsAppModuleService` interface, by calling `ServiceFactory.getServiceProxy()` and specifying the name of the web service, which is indicated by the constant `HrComponentsAppModuleService.NAME` in the service interface. Then we called the web service through the retrieved interface.

In the last step, we had to provide the necessary configuration in the `connections.xml` so that we will be able to call the web service from an RMI client (the ADF Model Tester). This file is used by the web service client to locate the web service. For the most part the `<Reference>` information that was added to it was generated automatically by JDeveloper in the Exposing a custom method as a Web service recipe, so it was copied from there. The extra configuration information that had to be added is the necessary JNDI context properties `jndiFactoryInitial` and `jndiProviderURL` necessary to resolve the web service on the deployed server. You should change these appropriately for your deployment.

To test calling the web service, ensure that you first have deployed it and that it is running. You can then use the ADF Model Tester, select the **adjustCommission** method and to execute it.

There's more...

For additional information related to such topics as securing the ADF web service, enabling support for binary attachments, deploying to WebLogic and more, refer to the Integrating Service-Enabled Application Modules section in the Fusion Developer's Guide for Oracle Application Development Framework.

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Exposing a custom method as a Web service, chapter 5.

A passivation/activation framework for custom session-specific data

In order to preserve a stateful notion while utilizing a stateless protocol (i.e., HTTP) the ADF Business Components framework implements the concept of Application Module pooling. This is the process of maintaining a limited number of Application Modules, the exact number specified by configuration, in a pool, which are preserved across multiple user requests for the same HTTP session. When all available Application Modules have been associated with specific sessions, an Application Module already linked with a particular session must be re-used. This requires that the data associated with the Application Module is saved. The process of saving the information associated with the specific Application Module is called **passivation** in Business Components terminology. The information is stored in a **passivation store**, usually a database, in XML format. The opposite process of restoring the state of the Application Module from the passivation store is called **activation**.

Custom data is associated with specific Application Modules, and therefore with specific user sessions, by using a user data `Hashtable` obtained from an `oracle.jbo.Session` object. This `Session` object is obtained by calling `getSession().getUserData()` from the Application Module implementation class. If you are using such custom data as part of some algorithm in your application and you expect the custom data to persist from one user request to another, passivation (and subsequent activation) support for these custom data must be implemented programmatically. You can add custom passivation and activation logic to your Application Module implementation class by overriding the `ApplicationModuleImpl` methods `passivateState()` and `activateState()` respectively. The `passivateState()` method creates the necessary XML elements for the Application Module custom data that must be passivated. On the other hand, the `activateState()` method detects the specific XML elements that identify the custom data in the passivated XML document and restores them back into the session custom data.

This recipe will show you how to do this and at the same time build a mini framework to avoid duplication of the basic passivation/activation code that you must write for all your Application Modules in your project.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

This recipe is also using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the shared components workspace in JDeveloper and load the `ExtApplicationModuleImpl` Application Module framework extension class in the Java editor.
2. Add the following methods to the `ExtApplicationModuleImpl` Application Module framework extension class:

```
protected String[] onStartPassivation() {
  // default implementation: no passivation ids
  // are defined
  return new String[] { };
}

protected String onPassivate(String passivationId) {
  // default implementation: passivates nothing
  return null;
}

protected void onEndPassivation() {
  // default implementation: does nothing
}

protected String[] onStartActivation() {
  // default implementation: no activation ids
  // are defined
  return new String[] { };
}
```

```
    protected void onActivate(String activationId,
                              String activationData) {
      // default implementation: activates nothing
    }

    protected void onEndActivation() {
      // default implementation: does nothing
    }
```

3. Override the void `passivateState(Document, Element)` method. Add the following code after the call to `super.passivateState()`:

```
    // begin custom data passivation: returns a
    // list of the custom data passivation identifiers
    String[] passivationIds = onStartPassivation();
    // process all passivation identifiers
    for (String passivationId : passivationIds) {
      // check for valid identifier
      if (passivationId != null &&
          passivationId.trim().length() > 0) {
        // passivate custom data: returns
        // the passivation data
        String passivationValue =
                onPassivate(passivationId);
        // check for valid passivation data
        if (passivationValue != null &&
            passivationValue.length() > 0) {
          // create a new text node in the
          // passivation XML
          Node node =
           document.createElement(passivationId);
          Node cNode =
          document.createTextNode(passivationValue);
          node.appendChild(cNode);
          // add the passivation node to the
          // parent element
          element.appendChild(node);
        }
      }
    }
```

```
// inform end of custom data passivation
onEndPassivation();
```

4. Override the `activateState(Element element)` method. Add the following code after the call to `super.activateState()`:

```
// check for element to activate
if (element != null) {
   // begin custom data activation: returns a
   // list of the custom data activation identifiers
   String[] activationIds = onStartActivation();
   // process all activation identifiers
   for (String activationId : activationIds) {
      // check for valid identifier
      if (activationId != null &&
            activationId.trim().length() > 0) {
         // get nodes from XML for the specific
         // activation identifier
         NodeList nl =
         element.getElementsByTagName(activationId);
         // if it was found in the activation data
         if (nl != null) {
            // activate each node
            for (int n = 0, length =
               nl.getLength(); n < length; n++) {
               Node child =
                  nl.item(n).getFirstChild();
               if (child != null) {
                  // do the actual custom data
                  // activation
                  onActivate(activationId,
                   child.getNodeValue().toString());
                  break;
               }
            }
         }
      }
   }
```

```
        // inform end of custom data activation
        onEndActivation();
      }
```

5. Rebuild and redeploy the shared components ADF Library JAR.
6. Open the `HRComponents` workspace and load the `HrComponentsAppModuleImpl` `HrComponentsAppModule` Application Module custom implementation class into the Java editor.
7. Add the following `getActivationPassivationIds()` helper method. Also ensure that you define a constant called `CUSTOM_DATA_PASSIVATION_ID` indicating the custom data passivation identifier.

```
private static final String CUSTOM_DATA_PASSIVATION_ID =
                                "customDataPassivationId";

private String[] getActivationPassivationIds() {
    // return the passivation/activation identifiers
    return new String[] { CUSTOM_DATA_PASSIVATION_ID };
}
```

8. Override the `onStartPassivation()`,`onPassivate()`,`onStartActivation()` and `onActivate()` methods. Provide the following implementation for them:

```
protected String[] onStartPassivation() {
  // return the passivation identifiers
  return getActivationPassivationIds();
}

protected String onPassivate(String passivationId) {
  String passivationData = null;
  // passivate this Application Module's
  // custom data only
  if (CUSTOM_DATA_PASSIVATION_ID.equals(
                  passivationId)) {
    // return the custom data from the Application
    // Module session user data
    passivationData = (String)getSession()
      .getUserData().get(CUSTOM_DATA_PASSIVATION_ID);
  }
  return passivationData;
}

protected String[] onStartActivation() {
```

```
    // return the activation identifiers
    return getActivationPassivationIds();
  }

  protected void onActivate(String activationId,
                            String activationData) {
    // activate this Application Module's custom data only
    if (CUSTOM_DATA_PASSIVATION_ID.equals(activationId)) {
      // add custom data to the Application
      // Module's session
      getSession().getUserData().put(
        CUSTOM_DATA_PASSIVATION_ID, activationData);
    }
  }
```

9. Finally, for testing purposes, override the `prepareSession()` method and add the following code after the call to `super.prepareSession()`:

```
// add some custom data to the Application
// Module session
getSession().getUserData()
    .put(CUSTOM_DATA_PASSIVATION_ID,
                  "Some custom data");
```

How it works...

In the first two steps, we have laid out a basic passivation/activation framework by adding a number of methods to the `ExtApplicationModuleImpl` Application Module framework extension class dealing specifically with this process. Specifically, these methods are:

- `onStartPassivation()` – The framework calls this method to indicate that a passivation process is about to start. Derived Application Modules that need to passivate custom data will override this method and return a `java.lang.String` array of passivation identifiers, indicating custom data that needs to be passivated.
- `onPassivate()` – The framework calls this method to indicate that some specific custom data, identified by the `passivationId` parameter, needs to be passivated. Derived Application Modules will override this method to passivate the specific custom data. It returns the passivated data as a `java.lang.String`.
- `onEndPassivation()` – This method is called by the framework to indicate that the passivation process is complete. Derived Application Modules could override this method to perform post-passivation actions.
- `onStartActivation()` – Called by the framework to indicate that an activation process is about to begin. Derived Application Modules in need of activating custom data, should override this method and return a list of activation identifiers.

- `onActivate()` – It is called by the framework when some custom data, the parameter `activationData`, needs to be activated. The custom data is identified by a unique identifier indicated by the parameter `activationId`. Derived Application Modules should override this method and restore the custom data being activated into the Application Module's user data `Hashtable`.
- `onEndActivation()` – Indicates the end of the activation process. It can be overrided by derived Application Modules to do some post-activation actions.

Note that these methods do nothing at the base class level. It is when they are overridden by derived Application Modules (see step 8) that come to life.

In step 3 we overrode the ADF Business Components framework method `passivateState()` and hooked up our own passivation/activation framework to it. ADF calls this method to indicate that a passivation is taking place. In it, after calling `super.passivateState()` to allow for the ADF processing, we first call `onStartPassivation()`. If a derived Application Module has overridden this method, it should have returned a list of passivation identifiers. These identifiers should uniquely identify at the Application Module level the Application Module custom data to be passivated. We then iterate over the passivation identifiers calling `onPassivate()` for each one to retrieve the passivation data. We create a new XML node for the passivation identifier, we add the passivation data to it and we append it to the parent XML node that is passed as a parameter by the ADF framework (the `element` parameter) to `passivateState()`. When all passivation identifiers have been processed, `onEndPassivation()` is called.

Step 4 is somewhat similar and does the activation. In this case we overrode the ADF `activateState()` method which is called by the framework to indicate that the activation process is taking place. In it, we first call `super.activateState()` to allow for the framework processing and then `onStartActivation()` to get a list of the activation identifiers. We iterate over the activation identifiers, looking for each identifier in the activated XML data for the Application Module element. This is done by calling `element.getElementsByTagName()`. `getElementsByTagName()` could possibly return multiple nodes, so for each we call `onActivate()` to activate the specific custom data. When we call `onActivate()` we pass the activation identifier and the activation data to it as arguments. It is then the responsibility of the derived Application Module to handle the specifics of the activation. Finally, when all activation identifiers have been processed, we call `onEndActivation()` to indicate that the activation process has ended.

After we added these changes to the `ExtApplicationModuleImpl` Application Module framework extension class, we made sure that the shared components ADF Library JAR was redeployed (in step 5).

In steps 6 through 8 we added passivation/activation support for custom data to the `HrComponentsAppModule` Application Module in the `HRComponents` workspace. This was done by overriding the `onStartPassivation()`, `onPassivate()`, `onStartActivation()` and `onActivate()` methods (in step 8). The list of passivation and activation identifiers was provided by the `getActivationPassivationIds()` method that we added in step 7. For this recipe, only a single custom data, identified by the constant `CUSTOM_DATA_PASSIVATION_ID`, is passivated. Custom data is saved at the user data `Hashtable` in the `oracle.jbo.Session` associated with the specific Application Module. It is retrieved by calling `getSession().getUserData().get(CUSTOM_DATA_PASSIVATION_ID)` in the `onPassivate()`. Similarly it is set in `onActivate()` by calling `getSession().getUserData().put(CUSTOM_DATA_PASSIVATION_ID, activationData)`. Notice in this case that the activation data is passed as an argument (the `activationData` parameter) to the `onActivate()` by the `activateState()` implemented in Application Module framework extension class in step 4.

Finally, note the code in step 9. In the overridden prepareSession() we have initialized the custom data by calling `getSession().getUserData().put(CUSTOM_DATA_PASSIVATION_ID, "Some custom data")`.

To test the custom data passivation/activation framework, run the Application Module with the ADF Model Tester. The ADF Model Tester provides support for passivation and activation via the **Save Transaction State** and **Restore Transaction State** menu items under the **File** menu. Observe the generated passivation XML data in the JDeveloper Log window when **File > Save Transaction State** is chosen. In particular, observe that the `<customDataPassivationId>Some custom data</customDataPassivationId>` node is added to the `<AM>` node of the passivated XML document.

```
[81] **syncSequenceIncrementSize** altered sequence 'increment by' value to 50
[82] <AM MomVer="0">
   <cd></cd>
   <TXN Def="0" New="0" Lok="2" tsi="0" pcid="1"/>
   <CONN/>
   <VO>
      <VO sig="1310500543716" qf="0" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees" Name="Employees"/>
      <VO sig="1310500543836" qf="0" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Departments"
Name="Departments"/>
      <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Departments"
Name="DepartmentsManaged"/>
      <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees"
Name="EmployeesManaged"/>
      <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees"
Name="DepartmentEmployees"/>
   </VO>
   <customDataPassivationId>Some custom data</customDataPassivationId>
</AM>
```

There's more...

Note that `activateState()` method is called by the ADF Business Components framework after the view objects instances associated with the Application Module have been activated by the framework. If you need to activate custom data that would be subsequently accessed by your View objects, then you will need to enhance the custom data passivation/activation framework by overriding `prepareForActivation()` and provide the activation logic there instead.

Also, note that the ADF Business Components framework provides similar `passivateState()` and `activateState()` methods at the View object level for passivating and activating View object custom data. Custom data in this case is stored in the user data `Hashtable` of the `oracle.jbo.Session` associated with the specific Application Module that contains the particular View object in its data model.

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Displaying Application Module pool statistics

In the *A passivation/activation framework for custom session-specific data* recipe in this chapter, we touched upon how Application Module pools are used by the ADF Business Components framework. In this recipe we will introduce the `oracle.jbo.common.ampool.PoolMgr` Application Module pool manager and `oracle.jbo.common.ampool.ApplicationPool` Application Module pool classes and how they can be utilized to collect statistical pool information. This may come handy when debugging.

The use case that will be implemented by the recipe would be to collect Application Module statistics and make them available in a generic View Object that can be used by all Application Modules to gather and present statistical information to the front end user interface.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in chapter 1.

This recipe is also using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the shared components workspace in JDeveloper.
2. Create a new View object called `ApplicationModulePoolStatistics` using the following SQL query as its data source:

   SELECT NULL AS POOL_NAME, NULL AS APPLICATION_MODULE_CLASS, NULL AS AVAILABLE_INSTANCE_COUNT, NULL AS INIT_POOL_SIZE, NULL AS INSTANCE_COUNT, NULL AS MAX_POOL_SIZE, NULL AS NUM_OF_STATE_ACTIVATIONS, NULL AS NUM_OF_STATE_PASSIVATIONS, NULL AS NUM_OF_INSTANCES_REUSED, NULL AS REF_INSTANCES_RECYCLED, NULL AS UNREF_INSTANCES_RECYCLED, NULL AS REFERENCED_APPLICATION_MODULES, NULL AS NUM_OF_SESSIONS, NULL AS AVG_NUM_OF_SESSIONS_REF_STATE FROM DUAL
3. With the exception of the `PoolName` and `ApplicationModuleClass` attributes which should be `String` data types, all other attributes should be `Number` types.
4. Designate the `PoolName` and `ApplicationModuleClass` attributes as key attributes.
5. In the **Java** section, create a custom View Row Class and ensure that the **Include accessors** checkbox is also checked.
6. Open the `ExtApplicationModuleImpl` Application Module custom framework class in the Java editor and add the following two methods to it:

```
public ExtViewObjectImpl
        getApplicationModulePoolStatistics() {
   return (ExtViewObjectImpl)findViewObject(
           "ApplicationModulePoolStatistics");
}

public void getAMPoolStatistics() {
   // get the pool manager
   PoolMgr poolMgr = PoolMgr.getInstance();
   // get the pools managed
   Enumeration keys = poolMgr.getResourcePoolKeys();
   // iterate over pools
   while (keys != null && keys.hasMoreElements()) {
      // get pool name
      String poolname = (String)keys.nextElement();
```

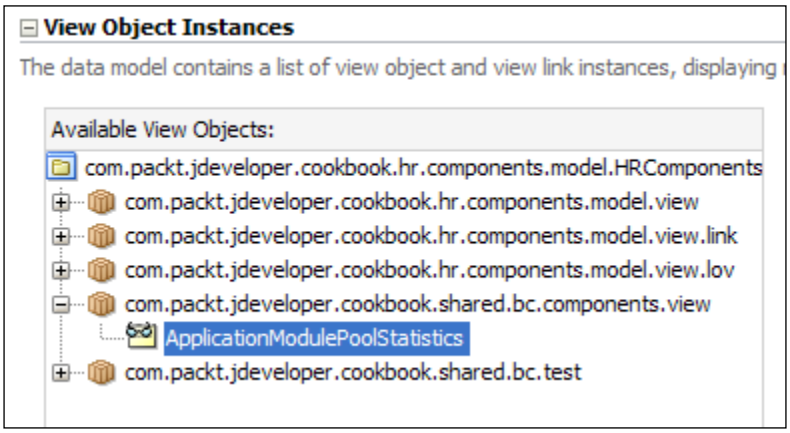
```
// get the pool
ApplicationPool pool =
(ApplicationPool)poolMgr.getResourcePool(poolname);
// get the pool statistics
Statistics statistics = pool.getStatistics();
// get and populate pool statistics view object
ExtViewObjectImpl amPoolStatistics =
      getApplicationModulePoolStatistics();
if (amPoolStatistics != null) {
   // empty the statistics
   amPoolStatistics.executeEmptyRowSet();
   // create and fill a new statistics row
   ApplicationModulePoolStatisticsRowImpl
     poolInfo =
      (ApplicationModulePoolStatisticsRowImpl)
        amPoolStatistics.createRow();
   poolInfo.setPoolName(pool.getName());
   poolInfo.setApplicationModuleClass(
     pool.getApplicationModuleClass());
   poolInfo.setAvailableInstanceCount(new
     Number(pool.getAvailableInstanceCount()));
   poolInfo.setInitPoolSize(new
     Number(pool.getInitPoolSize()));
   poolInfo.setInstanceCount(new
     Number(pool.getInstanceCount()));
   poolInfo.setMaxPoolSize(new
     Number(pool.getMaxPoolSize()));
   poolInfo.setNumOfStateActivations(new
     Number(statistics.mNumOfStateActivations));
   poolInfo.setNumOfStatePassivations(new
     Number(statistics.mNumOfStatePassivations));
   poolInfo.setNumOfInstancesReused(new
     Number(statistics.mNumOfInstancesReused));
   poolInfo.setRefInstancesRecycled(new
Number(statistics.mNumOfReferencedInstancesRecycled));
   poolInfo.setUnrefInstancesRecycled(new
Number(statistics.mNumOfUnreferencedInstancesRecycled));
   poolInfo.setReferencedApplicationModules(new
Number(statistics.mReferencedApplicationModules));
```

```
            poolInfo.setNumOfSessions(new
              Number(statistics.mNumOfSessions));
            poolInfo.setAvgNumOfSessionsRefState(new
        Number(statistics.mAvgNumOfSessionsReferencingState));
            // add the statistics
            amPoolStatistics.insertRow(poolInfo);
          }
        }
      }
```

7. Open the `ExtApplicationModule` Application Module custom framework interface and add the following to it:

```
public void getAMPoolStatistics();
```

8. Redeploy the `SharedComponents.jar` shared components ADF Library JAR.
9. Now, open the `HRComponents` workspace and in the **Resource Palette** create a **File System** connection for the `ReUsableJARs` directory where the `SharedComponents.jar` is deployed. Add the `SharedComponents.jar` to the `HRComponentsBC` Business Components project.
10. Double-click on the **HrComponentsAppModule** Application Module in the **Application Navigator** to open its definition.
11. Go to the **Data Model** section and locate the **ApplicationModulePoolStatistics** View object in the **Available View Objects** list. Shuttle it to the **Data Model** list.



12. Finally, go to the **Java** section, locate and add the **getAMPoolStatistics()** method to the `HRComponents` Application Module client interface.

How it works...

In steps 1 through 5 we created a read-only View object called `ApplicationModulePoolStatistics`, which we will use to collect the Application Module pool statistics. By adding this View object to the shared components workspace, it becomes available to all other projects in all workspaces throughout the ADF application that import the shared components ADF Library JAR. In step 6, we added the necessary functionality to collect the Application Module statistics and populate the `ApplicationModulePoolStatistics` View object. This was done in the `getAMPoolStatistics()` method. In this method we get an instance of the `oracle.jbo.common.ampool.PoolMgr` Application Module pool manager, via the call to the static `getInstance()`, and an `Enumeration` of the Application Module pools managed by the pool manager by calling `getResourcePoolKeys()` on the pool manager. We iterate over all the pools managed by the manager and we retrieve each pool by calling `getResourcePool()` on the pool manager. Then for each pool we call `getStatistics()` to get the pool statistics. We create a new `ApplicationModulePoolStatistics` View object row and we populate it with the statistics information.

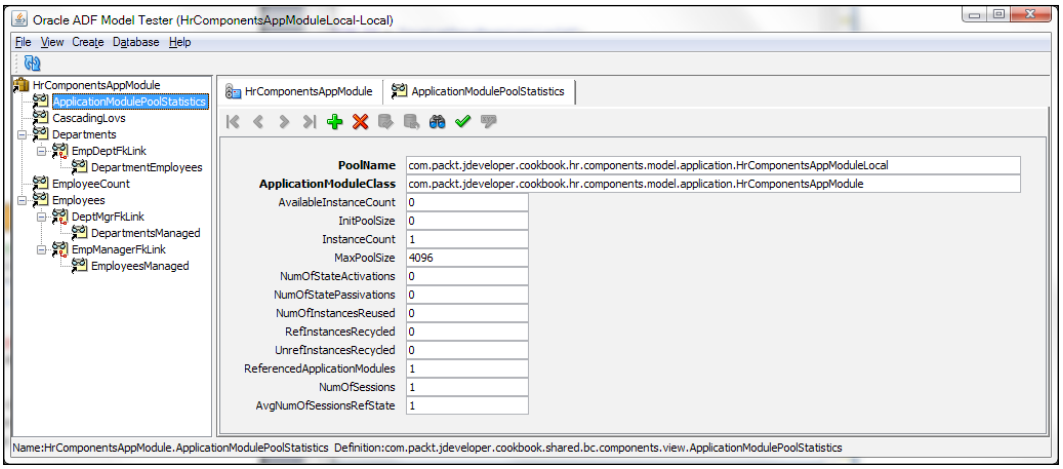
In step 7 we added the `getAMPoolStatistics()` to the `ExtApplicationModule` Application Module framework extension interface so that it becomes available to all Application Modules throughout the application.

In steps 8 and 9 we redeployed the shared components library and we created a file system connection in the Resource Palette. We used this file system connection to add the shared components `SharedComponents.jar` ADF Library JAR to the HRComponents Business Components project.

In steps 10 through 11, we added the `ApplicationModulePoolStatistics` View object to the `HrComponentsAppModule` Application Module data model. Notice how the `ApplicationModulePoolStatistics` View object is listed in the available View objects list although it is implemented in the shared components workspace.

Finally, in step 12, we added the `getAMPoolStatistics()` to the `HrComponentsAppModule` Application Module client interface. By doing so, we will be able to call it using the ADF Model Tester.

To test the recipe, run the `HrComponentsAppModule` Application Module with the ADF Model Tester. In the ADF Model Tester double click on the `HrComponentsAppModule` Application Module to open it, select the **getAMPoolStatistics** method from the **Method** combo, and click on the **Execute** button. Then open the **ApplicationModulePoolStatistics** View object to see the results.



Now you can bind both the `getAMPoolStatistics` method and the `ApplicationModulePoolStatistics` View object to any of your ViewController projects in your ADF application and present a visual of this statistical information for debugging purposes.

There's more...

Note that the `oracle.jbo.common.ampool.ApplicationPool` interface provides a method called `dumpPoolStatistics()` to dump all pool statistics to a `PrintWriter` object. You can use this method to quickly print the Application Module pool statistics to the JDeveloper Log window as shown in following code:

```
PrintWriter out = new PrintWriter(System.out, true);
pool.dumpPoolStatistics(new PrintWriter(out));
out.flush();
```

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Creating and using generic extension interfaces, chapter 5.

Using a shared Application Module for static lookup data

Shared Application Modules allow you to share static read-only data models across multiple user sessions. They are the ideal place to collect all of your static read-only view accessors used throughout your ADF application for validation purposes or as data sources for your List-of-Values (LOVs). This is because a single Application Module is constructed and used throughout the ADF application for all user sessions, thus minimizing the system resources used by it. In this case, a single database connection is used. In addition, by collecting all of your static read-only View objects in a shared Application Module, you avoid possible duplication and redefinition of read-only View objects throughout your ADF application.

Internally, the ADF Business Components framework manages a pool of query collections for each View object since it may accessed by multiple sessions by utilizing a query collection pool, something comparable to Application Module pools used for session-specific Application Modules. The framework offers a number of configuration options to allow for better management of this query collection pool. Moreover, since multiple threads will be accessing the data, the framework partitions the iterator space by supporting multiple iterators for the same rowset to prevent race conditions among the iterators used by the different sessions.

In this recipe we will define a shared Application Module called `HrSharedAppModule`, and we will migrate all of the static read-only View objects defined so far for the `HrComponents` Business Components project to it. Furthermore, we will update all the View objects that currently reference these static read-only View objects, so that they are now referencing the View objects in the shared Application Module.

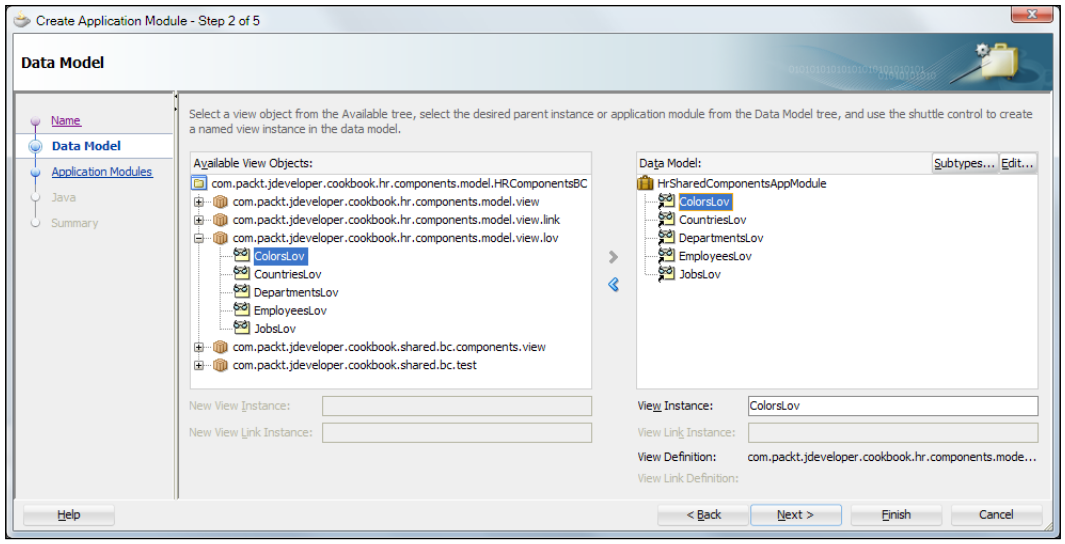
Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

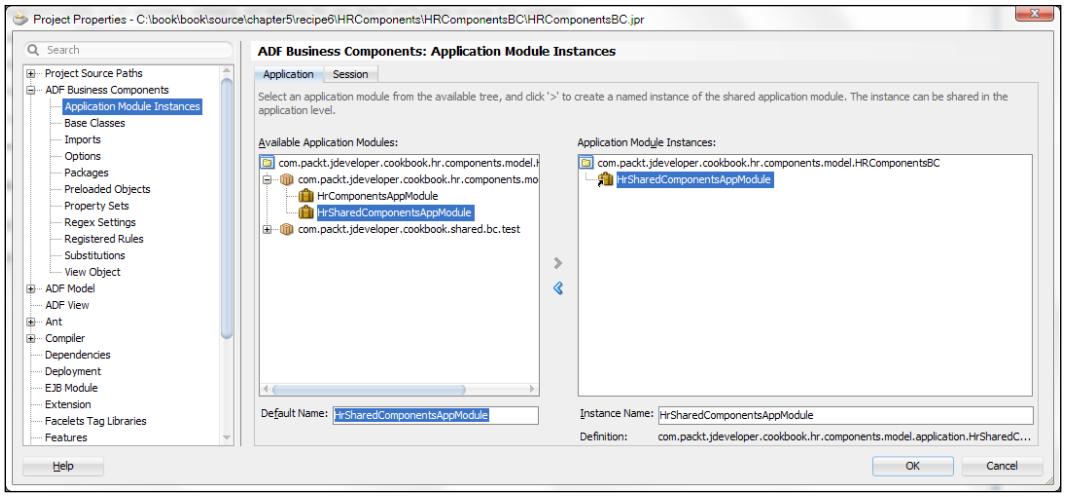
How to do it...

1. Right-click on the `com.packt.jdeveloper.cookbook.hr.components.model.application` package on the **Application Navigator** and select **New Application Module...**.
2. Follow the steps in the **Create Application Module** wizard to create an Application Module called `HrSharedComponentsAppModule`.

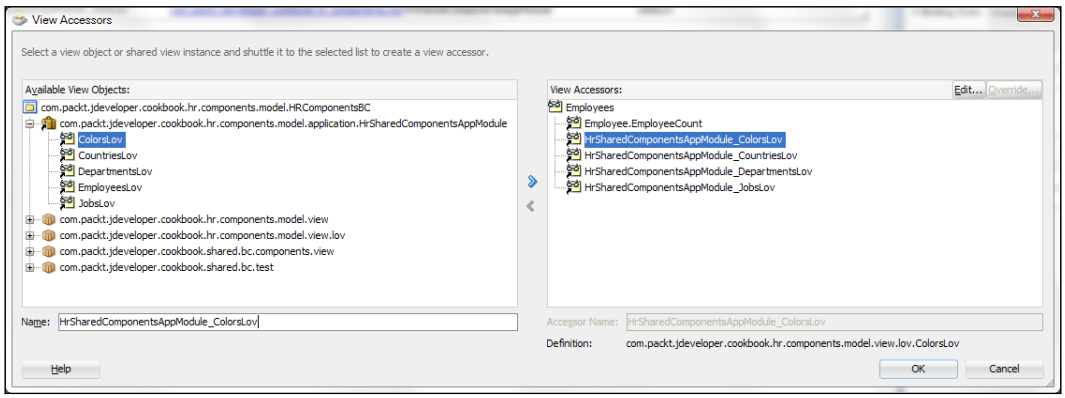
3. In the **Data Model** page, expand the `com.packt.jdeveloper.cookbook.hr.components.model.view.lov` package and shuttle all of the View objects currently under this package from the **Available View Objects** list to the **Data Model** list. Click **Finish** when done.



4. Now double-click on the **HRComponentsBC** in the **Application Navigator** to bring up the **Project Properties** dialog.
5. Locate the **Application Module Instances** page by selecting **ADF Business Components > Application Module Instances** in the selection tree.
6. Click on the **Application** tab and shuttle the **HrSharedComponentsAppModule** Application Module from the **Available Application Modules** list to the **Application Module Instances** list. Click **OK** to dismiss the **Project Properties** dialog.



7. For each View object that was added to the **HrSharedComponentsAppModule** shared Application Module, locate using **Find Usages** where it is used as a view accessor and change its usage so that it is referenced from inside the **HrSharedComponentsAppModule** shared Application Module. The picture below shows the view accessors that were added to the `Employees` View object.



8. Also for each view accessor used as a data source for an LOV, ensure that you are now using the view accessor that is included in the shared Application Module `HrSharedComponentsAppModule`.

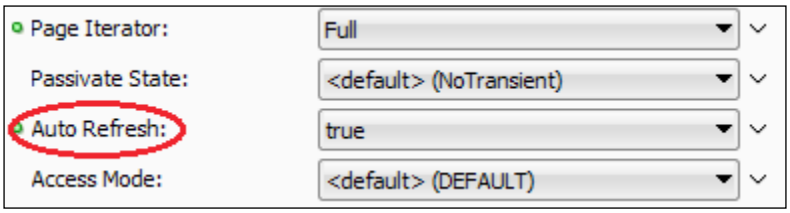
How it works...

In steps 1 through 6 we defined a new Application Module called `HrSharedComponentsAppModule` and we added in its data model all static read-only View objects developed so far throughout the `HRComponents` Business Components project. We have indicated that `HrSharedComponentsAppModule` will be a shared Application Module through the **Application Module Instances** page in the **Project Properties** when we indicated that `HrSharedComponentsAppModule` will be an instance at application-level rather than an instance at session-level (in steps 4 through 6). By defining an Application Module at application-level, we allow all user sessions to access the same view instances contained in the Application Module data model.

In steps 7 and 8 we have identified all read-only View objects used as view accessors throughout the `HRComponents` Business Components project and we updated each one at a time so that the View object instance residing within the shared `HrSharedComponentsAppModule` Application Module is used. We have also ensured that for each LOV we redefined its data source by using the updated view accessor.

There's more...

Ideally the shared Application Module should contain a static read-only data model. If you expect that the data returned by any of the View objects might be updated, ensure that it will always return the latest data from the database by setting the View object **Auto Refresh** property to **true** in the **Tuning** section of the **Property Inspector**. This property is accessible while at the **General** section of the View object definition.



The auto-refresh feature relies on the database change notification feature, so ensure that the data source database user has database notification privileges. This can be achieved by issuing the following `grant` command for the database connection user:

```
grant change notification to <ds_user_name>
```

See also

Overriding remove() to delete associated children entities, chapter 2.

Using a custom database transaction

In the *Setting up BC base classes* recipe in chapter 1, we introduced a number of custom framework extension classes for most of the ADF Business Components. Among these custom framework extension classes there are classes that can be used to extend the global ADF framework transaction implementation, in particular the `ExtDatabaseTransactionFactory` and `ExtDBTransactionImpl2` classes. In this recipe we will go over the steps of how to use these classes so that we can implement our own custom transaction implementation. The use case for this recipe will be to provide logging support for all transaction commit and rollback operations.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. Additional functionality will be added to the `ExtDatabaseTransactionFactory` and `ExtDBTransactionImpl2` custom framework classes that were developed in the *Setting up BC base classes* recipe in chapter 1.

This recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

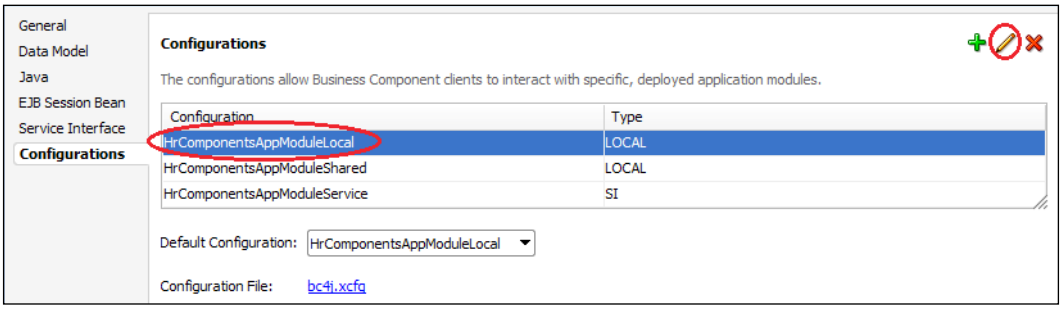
1. Open the shared components workspace and open the `ExtDatabaseTransactionFactory.java` file in the Java editor.
2. Override the `DatabaseTransactionFactory create()` method and replace the `return super.create()` with the following code:

```
// return custom transaction framework
// extension implementation
return new ExtDBTransactionImpl2();
```

3. Load the `ExtDBTransactionImpl2.java` file in the Java editor, add an `ADFLogger` to it and override the `commit()` and `rollback()` `DBTransactionImpl2` methods. The code should look like this:

```
// create an ADFLogger
private static final ADFLogger LOGGER =
   ADFLogger.createADFLogger(ExtDBTransactionImpl2.class);

public void commit() {
    // log a trace
    LOGGER.info("Commit was called on the transaction");
    super.commit();
}

public void rollback() {
    // log a trace
    LOGGER.info("Rollback was called on the transaction");
    super.rollback();
}
```

4. Rebuild and redeploy the shared components workspace into an ADF Library JAR.
5. Open the `HRComponents` workspace and open the **HrComponentsAppModule** Application Module definition by double-clicking on it in the **Application Navigator**.
6. Go to the **Configurations** section.
7. Select the **HrComponentsAppModuleLocal** configuration and click the **Edit selected configuration object** button (the pen icon).



8. In the **Edit Configuration** dialog click on the **Properties** tab and locate the **TransactionFactory** property. For the property value enter the custom transaction framework extension class **com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory**.

How it works...

In steps 1 and 2 we have overridden the `DatabaseTransactionFactory create()` method for the custom transaction factory framework class `ExtDatabaseTransactionFactory` that we created in recipe *Setting up BC base classes* recipe in chapter 1, so that we return our custom transaction implementation class `ExtDBTransactionImpl2`. This informs the ADF Business Components framework that a custom `oracle.jbo.server.DBTransaction` implementation will be used. Then, in step 3 we provided custom implementations for our `ExtDBTransactionImpl2` transaction commit and rollback operations. In this case, we have provided a global transaction logging facility for all commit and rollback operations throughout the ADF application for Application Modules utilizing our custom `DBTransaction` implementation. We then rebuilt and redeployed the shared components workspace (step 4).

In steps 5 through 8, we have explicitly indicated in the `HrComponentsAppModule` local configuration, the one used to configure session-specific Application Modules, that a custom transaction factory will be used. We did this by setting the `TransactionFactory` configuration property to our custom transaction factory implementation class `com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory`.

There's more...

In order to change Application Module configuration parameters for all Application Modules throughout your ADF application, adopt the practice of using Java system-defined properties via the `-D` switch at JVM start-up. In this case ensure that no specific configuration parameters are defined for individual Application Modules, unless needed, as they would override the values specified globally with the `-D` Java switch. You can determine the specific parameter names that you must specify with the `-D` switch, from the **Property** field in the **Edit Configuration** dialog. For example, for this recipe, you will specify `-DTransactionFactory="com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory"` at JVM start-up to indicate that the custom transaction factory will be used. For WebLogic, these start-up parameters can be specified via the `JAVA_OPTIONS` environment variable or in any of the WebLogic start-up scripts (`setDomainEnv.*`, `startWebLogic.*`, `startManagedWebLogic.*`).

See also

Setting up BC base classes, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

6

Go with the Flow: Task Flows

In this chapter, we will cover:

- Using an Application Module function to initialize a page
- Using a task flow Initializer to initialize a task flow
- Calling a task flow as a URL programmatically
- Programmatically navigate to an action using NavigationHandler
- Retrieving the task flow definition programmatically using MetadataService
- Creating a train

Introduction

Task flows are used for designing the ADF Fusion Web Application's control flow. They were introduced with the advent of the JDeveloper 11g R1 release as an alternative to standard JSF navigation flows. As such, they allow for the decomposition of monolithic application navigation flows (as in the case of JSF navigation flows) into modular, transaction and memory scope aware controller flow components. The ADF Fusion Web Application is now composed of numerous task flows, called bounded task flows, most likely residing in various ADF Library JARs, calling each other in order to construct the application's overall navigation flow.

While in the traditional JSF navigation flow navigation occurs between pages, task flows introduce navigation between activities. A task flow activity is not necessarily a visual page component (View Activity) as in the case of JSF navigation flows. It can be a call to Java code (Method Call Activity), the invocation of another task flow (Task Flow Call Activity), a control flow decision (Router Activity) and so on. This approach provides a high degree of flexibility, modularity and reusability when designing the application's control flow.

Using an Application Module function to initialize a page

A common use case when developing an ADF Fusion Web application is to be able to do some sort of initialization before a particular page of the application is shown. Such an initialization could be the creation of a new View object row, which will in effect place the View object in insert mode, the execution of a View object query, which could populate a table on the page, the execution of a database stored procedure and so on. This can easily be accomplished by utilizing a **Method Call Activity**.

In this recipe we will demonstrate the usage of the Method Call task flow Activity by implementing the all familiar use case of placing a web page in insert mode. Before the page is presented, a custom Application Module method (implemented in another workspace) will be called to place the View object in insert mode.

Getting ready

You will need a skeleton Fusion Web Application (ADF) workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

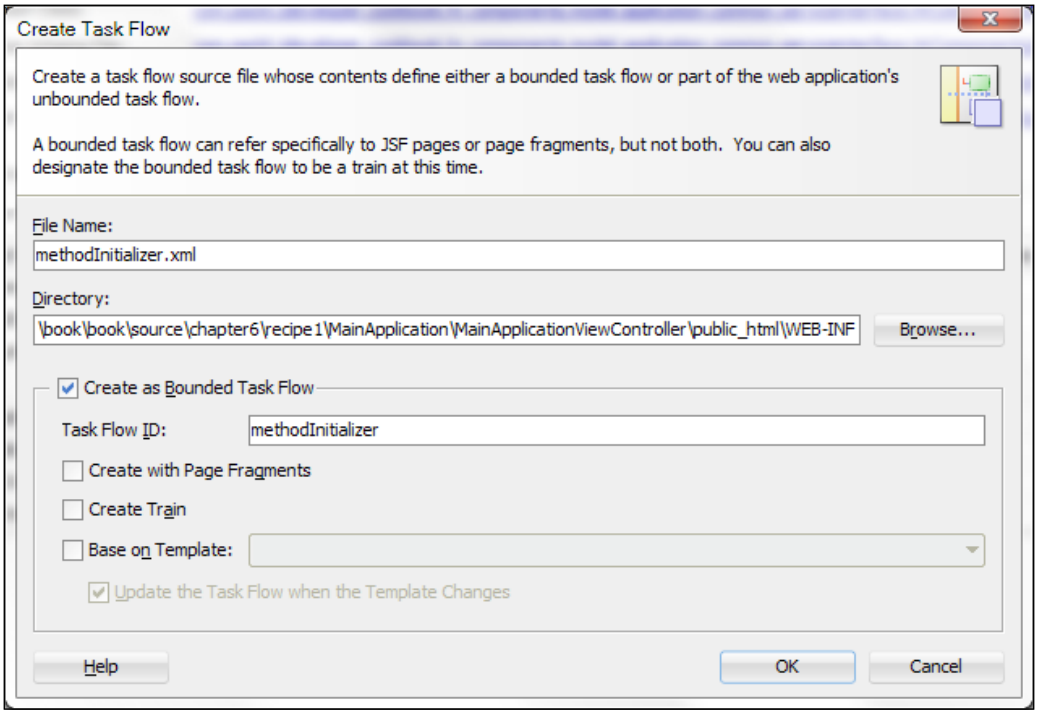
How to do it...

1. Open the shared components workspace in JDeveloper.
2. Load the `HrComponentsAppModuleImpl` custom Application Module implementation class into the Java editor and add the following `prepare()` method to it:

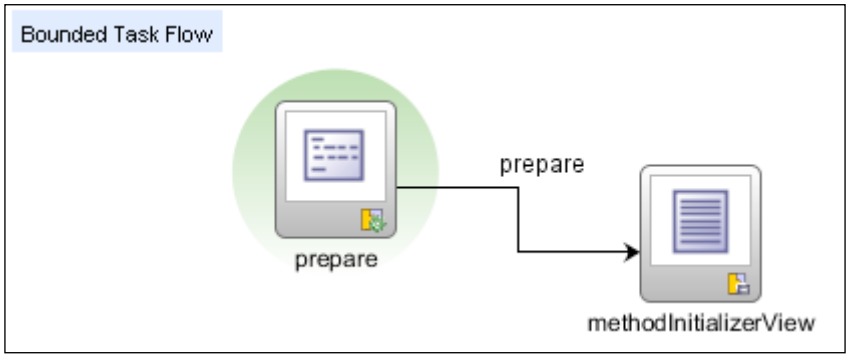
```
public void prepare() {
  // get the Employees view object instance
  EmployeesImpl employees = this.getEmployees();
  // remove all rows from rowset
  employees.executeEmptyRowSet();
  // create a new employee row
  Row employee = employees.createRow();
```

```
        // add the new employee to the rowset
        employees.insertRow(employee);
    }
```

3. Open the `HrComponentsAppModule` Application Module definition and go to the **Java** section. Click on the **Edit application module client interface** button (the pen icon) and shuttle the **prepare()** method from the **Available** methods list to the **Selected** list.
4. Rebuild and redeploy the `HRComponents` workspace into an ADF Library JAR.
5. Now open the `MainApplication` workspace and using the **Resource Palette** create a new **File System** connection to the `ReUsableJARs` directory where the `HRComponents.jar` ADF Library JAR is placed. Select the **ViewController** project in the **Application Navigator** and then right-click on the `HRComponents.jar` ADF Library JAR in the **Resource Palette**. From the context menu select **Add to Project...**.
6. Right-click on the **ViewController** project in the **Application Navigator** and select **New...**. Select **ADF Task Flow** from the **Web Tier > JSF/Facelets** category.
7. In the **Create Task Flow** dialog enter **methodInitializer.xml** for the task flow **File Name** and ensure that you have selected the **Create as Bounded Task Flow** checkbox. Also make sure that the **Create with Page Fragments** checkbox is not selected. Then click **OK**.



8. The `methodInitializer` task flow should open automatically in **Diagram** mode. If not, double-click on it in the **Application Navigator** to open it. Click anywhere in the task flow and in the **Property Inspector** change the **URL Invoke** property to **url-invoke-allowed**.
9. Expand the **Data Controls** section In **Application Navigator**, locate and expand the **HrComponentsAppModuleDataControl** data control. Find the **prepare()** method and drag and drop it onto the `methodInitializer` task flow. JDeveloper will create a Method Call activity called **prepare**.
10. Drag and a drop a **View** activity from the **Component Palette** onto the `methodInitializer` task flow.
11. Using the **Component Palette** create a **Control Flow Case** from the **prepare** Method Call activity to the View activity.
12. Ensure that the **prepare** Method Call activity is marked as the default task flow activity by clicking on the **Mark Default Activity** button in the toolbar. The task flow should look like this:



13. Double-click on the View activity to bring up the **Create JSF Page** dialog. In it, select **JSP XML** for the **Document Type**. For the **Page Layout** you may select any of the **Quick Start Layout** options. Click **OK**. The page should open automatically in **Design** mode. If not, double-click it in the **Application Navigator** to open it.
14. Expand the **Data Controls** section in the **Application Navigator** and locate the **Employees** View object under the **HrComponentsAppModuleDataControl**. Drag and drop the **Employees** View object onto the page.
15. From the **Create** context menu select **Form > ADF Form...**. This will present the **Edit Form Fields** dialog. Click **OK** to accept the defaults and proceed with the creation of the ADF form.

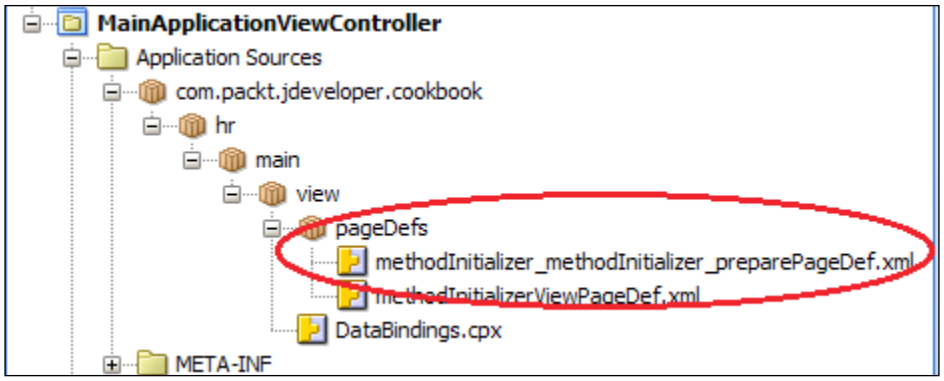
How it works...

In steps 1 through 4 we added a method called `prepare()` to the `HrComponentsAppModule` Application Module residing in the `HRComponents` workspace. In the `prepare()` method we retrieve the `Employees` View object instance by calling the `getEmployees()` getter, and call `executeEmptyRowSet()` on the View object to empty its rowset. We then create an employee row by calling `createRow()` on the `Employees` View object, and add the new row to the `Employees` View object rowset by calling `insertRow()` and passing the newly created employee row as an argument to it. This will in effect place the `Employees` View object in insert mode. We exposed the `prepare()` method to the Application Module client interface (in step 3), so that we will be able to call this method via the bindings layer using a task flow Method Call activity. Then (in step 4), we rebuilt and redeployed the `HRComponents` workspace to an ADF Library JAR. This will allow us to import the ADF components implemented in the ADF Library JAR to other projects throughout the ADF application.

In order to be able to reuse the components defined and implemented in the `HRComponents` ADF Library JAR we created a File System connection using the Resource Palette in JDeveloper and added the library to our main project. This was done in step 5.

In steps 6 through 8 we created a bounded task flow called `methodInitializer` and we ensured (in step 8) that its `URL Invoke` property was set to `url-invoke-allowed`. We need to do this because the Method Call activity that is added in step 9 to call the `prepare()` method in the `HrComponentsAppModule` Application Module will be indicated as the default task flow activity (in step 12). In this case, leaving the default setting of `calculated` for the `URL Invoke` property will produce an *HTTP 403* error.

In step 9 we located the `prepare()` method in the `HrComponentsAppModuleDataControl` data control in the Data Controls section of the **Application Navigator**, and we drag and dropped it onto the task flow. This creates the Method Call activity and the necessary bindings to bind the Method Call activity to the `prepare()` method in the `HrComponentsAppModule` Application Module. By default the page definition is placed in the `pageDefs` package under the default package defined for the ViewController project (`com.packt.jdeveloper.cookbook.hr.main.view` in this case). Note that the `HrComponentsAppModuleDataControl` data control becomes available once the `HRComponents` ADF Library JAR is added to the project.

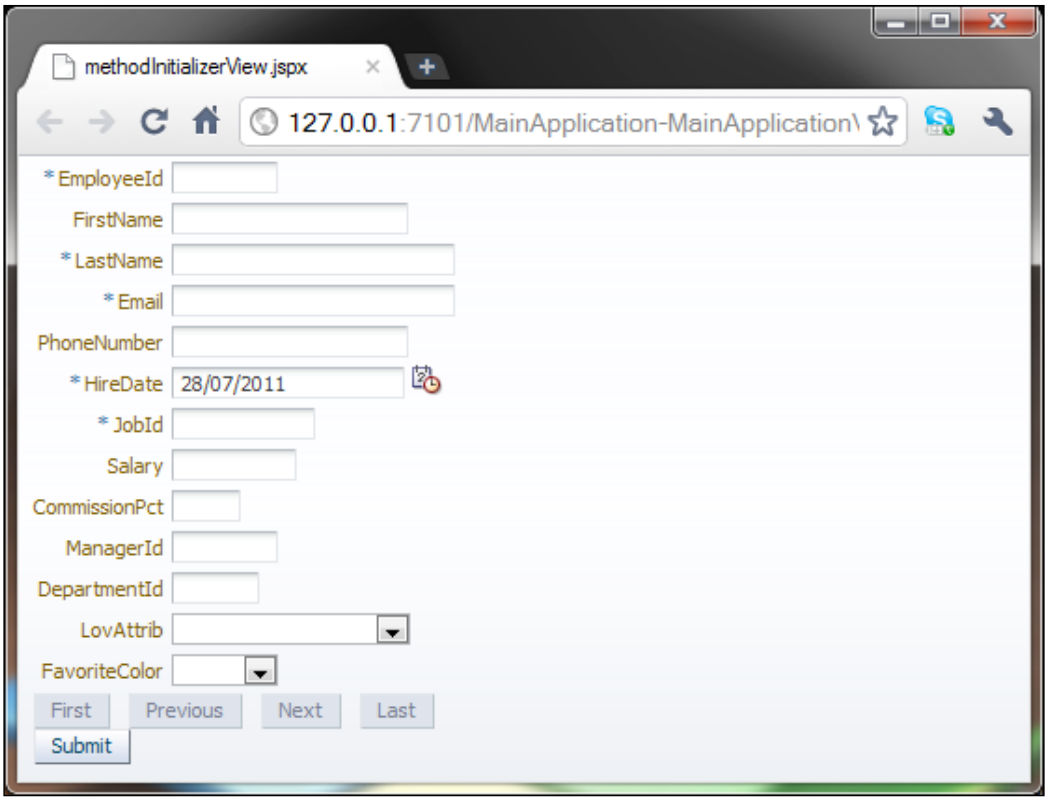


In steps 10 and 11 we placed a View activity onto the task flow and added a Control Flow Case (called `prepare` by default) to allow the transition from the `prepare` Method Call activity to the View activity.

The definition of the task flow was completed by ensuring that the `prepare` Method Call activity was marked as the default task flow activity (step 12). This indicates that it will be the first activity to be executed in the task flow.

Finally, in steps 13 through 15 we created a JSP XML (JSPX) page for the task flow View activity and added an ADF Form to it for the Employees view object. We did this by drag and dropping the `Employees` View object from the `HrComponentsAppModuleDataControl` data control onto the JSPX page.

To test the recipe, right-click on the **methodInitializer** task flow in the **Application Navigator** and select **Run** or **Debug** from the context menu. This will build and deploy the workspace into the integrated WebLogic application server. As you can see, the `prepare` Method Call activity is called prior to transitioning to the View activity. The effect of calling the `prepare()` method is to place the `Employees` View object in insert mode.



There's more...

Using a task flow Method Call activity is one of the possible ways to do an initialization before displaying a page. Another approach, explained in more detail in the *Using a task flow Initializer for task flow initialization* recipe in this chapter, is to use a task flow **Initializer**. The difference between the two is that the task flow Initializer is called once during the instantiation of the task flow, while multiple Method Call activities may be placed anywhere in the task flow. Also consider the definition of an **invokeAction** in the page definition to do page-specific initialization. As a best practice, consider using either a Method Call task flow activity or a task flow Initializer since they are highly abstracted and less coupled. Using an invokeAction on the other hand would be more appropriate if you want the initialization method to be executed for multiple phases of the page's lifecycle.

Furthermore note the signature of the `prepare()` method we used in this recipe for initializing the page, i.e. `public void prepare()`. It does not accept any parameters and it returns nothing. If your initialization method requires a number of parameters to be specified, they can be specified either in the **Parameters** section of the **Edit Action Binding** dialog (for a data control bound method) or otherwise using the **Parameters** section in the Method Call **Property Inspector**. In either case, the parameter values are usually communicated via the pageFlowScope. Finally, based on the return type of your initialization method you could set the value of the **toString()** outcome in the **Outcome** section (in the **Property Inspector**) and allow further processing of the return value, using for instance a **Router** activity. When returning `void`, the outcome must be Fixed and **toString()** cannot be used (must be set to `false`).

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using a task flow Initializer to initialize a task flow, chapter 6.

Using a task flow Initializer to initialize task flow

In the *Using an Application Module function to initialize a page* recipe in this chapter we demonstrated how to use a method residing in the Application Module to perform page initialization by bounding the method as a Method Call activity in the task flow. This recipe shows a different way to accomplish the same task by using a task flow Initializer method instead. Unlike the Method Call activity, which once bound to the task flow may be called multiple times in the task flow, the Initializer method is called once during the task flow initialization.

Getting ready

You will need a skeleton Fusion Web Application (ADF) workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

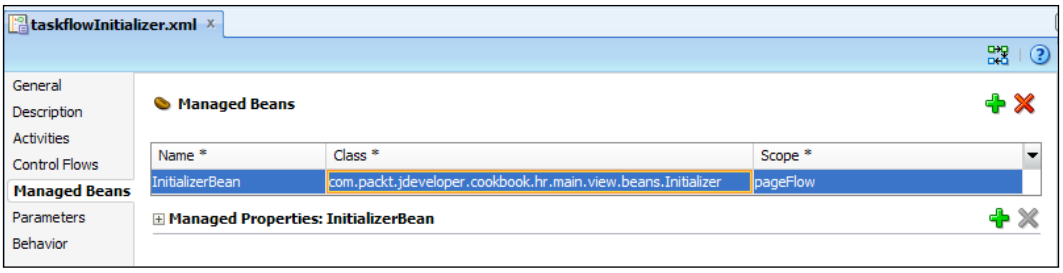
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

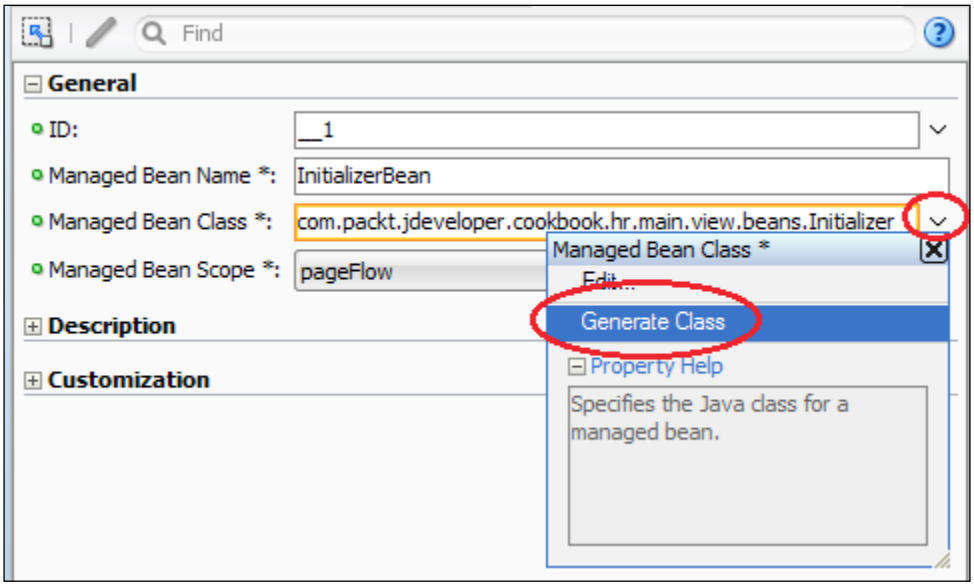
1. Open the shared components workspace in JDeveloper.
2. Load the `HrComponentsAppModuleImpl` custom Application Module implementation class into the Java editor and add the following method to it:

```
public void prepare() {
    // get the Employees view object instance
    EmployeesImpl employees = this.getEmployees();
    // remove all rows from rowset
    employees.executeEmptyRowSet();
    // create a new employee row
    Row employee = employees.createRow();
    // add the new employee to the rowset
    employees.insertRow(employee);
}
```

3. Open the `HrComponentsAppModule` Application Module definition and go to the **Java** section. Click on the **Edit application module client interface** button (the pen icon) and shuttle the **prepare()** method from the **Available** methods list to the **Selected** list.
4. Rebuild and redeploy the `HRComponents` workspace into an ADF Library JAR.
5. Now open the `MainApplication` workspace and using the **Resource Palette** create a new **File System** connection to `ReUsableJARs` directory where the `HRComponents.jar` ADF Library JAR is placed. Select the **ViewController** project in the **Application Navigator** and then right-click on the `HRComponents.jar` ADF Library JAR in the **Resource Palette**. From the context menu select **Add to Project...**.
6. Right-click on the **ViewController** project in the **Application Navigator** and select **New...**. Select **ADF Task Flow** from the **Web Tier > JSF/Facelets** category.
7. In the **Create Task Flow** dialog enter **taskflowInitializer.xml** for the task flow **File Name** and ensure that you have selected the **Create as Bounded Task Flow** checkbox. Also make sure that the **Create with Page Fragments** checkbox is not selected. Then click **OK**.
8. The `taskflowInitializer` task flow should open automatically in **Diargam** mode. If not, double-click on it in the **Application Navigator** to open it. Click anywhere in the task flow and in the **Property Inspector** change the **URL Invoke** property to **url-invoke-allowed**.
9. Go to the task flow **Overview > Managed Beans** section and add a managed bean called **InitializerBean**. Enter **com.packt.jdeveloper.cookbook.hr.main.view.beans.Initializer** for the managed bean **Class** and select **pageFlow** for the bean **Scope**.



10. While at the **Managed Beans** section, select **Generate Class** from the **Property Menu** next to the **Managed Bean Class** in the **Property Inspector**.

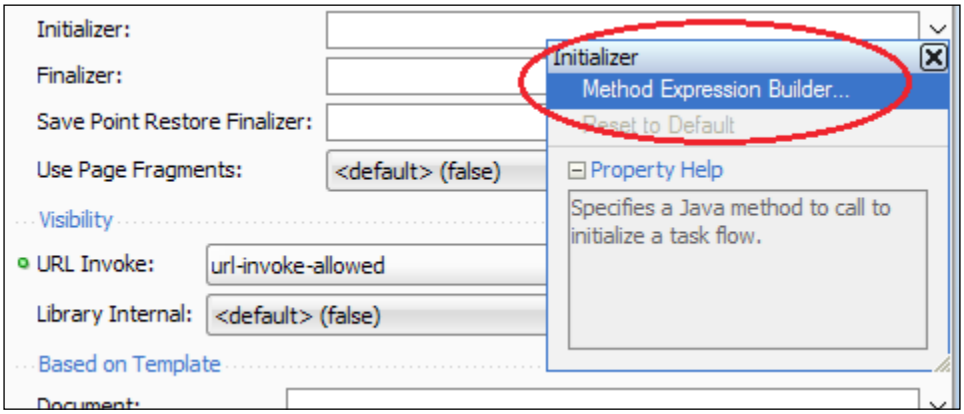


11. Locate the **Initializer.java** bean in the **Application Navigator** and open it in the Java editor. Add the following `initialize` method to it:

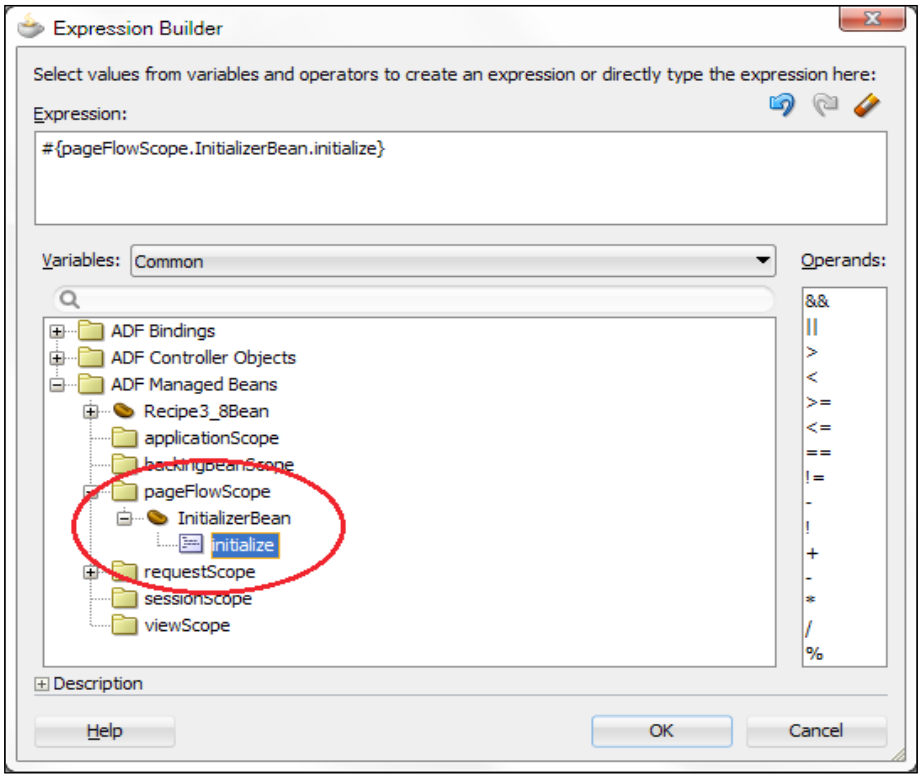
```
public void initialize() {
  // get the Application Module
  HrComponentsAppModule hrComponentsAppModule =
      (HrComponentsAppModule)ADFUtils
        .getApplicationModuleForDataControl(
          "HrComponentsAppModuleDataControl");
  if (hrComponentsAppModule != null) {
    // call the initializer method
```

```
        hrComponentsAppModule.prepare();
    }
}
```

12. Return to the task flow **Diagram** and add a task flow initializer by clicking on the **Property Menu** next to the **Initializer** property in the **Property Inspector** and selecting **Method Expression Builder...**.



13. In the **Expression Builder** dialog that opens locate and select the **initialize** method of the **InitilizerBean** under the **ADF Managed Beans** node. The click **OK** to dismiss the dialog. The initializer expression `#{pageFlowScope.InitializerBean.initialize}` should be reflected in the **Initializer** property of the task flow in the **Property Inspector**.



14. Drag and a drop a **View** activity from the **Component Palette** onto the `taskflowInitializer` task flow.
15. Double-click on the View activity to bring up the **Create JSF Page** dialog. In it, select **JSP XML** for the **Document Type**. For the **Page Layout** you may select any of the **Quick Start Layout** options. Click **OK**. The page should open automatically in **Design** mode. If not, double-click it in the **Application Navigator** to open it.
16. Expand the **Data Controls** section in the **Application Navigator** and locate the **Employees** View object under the **HrComponentsAppModuleDataControl**. Drag and drop the **Employees** View object onto the page.
17. From the **Create** context menu select **Form > ADF Form...**. This will present the **Edit Form Fields** dialog. Click **OK** to accept the defaults and proceed with the creation of the ADF form.

How it works...

Steps 1 through 8 have been thoroughly explained in the *Using an Application Module function to initialize a page* recipe in this chapter, so we won't get into the specific details here.

In steps 9 and 10 we defined a managed bean called `InitializerBean` and we generated a Java class for it. We used `pageFlow` for the bean's memory scope. This will ensure that the `InitializerBean` bean will persist throughout the task flow's execution.

In step 11 we added an `initialize()` to the `InitializerBean` bean. This will be the method indicated as the task flow Initializer in steps 12 and 13. Inside the `initialize()` method we get hold of the `HrComponentsAppModule` by utilizing the `ADFUtils.getApplicationModuleForDataControl()` helper method. We introduced the `ADFUtils` helper class back in chapter 1 in the *Using ADFUtils/JSFUtils* recipe. We have packaged the `ADFUtils` helper class inside the shared components ADF Library JAR (`SharedComponents.jar`), which is imported into the project in step 5. The `getApplicationModuleForDataControl()` method returns an `oracle.jbo.ApplicationModule` interface, which we then cast to our specific `HrComponentsAppModule` custom Application Module interface. Through the `HrComponentsAppModule` interface we call the `prepare()` method to do the necessary initializations. We explained the logic in `prepare()` in the *Using an Application Module function to initialize a page* recipe in this chapter.

In steps 12 and 13 we declaratively setup the task flow Initializer property using the Expression Language expression `#{pageFlowScope.InitializerBean.initialize}`. This expression indicates that the `initialize()` method of the `InitializerBean` is called during the instantiation of the task flow.

Finally, in steps 14 through 17 we defined a View activity and the corresponding JSPX page. Again, we explained these steps in more detail in the *Using an Application Module function to initialize a page* recipe in this chapter.

To test the recipe, right-click on the **taskFlowInitializer** task flow in the **Application Navigator** and select **Run** from the context menu. This will build and deploy the workspace into the integrated WebLogic application server. The page displayed in the browser will be presented in insert mode, since the task flow Initializer method calls the Application Module `prepare()` method to set the `Employees` View object in insert mode.

There's more...

Both this technique and the one presented in the *Using an Application Module function to initialize a page* in this chapter may be used to run task flow initialization code. Note however one difference pertaining to their handling of the Web browser's Back button. While the task flow Initializer approach calls the Initializer method upon reentry via the browser's Back button, no task flow initialization code is called when reentering the task flow via the browser's back button in the Method Call activity approach. This behavior however, seems to be inconsistent among browsers. For more information about this, refer to section *About Creating Complex Task Flows* in the *Fusion Developer's Guide for Oracle Application Framework.*

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using an Application Module function to initialize a page, chapter 6.

Calling a task flow as a URL programmatically

A task flow that is indicated as URL invoke-able (by setting its `visibility` attribute `url-invoke-allowed` to `true`), may be accessed directly by constructing and invoking its URL. This can be done programmatically by using the `oracle.adf.controller.ControllerContext.getTaskFlowURL()` method and specifying the task flow identifier and parameters.

For this recipe, to demonstrate calling a task flow via its URL, we will create a task flow that is URL invoke-able and call it from a JSPX page programmatically. The task flow accepts a parameter and based on the parameter's value determines whether to call any of the `methodInitializer` or `taskflowInitializer` task flows. These task flows were developed in the *Using an Application Module function to initialize a page* and *Using a task flow Initializer to initialize a task flow* recipes respectively in this chapter.

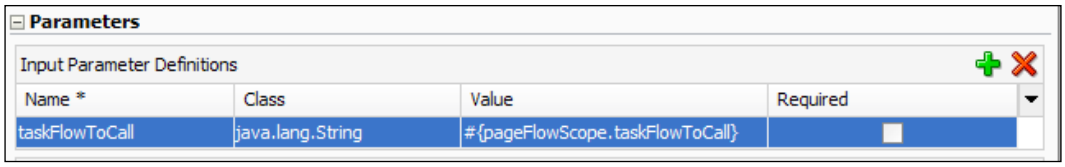
Getting ready

You need to have access to the `methodInitializer` and `taskflowInitializer` task flows that were developed in the *Using an Application Module function to initialize a page* and *Using a task flow Initializer to initialize a task flow* recipes in this chapter. Also, note the additional prerequisites stated for those recipes, i.e. the usage of the `HRComponents` and `MainApplication` workspaces and the database connection to the `HR` schema.

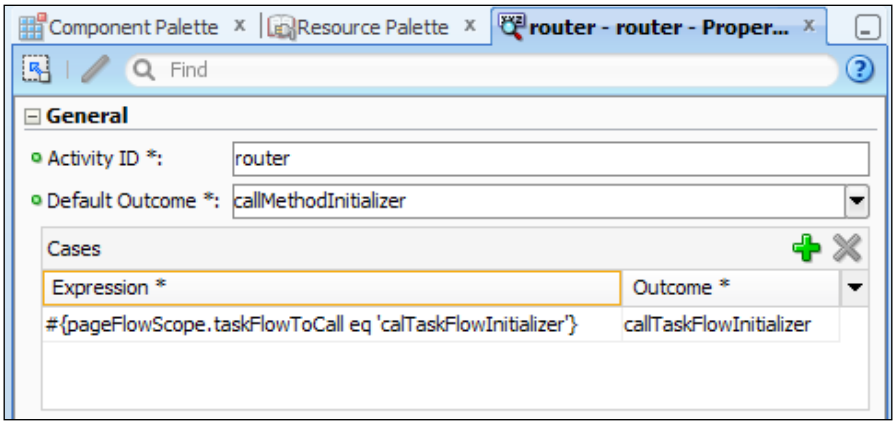
How to do it...

1. Start by creating a new task flow called `programmaticallyInvokeTaskFlow`. Ensure that you create it as a bounded task flow and that it is not created with page fragments.
2. In the **Visibility** section in the task flow **Property Inspector**, make sure that the **URL Invoke** attribute is set to **url-invoke-allowed**.

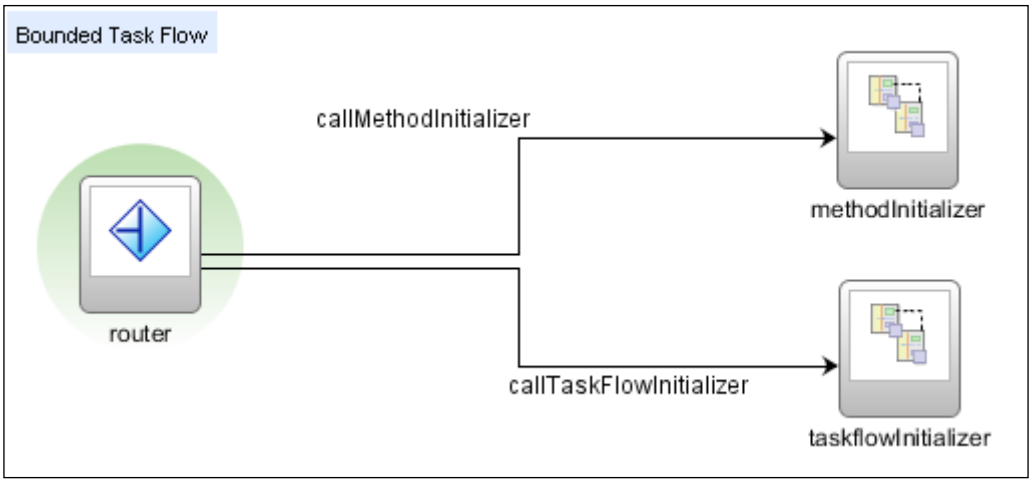
3. While at the task flow **Property Inspector**, in the **Parameters** section add a parameter called **taskFlowToCall** of type **java.lang.String**. For the parameter **Value** enter **#{pageFlowScope.taskFlowToCall}**.



4. From the **Component Palette** drop a **Router** activity on the task flow.
5. Locate the **methodInitializer** and **taskflowInitializer** task flows in the **Application Navigator** and drop them on the task flow.
6. Using the **Component Palette**, create control flow cases from the Router activity to **methodInitializer** and **taskflowInitializer** task flow calls. Call these control flow cases **callMethodInitializer** and **callTaskFlowInitializer** respectively.
7. Now, select the Router activity and in the **Property Inspector** set its **Default Outcome** property to **callMethodInitializer. Also, add the following expression #{pageFlowScope.taskFlowToCall e.g. 'calTaskFlowInitializer'} in the Cases section and callTaskFlowInitializer** as the expression's **Outcome** value. The router's properties in the Property Inspector should look like this:



8. The complete **programmaticallyInvokeTaskFlow** task flow should look like this:



9. Now, locate the **adfc-config.xml** unbounded task flow in the **Application Navigator** and double-click on it to open it. Go to the **Overview > Managed Beans** section and add a **TaskFlowURLCallerBean** managed bean. Specify **com.packt.jdeveloper.cookbook.hr.main.view.beans.TaskFlowURLCaller** for the bean **Class** and leave the default **request** for the bean's **Scope**.
10. Create the managed bean by selecting **Generate Class** from the **Property Menu** in the **Property Inspector**, next to the **Managed Bean Class** attribute.
11. Locate the **TaskFlowURLCallerBean** bean in the **Application Navigator** and double-click it to open it in the Java editor. Add the following methods to it:

```
public String getProgrammaticallyInvokeTaskFlow() {
        // setup task flow parameters
 Map<String, Object> parameters =
         new java.util.HashMap<String, Object>();
 parameters.put("taskFlowToCall", "calTaskFlowInitializer");
 // construct and return the task flow's URL
 return getTaskFlowURL("/WEB-INF/programmaticallyInvokeTaskFlow.xml#programm
aticallyInvokeTaskFlow", 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, "");
}
```

12. Finally, create a JSPX page called **taskFlowURLCaller.jspx** and drop a **Link (Go)** component on it from the **Component Palette**. Specify the link's text, destination and targetFrame properties as it is shown below:

```
<af:goLink text="Call programmaticallyInvokeTaskFlow as a URL"
  id="gl1"
  destination="#{TaskFlowURLCallerBean.programmaticallyInvokeTaskFlow}"
  targetFrame="_blank"/>
```

How it works...

In steps 1 through 3 we created a task flow called `programmaticallyInvokeTaskFlow` and set its visibility to `url-invoke-allowed`. This will allow us to call the task flow via a URL. If we don't do this, a security exception will be thrown when trying to access the task flow via a URL. We also added (in step 3) a single task flow parameter called `taskFlowToCall` to indicate which task flow to call once our `programmaticallyInvokeTaskFlow` is executed. We store the value of this parameter to a pageFlow scope variable called `taskFlowToCall`. This pageFlow scope parameter is accessible via the EL expression `#{pageFlowScope.taskFlowToCall}`. We will see in step 7 how this pageFlow scope variable is accessed to determine the subsequent task flow to call.

In steps 4 through 8 we completed the task flow definition by adding a Router activity and two task flow call activities, one for each of the `callMethodInitializer` and `callTaskFlowInitializer` task flows. Note in step 5 how we just dropped on to our task flow definition the `callMethodInitializer` and `callTaskFlowInitializer` task flows from Application Navigator, to create the task flow calls. Also, observe in step 6 how we created the control flow cases to connect the Router activity with each of the task flow call activities. Finally, note how in step 7 we configured the Router activity outcomes based on the value of the input task flow parameter `taskFlowToCall`. Specifically we checked the parameter's value using the EL expression `#{pageFlowScope.taskFlowToCall e.g. 'calTaskFlowInitializer'}`. In this case the router's outcome was set to `callTaskFlowInitializer`, which calls the `taskflowInitializer` task flow. In any other case, we configured the default router outcome to be `callMethodInitializer`, which calls the `methodInitializer` task flow.

In steps 9 through 11 we configured a globally accessible managed bean called `TaskFlowURLCallerBean`, by adding it to the application's unbounded task flow `adfc-config.xml`. We generated the bean class in step 10 and in step 11 we added the necessary code to be able to call our `programmaticallyInvokeTaskFlow` task flow programmatically. The specific details about this code follow.

We introduced two methods in the `TaskFlowURLCallerBean`. One called `getProgrammaticallyInvokeTaskFlow()`, which will be called from a page component to return the task flow's URL (see step 12) and another one, called `getTaskFlowURL()`, a helper method to do the actual work of determining and returning the task flow's URL. We call `getTaskFlowURL()` indicating the task flow specification and its parameters. Observe in `getProgrammaticallyInvokeTaskFlow()`, how we specify the parameter value and the task flow specifications. In `getTaskFlowURL()` we obtain an `oracle.adf.controller.TaskFlowId` from the task flow specifications, and then call the `oracle.adf.controller.ControllerContext.getTaskFlowURL()` method to retrieve the task flow URL. Once the task flow URL is returned, we strip the application's context path from it before returning it. This is something that we need to do before calling the task flow via a URL.

The final part of the implementation is done in step 12. In this step we created a new JSPX page, called `taskFlowURLCaller.jspx`, and added an `af:golink` ADF Faces UI component to it. We will use the go link to programmatically call our `programmaticallyInvokeTaskFlow` task flow via the URL returned by the `getProgrammaticallyInvokeTaskFlow()` method defined in the `TaskFlowURLCallerBean`. We do this by setting the destination attribute of the `af:golink` component to `#{TaskFlowURLCallerBean.programmaticallyInvokeTaskFlow}`. We have also indicated `_blank` for the go link `targetFrame` attribute, so that the called task flow opens in a new browser frame.

To test the recipe right-click on the **taskFlowURLCaller.jspx** JSPX page in the **Application Navigator** and select **Run** or **Debug** from the context menu.

There's more...

When calling a task flow programmatically via its URL, always use the ADF Controller API indicated in this recipe to obtain the task flow's URL. Do not hardcode the task flow's URL in your application or in database tables, as the specifications of calling task flows in the ADF framework may change unexpectedly in the future.

See also

Using an Application Module function to initialize a page, chapter 6.

Using a task flow Initializer to initialize a task flow, chapter 6.

Programmatically navigate to an action using NavigationHandler

When the ADF application navigation cannot be determined during design time, you can programmatically navigate to an action using a JSF `NavigationHandler`. In this case, the navigation action is determined dynamically at runtime based on the application logic.

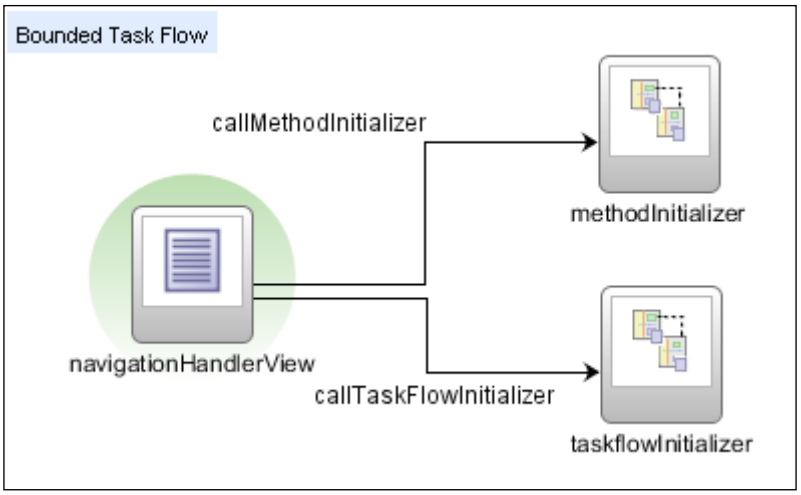
In this recipe we will demonstrate how to perform dynamic navigation by implementing the following use case: we will navigate to a specific action based on a particular user interface choice made. Specifically, we will present a list of navigation choices and based on the selection a specific navigation will be invoked.

Getting ready

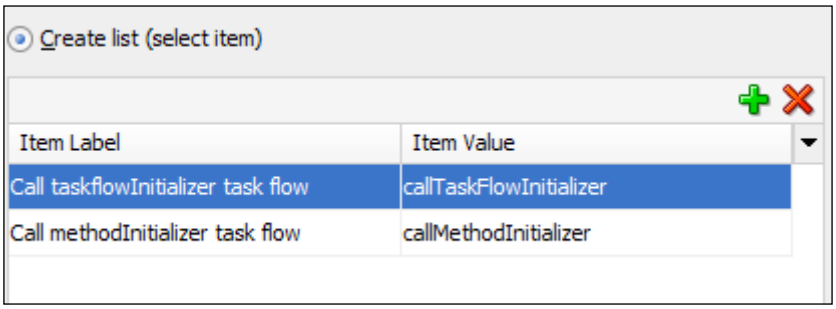
You need to have access to the `methodInitializer` and `taskflowInitializer` task flows that were developed in the *Using an Application Module function to initialize a page* and *Using a task flow Initializer to initialize a task flow* recipes in this chapter. Also, note the additional prerequisites stated for those recipes, i.e. the usage of the `HRComponents` and `MainApplication` workspaces and the database connection to the `HR` schema.

How to do it...

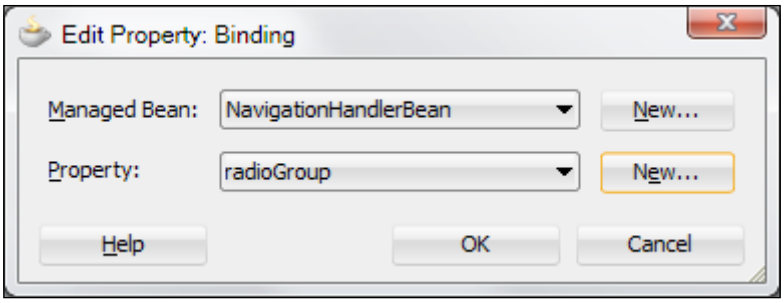
1. Create a bounded task flow called **navigationHandler**. Ensure that the **Create with Page Fragments** checkbox is not selected.
2. Create a **View** activity called **navigationHandlerView**. Also locate the **methodInitializer** and **taskflowInitializer** task flows in the **Application Navigator** and drop them onto the **navigationHandler** task flow.
3. Create control flow cases from the **navigationHandlerView** View activity to the **methodInitializer** and **taskflowInitializer** task flow call activities. Call these control flow cases **callMethodInitializer** and **callTaskFlowInitializer** respectively. The task flow should look like this:



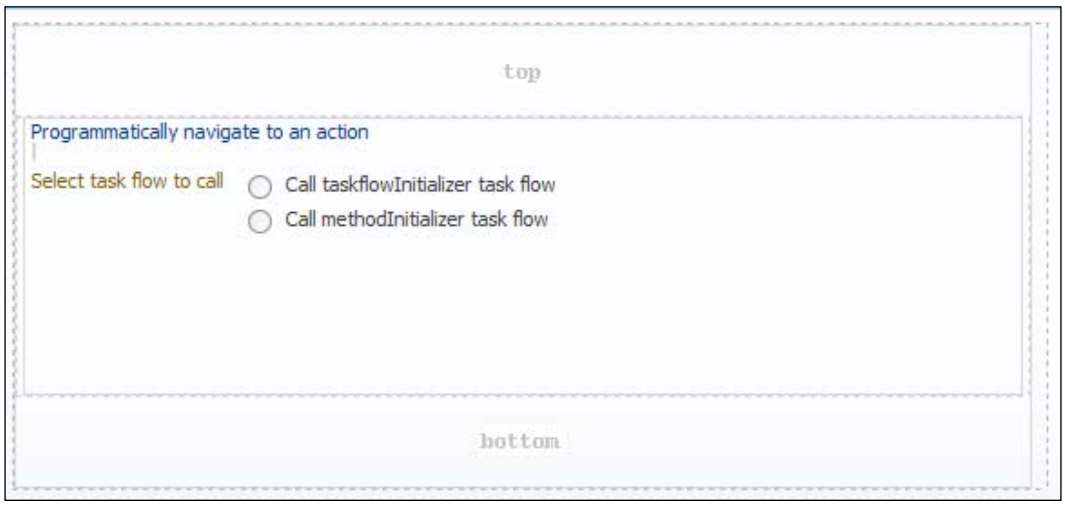
4. To complete the task flow definition, go to the **Overview > Managed Beans** section and create a managed bean called **NavigationHandlerBean**. Also generate the managed bean class.
5. Now, double-click on the **navigationHandlerView** View activity and create a JSPX page called **navigationHandler.jspx**. If the **navigationHandler.jspx** page is not shown, double-click on in the **Application Navigator** to open it.
6. From the **ADF Faces Component Palette**, locate and drop onto the page a **Radio Group** component. In the **Insert Radio Group** dialog, click on the **Create list (select item)** radio and add the following items shown below:



7. Go to the radio group **Property Inspector** and using the **Binding Property Menu** next to the **Binding** attribute, create a binding for the radio group component. On the **Edit Property: Binding** dialog that is displayed, select **NavigationHandlerBean** for the **Managed Bean** field and click on the **New...** button to create a new property for the **Property** field.



8. From the **ADF Faces Component Palette**, locate and drop on the page a **Link** component. Go to the **Property Inspector** for the link component and add an action listener by selecting **Edit...** from the **ActionListener Property Menu** next to the **ActionListener** attribute. This will display the **Edit Property: ActionListener** dialog. In it select **NavigationHandlerBean** for the **Managed Bean** field and click on the **New...** button to create a new method for the **Method** field. The method we specified for this recipe is called **onLinkActionListener**. The JSPX page at this point should look something like this:



9. Open the **NavigationHandlerBean** managed bean in the Java editor and locate the `onLinkActionListener()` action listener method. Add the following code to it:

```
// get from outcome
String fromOutcome = (String)radioGroup.getValue();
// if valid outcome, navigate to it
if (fromOutcome != null) { // get JSF navigation handler
```

```
FacesContext context = JSFUtils.getFacesContext();
NavigationHandler handler =
        context.getApplication().getNavigationHandler();
// handle navigation
handler.handleNavigation(context, null, fromOutcome);
}
```

How it works...

In steps 1 through 4 we defined a task flow called `navigationHandler` comprised of one view and two task flow call activities. For the task flow call activities we re-used the task flows implemented earlier in the chapter. We created the necessary control flow cases to connect the activities in step 3 and in step 4 we added a managed bean called `NavigationHandlerBean` to the task flow.

In steps 5 through 8 we created the JSPX page associated with the task flow View activity. We added two ADF Faces user interface components to it:

- an `af:selectOneRadio` Radio Group component, comprised of two `af:selectItem` components. For the sake of this recipe, we have provided hard-coded values for the `label` and `value` attributes. Note, that we have indicated the specific `navigationHandler` task flow From Outcomes `callTaskFlowInitializer` and `callMethodInitializer` for the values. Note also in step 7 how we bound the `af:selectOneRadio` Radio Group component to the `NavigationHandlerBean` managed bean that we defined earlier. This binding adds a `binding="#{NavigationHandlerBean.radioGroup}"` attribute to the `af:selectOneRadio` component and the following code to the `NavigationHandlerBean` managed bean:

```
private RichSelectOneRadio radioGroup;

public void setRadioGroup(RichSelectOneRadio radioGroup) {
    this.radioGroup = radioGroup;
}

public RichSelectOneRadio getRadioGroup() {
    return radioGroup;
}
```

- an `af:commandLink` Link component to navigate to the selected task flow. In step 8 where we defined the Link component, we also indicated an actionListener to the `NavigationHandlerBean` managed bean `onLinkActionListener` method. This is the method that will be execute in response to an action on the Link.

Finally in step 9 we implemented the `onLinkActionListener` action listener. First we retrieved the currently selected value of the Radio Group component. This was done by calling `getValue()` on the bound `radioGroup RichSelectOneRadio` object. Remember, from the Radio Group's definition, this value represents the task flow's From Outcome. If the From Outcome is valid, i.e. a radio selection was clicked, we create a `javax.faces.application.NavigationHandler` by calling `getNavigationHandler()` on the JSF `Application` and call its `handleNavigation()` method to handle the navigation. Navigation is handled by specifying the From Outcome string (the `fromOutcome` variable in the code) retrieved from the Radio Group.

To test the recipe right-click on the **navigationHandler** task flow in the **Application Navigator** and select **Run** or **Debug** from the context menu.

There's more...

When a matching control flow case is found, navigation is handled by the ADF Controller layer through the ADF Controller `NavigationHandler`, otherwise navigation handling is delegated to the JSF NavigationHandler. Note that in this case, not all ADF Controller functionality is guaranteed to work as designed.

For more information about how control flow rules are evaluated by the ADF Controller layer, refer to section *What Happens at Runtime: Evaluating Control Flow Rules* in the *Fusion Developer's Guide for Oracle Application Framework*.

See also

Using an Application Module function to initialize a page, chapter 6.

Using a task flow Initializer to initialize a task flow, chapter 6.

Retrieving the task flow definition programmatically using MetadataService

Task flow definition in JDeveloper is done usually through the declarative support provided by the IDE. This includes defining the task flow activities and their relevant control flow cases by drag and dropping task flow components from the **Component Palette** to the **Diagram** tab and adjusting their properties through the **Property Inspector**, defining managed beans in the **Overview** tab, and so on. JDeveloper saves the task flow definition metadata in an XML document, which is accessible in JDeveloper anytime you click on the **Source** tab. The task flow definition metadata is available programmatically at runtime through the `oracle.adf.controller.metadata.MetadataService` object by calling `getTaskFlowDefinition()`.

In this recipe we will show how to get the task flow definition metadata by implementing the following use case: for each task flow in our ADF Application provide a generic technique for logging the task flow input parameters upon task flow entry and the task flow return values upon task flow exit.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. New functionality will be added to the ViewController project that is part of the shared components workspace.

Moreover, this recipe enhances the `taskflowInitializer` task flow developed in the *Using a task flow Initializer to initialize a task flow* recipe in this chapter. Note the additional prerequisites stated for those recipes, i.e. the usage of the `HRComponents` and `MainApplication` workspaces and the database connection to the `HR` schema.

How to do it...

1. Open the shared components workspace and create a new Java class called `TaskFlowBaseBean`. Add the following methods to it:

```
public void initialize() {
  // get task flow parameters
  Map<String, TaskFlowInputParameter> taskFlowParameters =
                              getTaskFlowParameters();
  // log parameters
  logParameters(taskFlowParameters);
}
```

```
public void finalize() {
   // get task flow return values
   Map<String, NamedParameter> taskFlowReturnValues =
                                 getReturnValues();
   // log return values
   logParameters(taskFlowReturnValues);
}

protected TaskFlowId getTaskFlowId() {
   // get task flow context from the current view port
   TaskFlowContext taskFlowContext =
      ControllerContext.getInstance().getCurrentViewPort()
                                 .getTaskFlowContext();
   // return the task flow id
   return taskFlowContext.getTaskFlowId();
}

protected TaskFlowDefinition getTaskFlowDefinition() {
   // use MetadataService to return the task flow
   // definition based on the task flow id
   return MetadataService.getInstance()
            .getTaskFlowDefinition(getTaskFlowId());
}

protected Map<String, TaskFlowInputParameter>
                     getTaskFlowParameters() {
   // get task flow definition
   TaskFlowDefinition taskFlowDefinition =
                      getTaskFlowDefinition();
   // return the task flow input parameters
   return taskFlowDefinition.getInputParameters();
}

protected Map<String, NamedParameter> getReturnValues() {
   // get task flow definition
   TaskFlowDefinition taskFlowDefinition =
                    getTaskFlowDefinition();
   // return the task flow return values
   return taskFlowDefinition.getReturnValues();
}
```

```
public void logParameters(Map taskFlowParameters) {
  // implement parameter logging here
}
```

2. Rebuild and redeploy the shared components ADF Library JAR.
3. Open the `MainApplication` workspace and add to its ViewController project the shared components ADF Library JAR deployed in the previous step.
4. Load the `InitializerBean` managed bean implementation class `com.packt.jdeveloper.cookbook.hr.main.view.beans.Initializer` into the Java editor and change it so that it extends the `TaskFlowBaseBean` class as it shown below:

   ```
   public class Initializer extends TaskFlowBaseBean
   ```

5. Also, update its `initialize()` method by adding a call to `super.initialize()` and add the following `finalize()` method:

   ```
   public void finalize() {
     // allow base class processing
     super.finalize();
   }
   ```

6. Finally add a Finalizer to the `taskflowInitializer` task flow using the following EL expression:

   ```
   #{pageFlowScope.InitializerBean.finalize}
   ```

How it works...

In step 1 we created a class called `TaskFlowBaseBean` that we can use throughout our ADF application as the base class from which beans providing task flow initializer and finalizer methods can be derived (as we did in step 3 in this recipe). This class consists of initializer and finalizer methods that retrieve and log the task flow input parameters and return values respectively. These methods are implemented by `initialize()` and `finalize()` and they are publicly accessible, which means that they can be directly used from within JDeveloper when defining task flow initializers and/or finalizers. This is the case if you don't want to provide any specific implementations of the task flow initializer and/or finalizer method. The `initialize()` method calls the helper `getTaskFlowParameters()` to retrieve the input task flow parameters and then calls `logParameters()` to log these parameters. Similarly `finalize()` calls `getReturnValues()` to retrieve the returned values and `logParameters()` to log them. The `getTaskFlowParameters()` and `getReturnValues()` helper methods, rely on getting the task flow definition `oracle.adf.controller.metadata.model.TaskFlowDefinition` object and calling `getInputParameters()` and `getReturnValues()` on it respectively.

The task flow definition is returned by the helper `getTaskFlowDefinition()`, which retrieves it by calling the `oracle.adf.controller.metadata.MetadataService` method `getTaskFlowDefinition()`. This method accepts an `oracle.adf.controller.TaskFlowId`, indicating the task flow identifier for which we are inquiring the task flow definition. We retrieve the current task flow identifier by calling the helper `getTaskFlowId()`. In `getTaskFlowId()` we retrieve the current task flow from the task flow context obtained from the current view port, as it is shown in the following lines of code:

```
// get task flow context from the current view port
TaskFlowContext taskFlowContext =
      ControllerContext.getInstance()
               .getCurrentViewPort().getTaskFlowContext();
// return the task flow id
return taskFlowContext.getTaskFlowId();
```

In step 2 we re-deployed the shared components workspace as an ADF Library JAR. Then, in step 3, we added it to the `MainApplication` ViewController project. One way to do this is through the **Resource Palette**.

To demonstrate the usage of the `TaskFlowBaseBean` class, we updated the `InitializerBean` managed bean class `Initializer` that was developed in an earlier recipe, so that it is derived from it (step 4). Then (in step 5), we updated the `Initializer` class `initialize()` method to call `TaskFlowBaseBean`'s `initialize()` to do the base class processing, i.e. to log any input parameters.

In steps 5 and 6, to completed the recipe, we added a task flow finalizer, which simply calls the base class' `super.finalize()` to log the returned task flow parameters.

There's more...

The implementation of `logParameters()`, not included in the book's source, is left as an exercise. This method should basically iterate over the task flow parameters and for each one obtain its value expression by calling the `oracle.adf.controller.metadata.model.Parameter.getValueExpression()` method. The parameter's value expression can be evaluated by calling the `javax.faces.application.Application.evaluateExpressionGet()` method.

Also, note that task flow metadata is loaded from ADF Controller metadata resources using the following search rules: first resources named `META-INF/adfc-config.xml` in the Classpath are loaded and then the existence of the web application configuration resource named `/WEB-INF/adfc-config.xml` is checked and loaded it if it exists. Once these resources are loaded they may reference other metadata objects that reside in other resources. These ADF Controller metadata resources are used to construct a model for the unbounded task flow. Metadata for bounded task flows is loaded on demand.

For a complete reference to all the methods available by the `MetaDataService` and `TaskFlowDefinition` classes consult the *Oracle Fusion Middleware Documentation Library 11g Release 2 Java API Reference for Oracle ADF Controller*. It can be found in this URL: `http://download.oracle.com/docs/cd/E16162_01/apirefs.1112/e17480/toc.htm`

See also

Breaking up the application in multiple workspaces, chapter 1.

Using a task flow Initializer to initialize a task flow, chapter 6.

Creating a train

Wizard-like user interfaces can be created in ADF using task flows created as trains and ADF Faces user interface components such as the `af:train` (Train) and `af:trainButtonBar` (Train Button Bar) components. Using such an interface you are presented with individual steps, called train stops, in a multi-step process, each step being a task flow activity or a combination of activities. Options exist that allow for the configuration of the train stops, controlling the sequential execution of the train stops, whether a train stop can be skipped or others. Furthermore, a train stop can combine other task flow activities, Method Calls for instance. Also other task flows themselves can be added as train stops in the train (as Task Flow Call activities).

In this recipe we will go over the creation of a simple train consisting of View, Method Call and Task Flow Call activities.

Getting ready

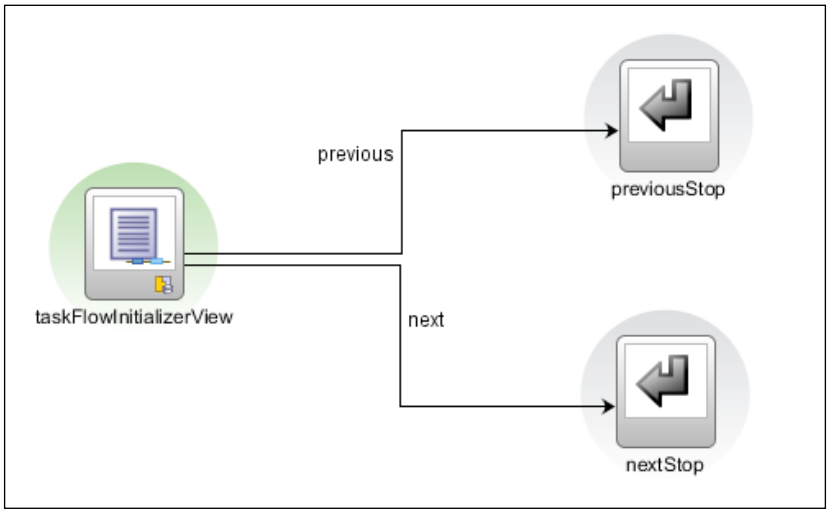
You will need a skeleton Fusion Web Application (ADF) workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

To demonstrate a method call activity as part of the train stop, the recipe uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2. Moreover, to demonstrate a task flow call as a train stop the recipe uses the `taskflowInitializer` task flow created in the *Using a task flow Initializer to initialize task flow* recipe in this chapter.

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

How to do it...

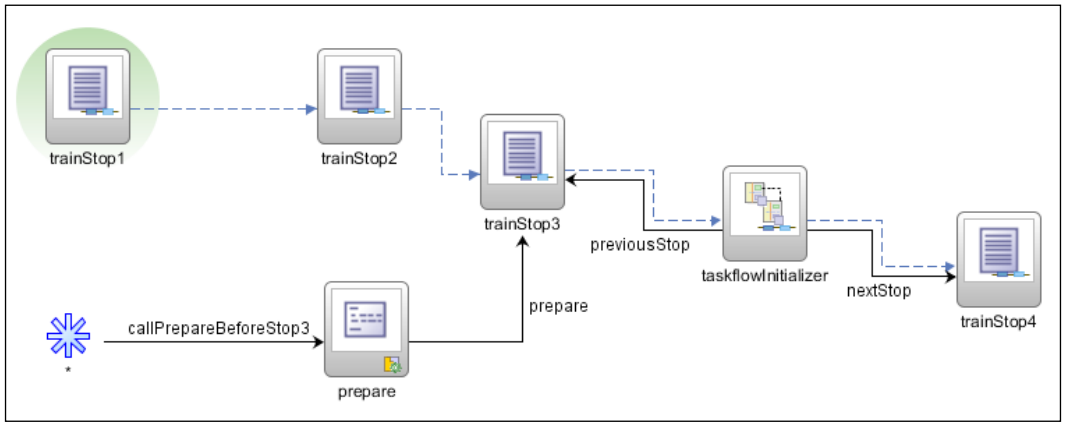
1. Create a bounded task flow called `trainTaskFlow`. Ensure that the **Create Train** checkbox in the **Create Task Flow** dialog is selected. We will not be using page fragments, so ensure that the **Create with Page Fragments** checkbox is not selected.
2. From the **Component Palette** drop four **View** activities on the task flow. Call the view activities `trainStop1`, `trainStop2`, `trainStop3`, and `trainStop4`.
3. Expand the **Data Controls** node in the **Application Navigator** and the **HrComponentsAppModuleDataControl**. Locate and drop on the task flow the **prepare()** method.
4. Create a Control Flow Case from the **prepare** Method Call to the **trainStop3** View activity.
5. Drop a **Wildcard Control Flow Rule** From the **Component Palette** to the task flow and create a Control Flow Case called **callPrepareBeforeStop3** from the Wildcard Control to the **prepare** Method Call.
6. Select the **trainStop3** View activity and in the **Property Inspector** enter **callPrepareBeforeStop3** for its **Outcome** attribute.
7. Locate the **taskflowInitializer** task flow in the **Application Navigator** and double-click on it to open it. From the **Component Palette**, drop two **Task Flow Return** components to it, called **previousStop** and **nextStop**.
8. From the **Component Palette** add two **Control Flow Case**s and connect them from the **taskFlowInitializerView** View activity to the **previousStop** and **nextStop** Task Flow Return activities. Call them **previous** and **next** respectively. The modified **taskflowInitializer** task flow should look like this:



Return to the **trainTaskFlow** task flow. In the **Application Navigator** locate the **taskflowInitializer** task flow and drop it in the **trainTaskFlow** task flow.

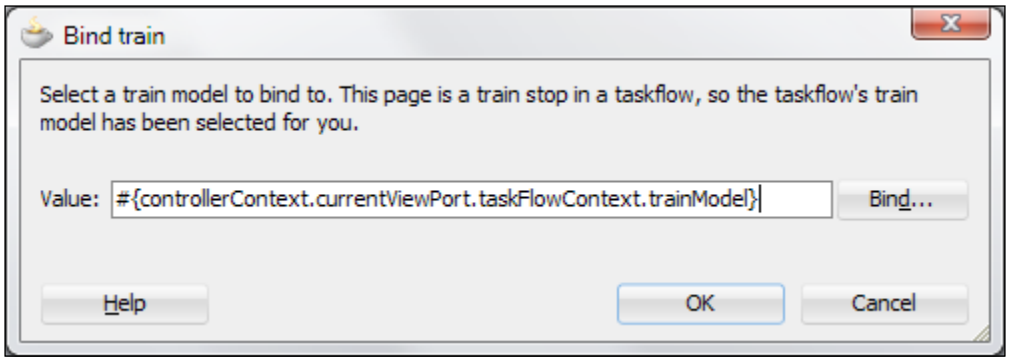
Right-click on the **trainStop4** View activity and select **Train > Move Backward** from the context menu.

Create two Control Flow Cases called **previousStop** and **nextStop** from the **taskflowInitializer** Task Flow Call activity to the **trainStop3** and **trainStop4** View activities. This complete **taskflowInitializer** task flow should look like this:



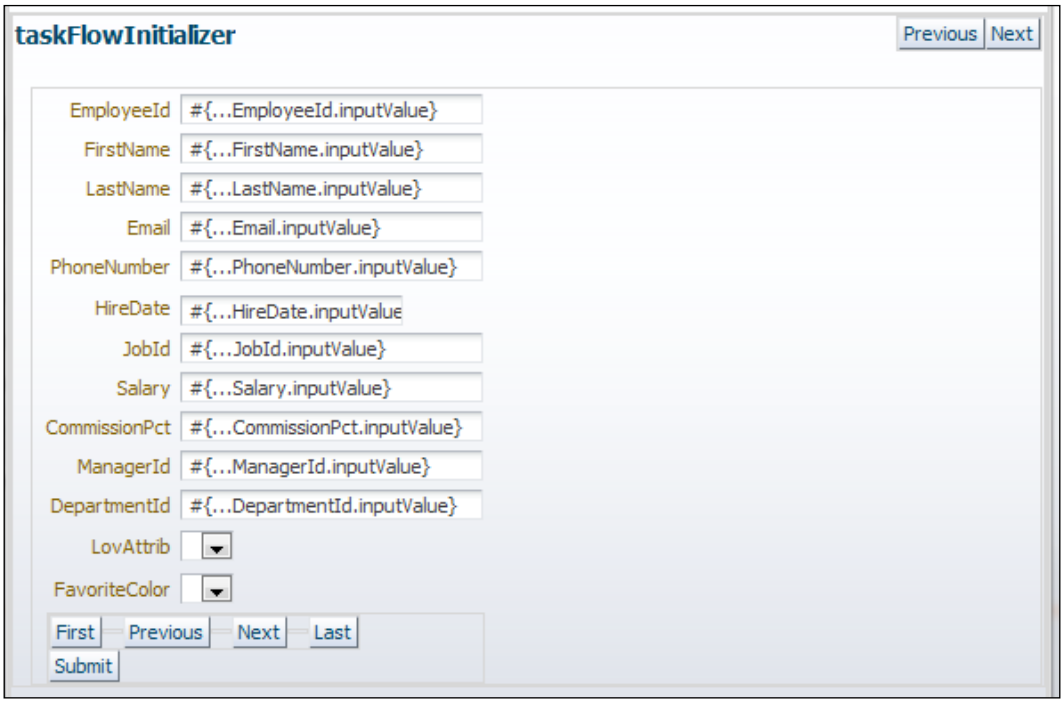
Now, double-click on each of the **trainStop1**, **trainStop2**, **trainStop3** and **trainStop4** View activities in the **taskflowInitializer** task flow to create the JSF pages. In the **Create JSF** page dialog select **JSP XML** for the **Document Type**.

For each of the pages created select a **Train** component from the **ADF Faces Component Palette** and drop them on the pages. On the **Bind train** dialog that is displayed accept the default binding and click **OK**.



Each page should look similar to one shown below:

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 this:



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 is selected. Then, in step 2 we dropped 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 was wired to a Wildcard Control Flow Rule called `callPrepareBeforeStop3` (in step 5) and in order to ensure that the `prepare` Method Call activity and the `trainStop3` View activity were combined in a single train step we 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 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 were 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 did 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 briefly explained below:

- Outcome – It is 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 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 eventually create different 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.*

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using a task flow Initializer to initialize task flow, chapter 6.

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:popup 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 popup reuse
- Exporting data to a file
- Uploading an image to the server

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 by using declarative partial page rendering allows you to develop complex pages using a declarative process by most part. 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, which implies that changes done to the view criteria are automatically reflected by the `af:query` component with no additional changes needed. This fact, along with the declarative support in JDeveloper for displaying the query results in a table (or a tree table) component, makes constructing a search form a straightforward task.

In this recipe we will go over the creation of a query search form with the associated results displayed in a table component.

Getting ready

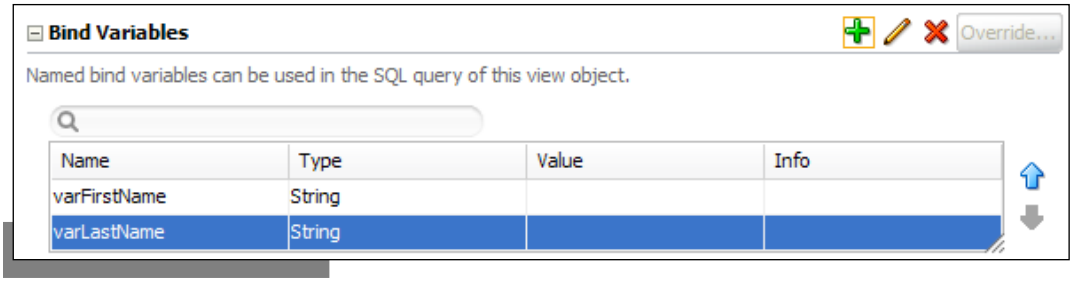
You will need to create a skeleton Fusion Web Application (ADF) workspace before you proceed with this recipe. For this we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

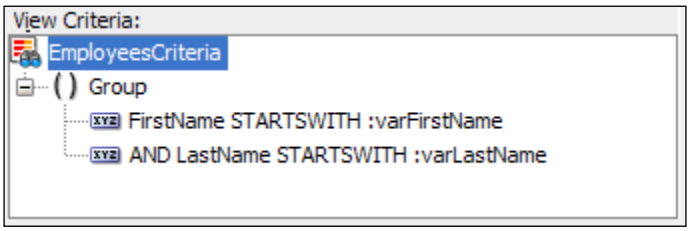
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

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 bind variables. Also in the **Control Hints** tab of the **Bind Variable** dialog ensure that the **Display Hint** is set to **Hide**.

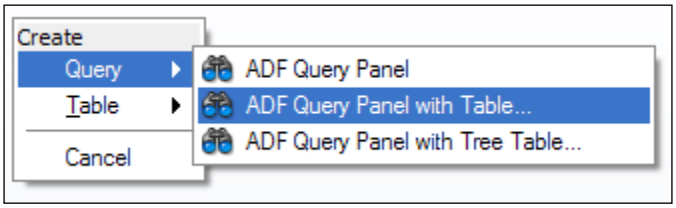


3. While in the **Query** section, add named View Criteria for the **FirstName** and **LastName** attributes using the bind variables **varFirstName** and **varLastName** respectively. For both criteria items ensure that the **Ignore Case** and **Ignore Null Values** checkboxes are checked and that the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

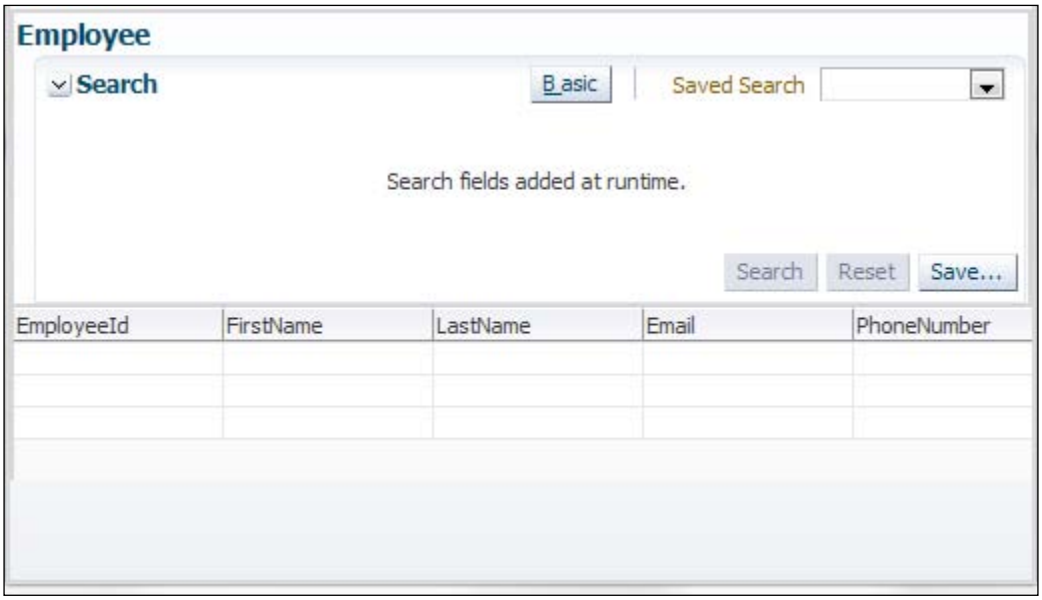


4. Rebuild and redeploy the `HRComponents` workspace as an ADF Library JAR.
5. Open the `MainApplication` workspace and ensure that the `HRComponents` and shared components ADF Library JARs are added to the ViewController project.
6. Create a bounded task flow called `queryTaskFlow` and add a View Activity 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 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...**.



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, the page should look like this:



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, enabling sorting or filtering. Adjust the table's width by specifying a width in pixels in the **Style** section of the **Table Property Inspector**.

How it works...

In steps 1 through 3 we updated the `Employees` View object, which is part of the `HRComponents` workspace, by adding named view criteria to it. Based on these criteria we will subsequently (in steps 8 and 9) associate an `af:query` component to create the search page. Note that the view criteria are comprised of two criteria items, one for the employee's first name and another for the employee's last name. We have based the criteria items on corresponding bind variables created in step 2. Also note the view criteria item settings that were used in step 3. For both the employee first name and last name criteria items 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 not required. Finally note that we based both of the criteria items on the **Starts with** operation and that the `AND` conjunction was used for the criteria items.

In steps 4 and 5 we redeployed the `HRComponents` workspace to an ADF Library JAR, which we then added to the `MainApplication` ViewController project. The `HRComponents` library has dependencies to the shared components workspace, so we make sure that the shared components 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 JSPX page for the View Activity.

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

```
<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 does much more in the background. It creates the corresponding `searchRegion` and `iterator` executables along with the `tree` binding used by the `af:table` component and the necessary glue code to associate the `searchRegion` executable and `tree` binding to the `iterator`. It also associates the the `af:query` component with the specific component that will be used to display the search results. This is done by specifying the result component's identifier (`resId1` in the sample code above) 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, 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.*

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using an af:popup component to edit a table row

An `af:popup` component when utilized in conjunction with an `af:dialog` can be used to display and edit data in the page. The popup 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 *Using an af:query component to construct a search page* recipe in this chapter, to allow for the editing of a table row. The use case that we will demonstrate is to raise an edit form inside a popup dialog by double-clicking on the table row. The changes done to the data inside the dialog is reflected back on 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.

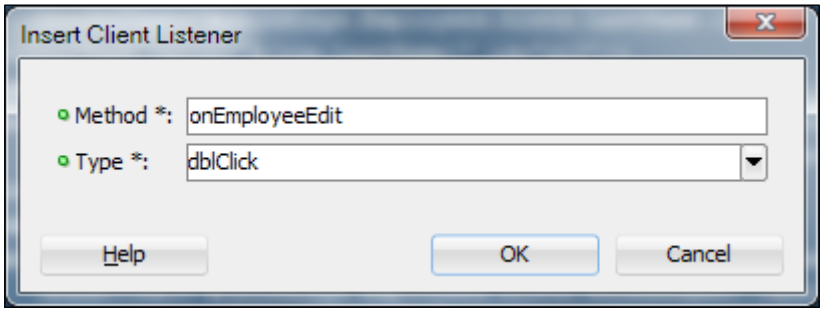
How to do it...

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 and 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:

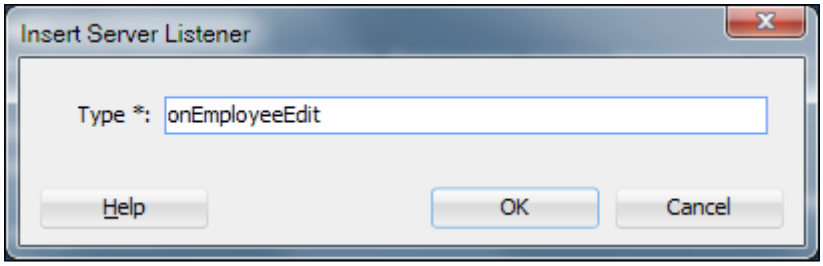
```
public void onEmployeeEdit(ClientEvent clientEvent) {
    FacesContext facesContext =
         FacesContext.getCurrentInstance();
    ExtendedRenderKitService service =
         Service.getRenderKitService(facesContext,
                  ExtendedRenderKitService.class);
    service.addScript(facesContext,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

4. Locate the **queryView.jspx** JSF page in the **Application Navigator** and double-click on it to open it.

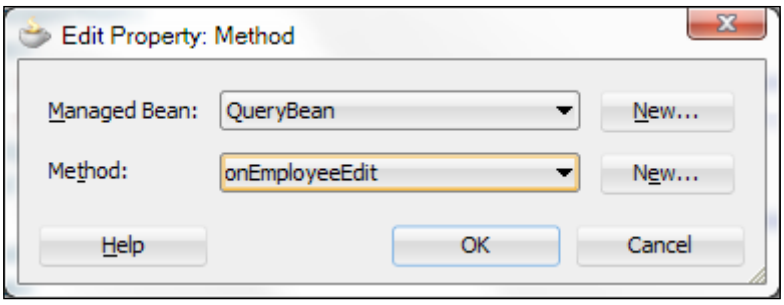
5. Locate and select the **af:table** component in the **Structure** window. Right-click on it and select **Insert Inside af:table** > **ADF Faces** > **Client Listener**.
6. In the **Insert Client Listener** dialog, enter **onEmployeeEdit** for the **Method** and select **dblClick** for the **Type**.



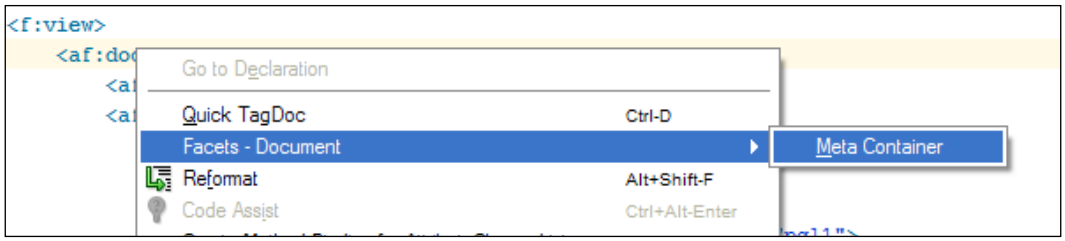
7. Right-click on the af:table component in the Structure window and this time select **Insert Inside af:table** > **ADF Faces** > **Server Listener**. For the **Type** field in the **Insert Server Listener** dialog type **onEmployeeEdit**.



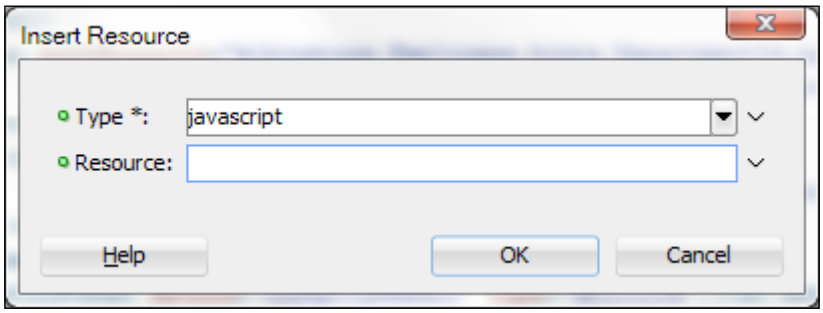
8. Use the **Property Inspector** to set the `af:serverListener` **Method** property to onEmployeeEdit method of the QueryBean.



9. With the af:document selected in the page, right-click and select **Faces – Document** > **Meta Container**.



10. Locate the **f:facet – metaContainer** in the **Structure** window and right-click on it. From the context menu select **Insert Inside f:facet – metaContainer** > **ADF Faces...**. Then select **Resource** from the **Insert ADF Faces Item** dialog.
11. In the **Insert Resource** dialog, specify **javascript** for the **Type** and click **OK**.



12. 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();
}
```

13. Locate a **Popup** component in the **Component Palette** and drop it in the page inside the **af:form** tag.
14. Right-click on the **af:popup** component in the **Structure** window and select **Insert Inside af:popup** > **Dialog** from the context menu.
15. Locate the **Employees** collection under the **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**.

16. Using the **Property Inspector** for the **af:popup** component, change the **Id** property to **editEmployee** and the **ContentDelivery** to **lazyUncached**. Also add the results table identifier in the **PartialTriggers** attribute. You can do this using the **Property Menu** next to the **PartialTriggers** attribute.
17. Finally, adjust the results table **PartialTriggers** by adding the **af:dialog** identifier to it.

How it works...

In steps 1 and 2 we 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. We proceeded in step 3 and added a method to the managed bean called `onEmployeeEdit()`. This method is used as an event response 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` popup is added to the page in step 13. The popup is shown programmatically by infusing dynamic JavaScript code to the page using the `addScript()` method implemented by the `ExtendedRenderKitService` interface. The JavaScript code that is added is specified as an argument to the `adScript()` method. In this case is the following:

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

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

The popup is raised by double-clicking on a table row. In order to accomplish this behavior a combination of an `af:clientListener` and an `af:serverListener` tags is used. We added 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 was added directly to the page in steps 9 through 12. The JavaScript `onEmployeeEdit()` method is shown below as well.

```
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 add the `af:serverListener` to the `af:table`, we identify the serverListener of type `onEmployeeEdit` and we indicate that the backing bean `QueryBean.onEmployeeEdit` method will be executed upon its activation. This is the method implemented in step 3 that programmatically raises the popup.

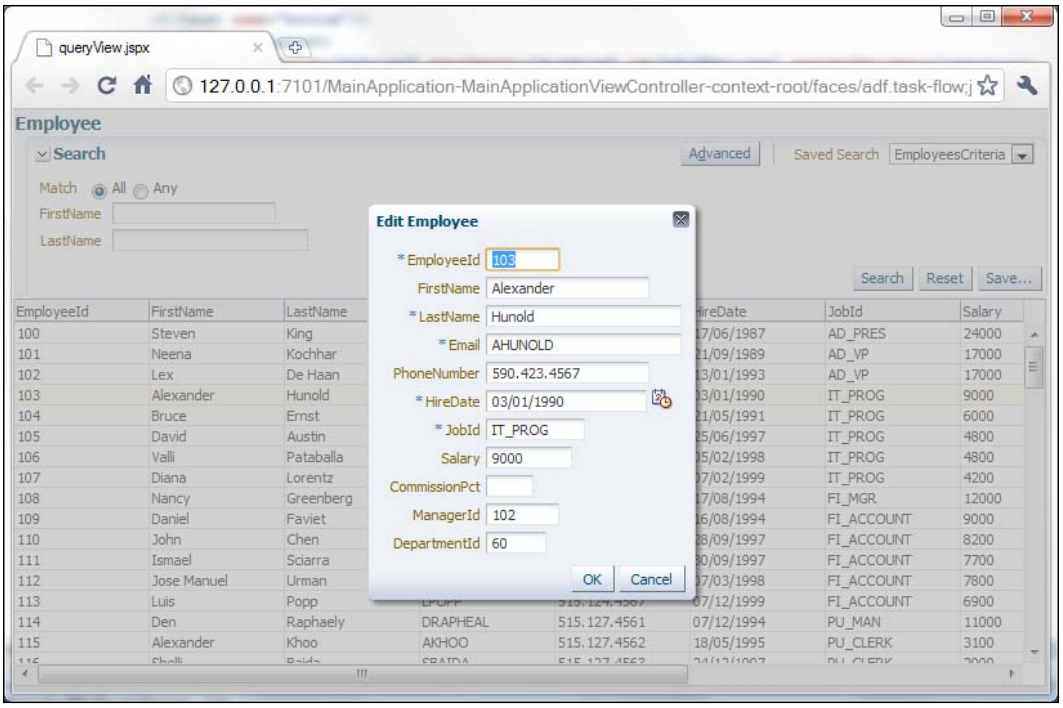
We mentioned earlier that the JavaScript code for the `af:clientListener onEmployeeEdit` method was added in steps 9 through 12. JavaScript was added directly on the page by first adding a `metaContainer` facet to the `af:document` and then using an `af:resource` component of type `javascript`. The actual page code looks like this:

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

The popup is added to the page in steps 13 through 15 using a combination of the `af:popup` and `af:dialog` components. In step 15, we dropped the `Employees` collection from the Data Controls right on the `af:dialog` as an editable form. Since the same collection is bound to the page's table, we will be editing the same data.

Finally, note the adjustments that we made to the popup and table components in steps 16 and 17. First we changed the popup identifier to `editEmployee`. This is necessary since we are programmatically raising the specific popup in step 3 by infusing the specific JavaScript code, i.e. `AdfPage.PAGE.findComponent('editEmployee').show()`. Then we set the popup's `contentDelivery` attribute to `lazyUncached`. The popup content delivery indicates how the popup content is delivered to the client. The `lazyUncached` content delivery setting is used because the data delivered to the popup 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 the popup to reflect the data of the table row that is double-clicked, so we added the table's identifier as a partial trigger to the popup. We also want the changes done to the data in the popup, to be reflected back on 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. You can save the changes to the employee, by clicking **OK** on the **Edit Employee** dialog. If you click **Cancel**, the changes are dismissed. The employee's new data are reflected back on the table.



There's more...

Note, that the page does not implement a commit or rollback functionality, so the changes done to the table's data cannot be committed to the database. To rollback the changes for now, just refresh the page in 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 of the records in the entity cache that were changed. 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 the results table. We will go over how to accomplish this in recipe *Using a custom af:query operation listener to clear both the query criteria and results* in chapter 8.

Moreover, note that this recipe shows how to launch a popup component programmatically using the `ExtendedRenderKitService` class by infusing dynamic JavaScript into the page. It is this infused JavaScript code that actually shows the popup. Another approach to programmatically launching a popup, is to bind the `af:popup` component to a backing bean as an `oracle.adf.view.rich.component.rich.RichPopup` object and use its `show()` method to show the popup. For more information about this technique, take a look at section *Programmatically invoking a Popup* in the *Web User Interface Developer's Guide for Oracle Application Development Framework*.

See also

Using an af:query component to construct a search page, chapter 7.

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 children 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 `selectionListener` to be able 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

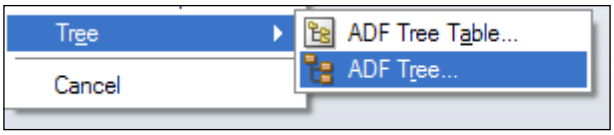
You will need to create a skeleton Fusion Web Application (ADF) workspace before you proceed with this recipe. For this we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

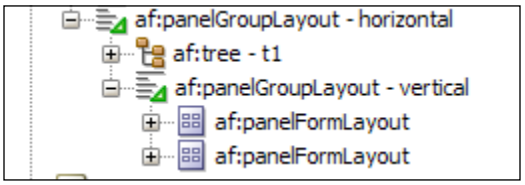
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

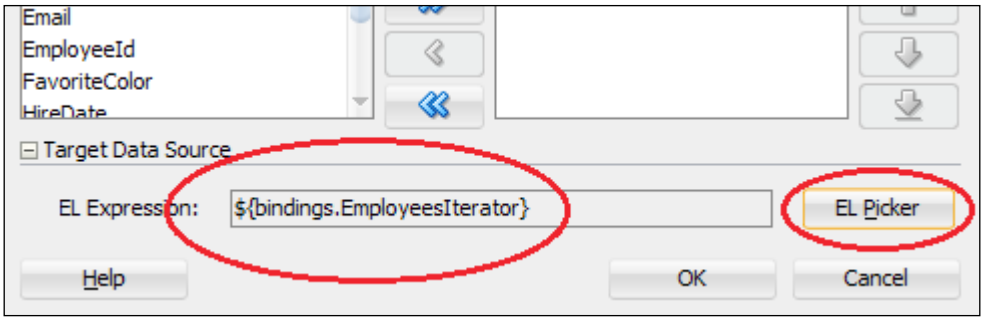
1. Ensure that the `HRComponents` and the shared components ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `treeView.jspx`. Use any of the predefined quick start layouts.
3. Expand the **Data Controls**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



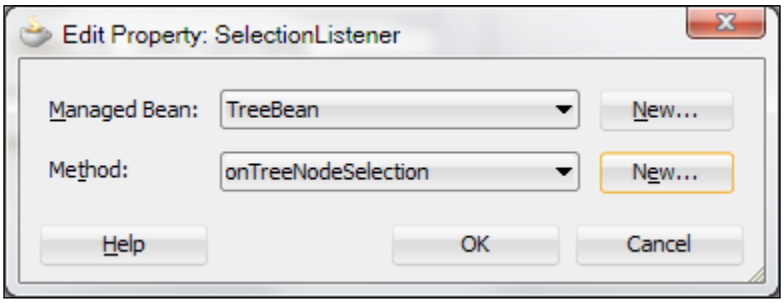
5. In the **Tree Level Rules** section of the **Edit Tree Binding** dialog click on the **Add Rule** button (the green plus sign icon) and add the **Employees** collection. Also adjust the attributes in the **Display Attributes** list so that the **DepartmentName** is listed for the **Departments** rule 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 **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 above 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 above for the `Departments` and `Employees` collections under 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 like this:



10. With the **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.



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



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

```
// invoke defaule 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 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);
```

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 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 shared components workspace, so we make sure that the shared components 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 add the Tree component to the page. The tree is comprised of two nodes or level rules: the parent node represents the departments and it is setup by drag and dropping the `Departments` collection of the `HrComponentsAppModuleDataControl` data control onto the page as an ADF Tree component. The children nodes represent the department employees and are setup 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 tree 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 drop the `Departments` and `Employees` collections on the page as editable forms (`af:panelFormLayout` components) and rearrange the page in such a way that the tree will be displayed on the left part of the page, while the department or employee information will be displayed on the right side. We also bind 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 is steps 11 through 13, we created a custom selection listener for the tree component so that we be 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 by invoking the expression `#{bindings.Departments.treeModel.makeCurrent}`. In order to do this, we used a helper method called `invokeMethodExpression()`. Then, we obtained 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 changed the visible property of the department and employee `af:panelFormLayout` components depending on the type of the selected node. We did this by calling `setVisible()` on the bound department and employee `af:panelFormLayout` components.

There's more...

Note that when adding an `af:tree` component on 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.

Overriding remove() to delete associated children entities, chapter 2.

Using an af:selectManyShuttle component

The `af:selectManyShuttle` ADF Faces component is a databound model-driven component that can be used to select many items from a list of available items. 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 retrieve and process programmatically the selected items.

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

Getting ready

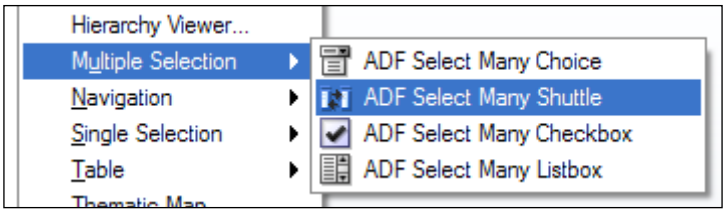
You will need to create a skeleton Fusion Web Application (ADF) workspace before you proceed with this recipe. For this we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

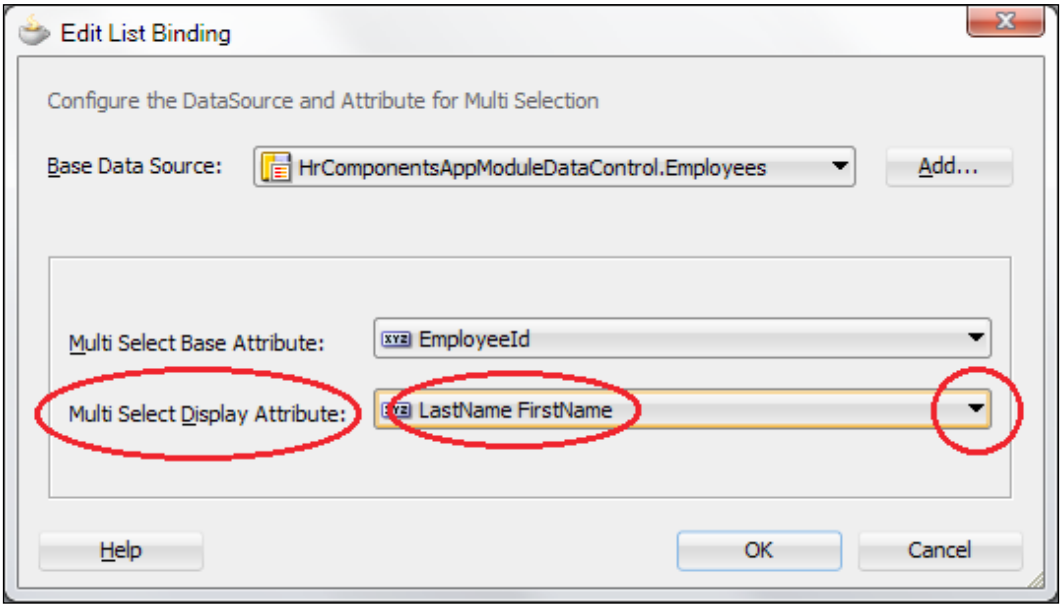
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

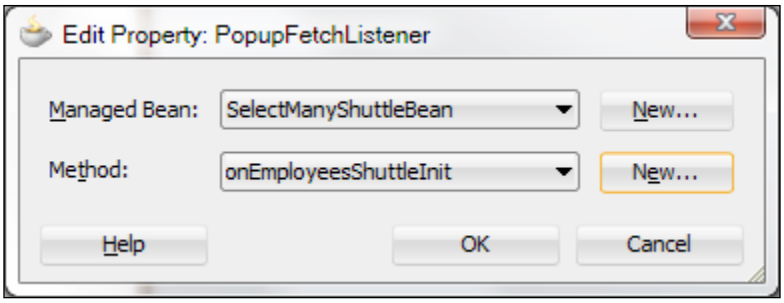
1. Ensure that the `HRComponents` and the shared components ADF Library JARs are added to the ViewController project of your workspace.
2. Using the **Create JSF Page** wizard, create a **JSP XML** page called `selectManyShuttleView.jspx`. Use any of the predefined quick start layouts.
3. Using the **Component Palette**, add a **Popup** component (`af:popup`) to the page. Add also a **Dialog** component (`af:dialog`) inside the popup.
4. Expand the **Data Controls** section in the **Application Navigator** and locate the **Employees** collection under the **HrComponentsAppModuleDataControl** data control. Drag it 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 **LastName** and **FirstName** attributes.



7. Select the **af:popup** in the **Structure** window. Using the **Property Menu** next to the **PopupFetchListener** attribute in the **Property Inspector** select **Edit...** to add a popup 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**, change also the **ContentDelivery** attribute to **lazyUncached**.



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

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

9. Select the **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 Popup Behavior component setup its `PopupId` attribute to point to the popup created above.

How it works...

Since we will be importing Business Components from the `HRComponents` workspace, in step 1 we ensure 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 shared components workspace, so we make sure that the shared components ADF Library JAR is also added to the project.

In steps 2 and 3 we created a new JSPX page called `selectManyShuttleView.jspx` and added a popup to it with a dialog component in it. We will display this popup 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 drag and dropping the `Employees` collection available under the `HrComponentsAppModuleDataControl` data control in the **Data Controls** section of the **Application Navigator**. The `HrComponentsAppModuleDataControl` data control 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 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`. This attribute indicates that 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 it shown below:

```
<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>
```

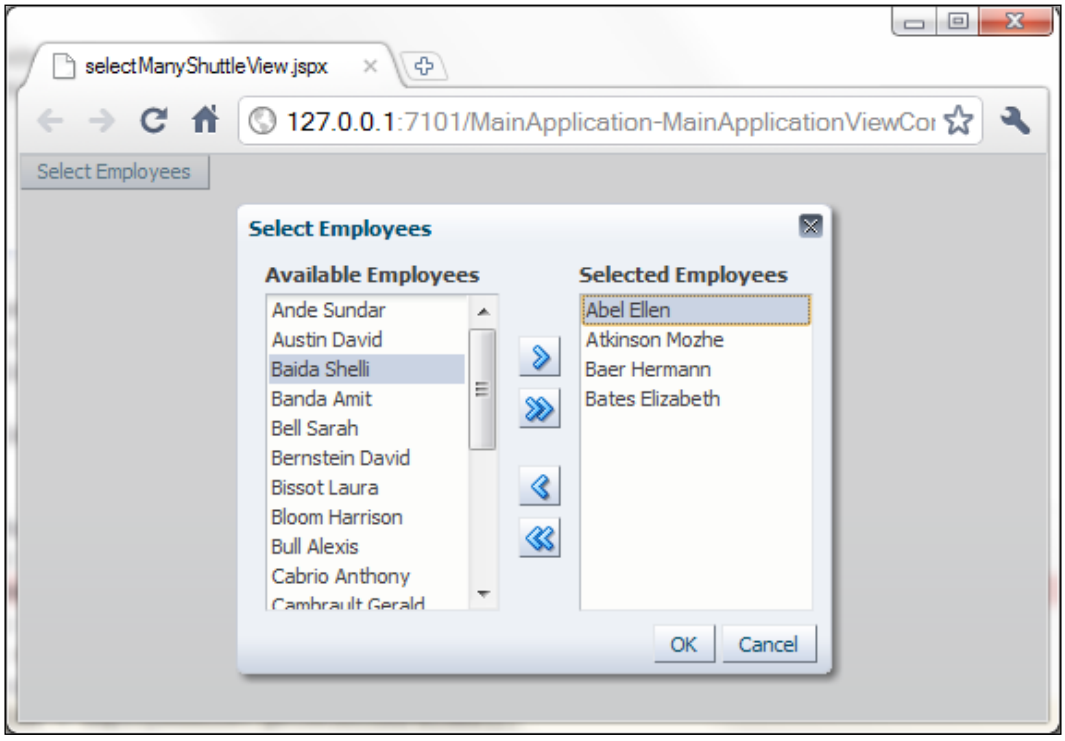
In steps 7 and 8 we had to device a way to initialize the shuttle's selections, each time before the popup is shown. We did this by adding a **PopupFetchListener** to the popup. A PopupFetchListener indicates a method that is executed when a popup fetch event is invoked during content delivery. For the listener method to be executed, the popup content delivery must be set to `lazyUnchached` or `lazy`. We did set the popup 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 have introduced the `ADFUtils` helper class in the *Using ADFUtils/JSFUtils* recipe in chapter 1. 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 will be empty once the popup is displayed.

To handle the list selections we added in steps 9 and 10 a **DialogListener** to the dialog. A DialogListener is a method that can be used to handle the dialog outcome from a dialog event. In it we first check 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 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 popup.

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 popup with a select many shuttle component displaying a list of available employees to select from.



There's more...

Note that ADF Faces provides an additional ordered 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 section *Using Shuttle Components* in the *Web User Interface Developer's Guide for Oracle Application Development Framework*.

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

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 alternative way to display rows of data (carousel items). As the name indicates, 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

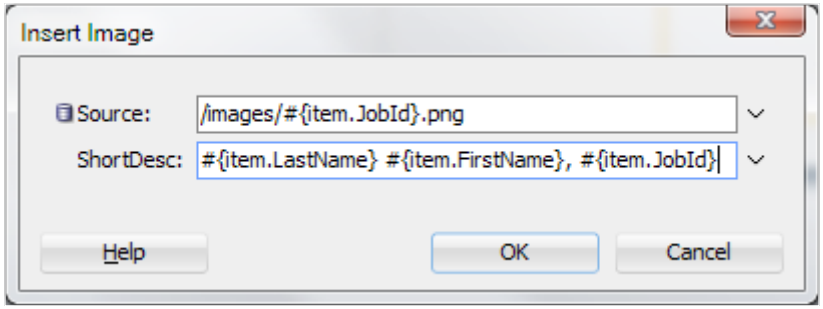
Both the `HRComponents` and `MainApplication` workspaces require database connections to the `HR` schema.

How to do it...

1. Ensure that the `HRComponents` and the shared components 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 it 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 EL 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**.



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 the `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 ensure 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 shared components workspace, so we make sure that the shared components ADF Library JAR is also added to the project.

In step 2 we created a JSPX page that we will use to demonstrate the `af:carousel` component. In the top part of the page we will add a table bound to the `Departments` collection. In the bottom part of the page we will add 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 is added in steps 3 and 4. The carousel is added in step 5. We simply expand the `HrComponentsAppModuleDataControl` data control in the Data Controls section of the Application Navigator and drop the collections on the page making the applicable selections from the Create menu each time. JDeveloper proceeds 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. This is what the page source looks like:

```
<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 bound to the `treeModel` for the `DepartmentEmployees` collection. The `DepartmentEmployees` tree binding is created when the `DepartmentEmployees` collection is dropped on the page as a carousel. The tree binding is used to iterate over the `DepartmentEmployeesIterator` iterator, 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 is 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 have added the images for each employee job identifier.

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

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 by passing a parameter to the servlet the image to retrieve. 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.

For more information about the ADF Faces Carousel component, take a look at section *Using the ADF Faces Carousel Component* in the *Fusion Developer's Guide for Oracle Application Development Framework*.

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

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 and polling can be disabled by setting the interval to a negative value.

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 the database changes done to the employees.

Getting ready

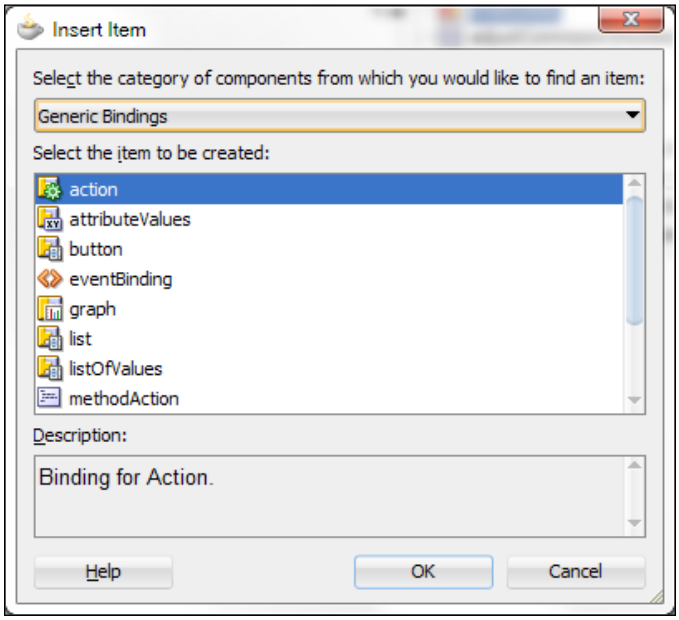
You will need to create a skeleton Fusion Web Application (ADF) workspace before you proceed with this recipe. For this we will use the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

How to do it...

1. Ensure that the `HRComponents` and the shared components 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 it and drop it on the `pollView.jspx` page. 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.



7. In the **Creation Action Binding** dialog select the **Employees** collection under the **HrComponentsAppModuleDataControl** and **Execute** for the **Operation**.
8. With the **Execute** action selected in the **Structure w**indow, 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 added the `HRComponents` ADF Library JAR to our application's ViewController project. We did 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 shared components workspace, so we make sure that the shared components ADF Library JAR is also added to the project.

Then, in step 2, we created a JSPX page called `pollView.jspx` that we will use to demonstrate the `af:poll` component. We will demonstrate its use 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 we adjusted the time interval that 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 used the `ADFUtils findOperation()` helper method to retrieve the `RefreshEmployees` action binding from the bindings container. The `ADFUtils` helper class was introduced in the *Using ADFUtils/JSFUtils* recipe in chapter 1. 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 refresh the employees 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.

Overriding remove() to delete associated children entities, chapter 2.

Using page templates for popup reuse

Back in chapter 1 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 popup dialogs to a page template definition that can 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 popup to the page template that can be utilized in a generic way by all pages based on the template. This will provide a uniform delete behavior for our application. For this, we will enhance the page template that we created in the *Using page templates* recipe in chapter 1.

Getting ready

You will need to have access the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in chapter 1. 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 in the *Using page templates* recipes in chapter 1.

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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

How to do it...

1. Open the shared components 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 popup source should look similar to this:

```
<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.findComponent('generic:"
            + popupId + "').show();");
```

8. Redeploy the shared components 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 shared components and `HRComponents` ADF Library JARs to the ViewController project.
10. Open the `adfc-config` unbounded task flow, go to the **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 shared components 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** check box.
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 expanded the `TemplateDef1` page template definition by adding a popup called `DeleteConfirmation`. We can raise this popup 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 popup should match the name used in the `CommonActions.onConfirmDelete()` method to display the popup. This is what the `CommonActions.onConfirmDelete()` method looks like:

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

The necessary code to display the popup was added in the `ADFUtils.showPopup()` method in step 7. The `ADFUtils` helper class was introduced in the *Using ADFUtils/JSFUtils* recipe in chapter 1. The `ADFUtils.showPopup()` method is shown below:

```
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:popup 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 popup id.

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

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

Notice how in steps 5 we added a **Continue** button to the delete confirmation popup and how in step 6 we explicitly specified the `CommonActions.onContinueDelete()` method as the continue button's action listener. The `CommonActions.onContinueDelete()` method looks like this:

```
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()`. This is the code of `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 below:

```
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 below:

```
public String getPageId() {
FacesContext fctx = FacesContext.getCurrentInstance();
return fctx.getViewRoot().getViewId().substring(
fctx.getViewRoot().getViewId().lastIndexOf("/"));
}
```

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

We continued in step 8 by redeploying the shared components workspace to an ADF Library JAR.

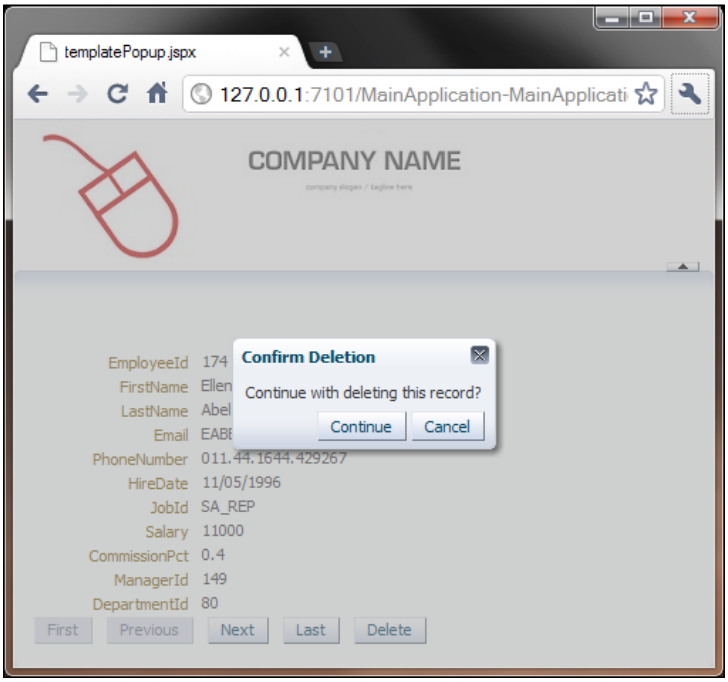
To test the generic template popup, in step 9, we created a Fusion Web Application (ADF) workspace and added the shared components and `HRComponents` ADF Library JARs to its ViewController project.

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

In steps 11 through 14, we created a page called `templatePopup.jspx` based on the `TemplateDef1` template and we dropped on it 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 was set to the same identifier value as in the template definition, i.e. `generic`. This is important for the code in step 7 that loads the popup 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 Delete** popup defined in the page template will be displayed and the `CommonActions` framework will be used to handle the delete action.



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, i.e. `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.

Using page templates, chapter 1.

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Exporting data to a client file

You can easily 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 demonstrate the File Download Action Listener component (`af:fileDownloadActionListener`) by implementing the following use case: we will export all employees into 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

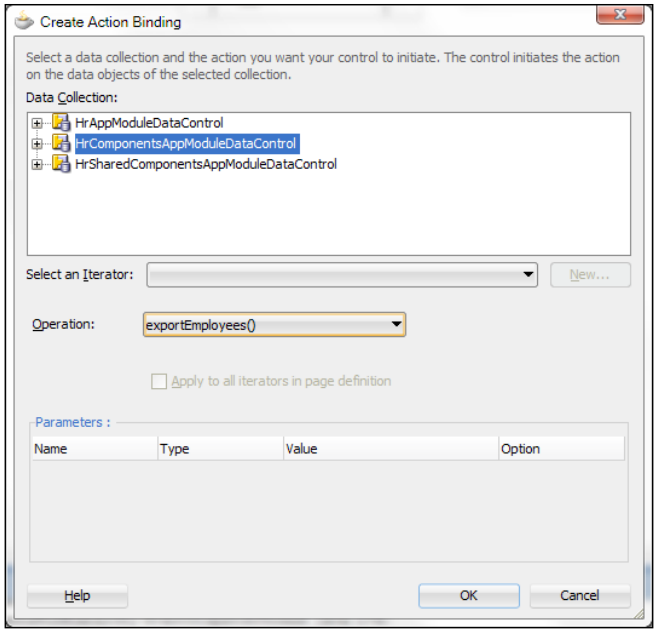
How to do it...

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 Navigator 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 shared components 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**.



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 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 added the `exportEmployees()` method to the Application Module's client interface to make available to the ViewController layer. Then, in step 4, we redeployed the **HRComponents** workspace into an ADF Library JAR.

Steps 5 through 13 were done to 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 added 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 shared components workspace, so we make sure that the shared components ADF Library JAR is also added to the project.

In step 6 we creates a JSPX 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. Note how we initially, in step 7, we 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 bind once more 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. By defining using this methodAction binding, will allows 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 to the download action listener method `exportEmployees()` was added in step 13. This code uses the `ADFUtils` helper class to programmatically execute the `exportEmployees` methodAction binding. This is the 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 the *Using ADFUtils/JSFUtils* recipe in chapter 1.

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, i.e. **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*.

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Uploading an image to the server

In the *Exporting data to a client file* recipe in this chapter, we went over the process of exporting data from the server and downloading it to a file on the client. In this recipe we will do the opposite: we will upload some data stored on the client to the server. The specific use case that we will implement for this recipe is the following: using the `HR` schema we will associate with every employee a picture. First, we will browse for a picture file on the client utilizing the ADF Faces Input File component (`af:inputFile`). Then, we will upload the picture selected to the server, associating it with the current employee displayed on the browser.

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* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

Finally, before proceeding with this recipe, you will need to update the `HR.EMPLOYEES` table so that an additional column, called `PICTURE` of type `BLOB`, is added. You can do this by running the following script:

```
ALTER TABLE HR.EMPLOYEES ADD (PICTURE BLOB);
```

How to do it...

1. Ensure that the `HRComponents` and the shared components 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 `imageUpload.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 it and drop it on the `imageUpload.jspx` page. From the **Create** menu, select **Form** > **ADF Read-only Form...**.
4. In the **Edit Form Fields** dialog, select the form fields and click on the **Include Navigation Controls** checkbox. Click **OK** to continue.
5. Locate an **Input File** component from the **Component Palette** and drop it on the page next to the navigation controls. Using the **Property Inspector** change the **Input File** component's **Label** property to **Picture**. Using the **Property Menu** next to the **Binding** property, add a binding to a newly created managed bean. Call the binding **employeePicture**.
6. Using the **Component Palette**, drop a **Button** component next to the **Input File** component. Change the button's **Text** property to **Upload Picture**. Using the **Property Menu** next to the **ActionListener** property, add an action listener method called **onEmployeePictureUpload** in the same managed bean.
7. Click on the **Bindings** tab to switch to the page bindings view. Ensure that an attribute binding exists for the Picture attribute. If not, Click on the **Create control binding** button (the green plus sign icon) in the **Bindings** list, and from the **Insert Item** dialog select **attributeValue** (under the **Generic Bindings**). In the **Create Attribute Binding** dialog, select **Employees** for the **Data Source**, and **Picture** forthe **Attribute**.
8. Repeat the step above to create a control binding. Click on the **Create control binding** button (the green plus sign icon) in the **Bindings** list, and from the **Insert Item** dialog select **action** (under the **Generic Bindings**). In the **Create Action Binding** dialog, select the **HrComponentsAppModuleDataControl** in the **Data Collection** tree, and **Commit** for the **Operation**.

9. Now, switch to the managed bean created in step 5 and add the following code in the **onEmployeePictureUpload** method:

```
UploadedFile employeePictureFile =
(UploadedFile)this.getEmployeePicture().getValue();
if (employeePictureFile != null) {
String contentType = employeePictureFile.getContentType();
   // check for image file type
   if (contentType != null
     && contentType.startsWith("image")) {
    try {
          // read image and write to a blob
         BufferedInputStream inputStream =
          new BufferedInputStream(
               employeePictureFile.getInputStream());
         byte[] imageBuffer = new byte[100 * 1024];
         int bytesRead;
         BlobDomain blob = new BlobDomain();
         BufferedOutputStream blobStream =
          new BufferedOutputStream(
            blob.getBinaryOutputStream());

         while ((bytesRead = inputStream.read(imageBuffer))
                                                 != -1) {
          blobStream.write(imageBuffer, 0, bytesRead);
         }

         // done with loading the image into a blob
         inputStream.close();
         blobStream.close();

         // set employee picture for current row
         ADFUtils.setBoundAttributeValue("Picture", blob);
         // and commit
         ADFUtils.findOperation("Commit").execute();

         // show success message
         FacesContext context =
          FacesContext.getCurrentInstance();
         FacesMessage message =
               new FacesMessage("Successfully uploaded file "
                 + employeePictureFile.getFilename()+ " ("
                 + employeePictureFile.getLength()
                 + " bytes)");
         context.addMessage(null, message);
```

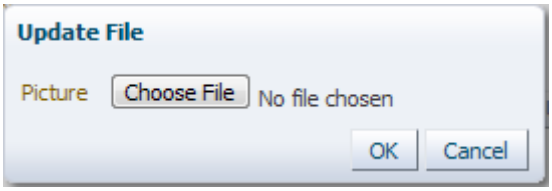
```
        } catch (Exception e) {
            // log exception
        }
    } else {
    // handle errors
        this.getEmployeePicture().setValue(null);
        FacesContext context =
            FacesContext.getCurrentInstance();
        FacesMessage message = null;
        if (employeePictureFile != null)
            message = new FacesMessage(
              "Error loading picture file "
              + employeePictureFile.getFilename());
        else
            message = new FacesMessage(
            "No picture file was specified");
           context.addMessage(null, message);
    }
}
```

How it works...

In step 1 we added the `HRComponents` ADF Library JAR to our application's ViewController project. We did 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 shared components workspace, so we make sure that the shared components ADF Library JAR is also added to the project.

We proceeded with creating a page called `imageUpload.jspx` (in step 2), and adding a read-only form by dropping the `Employees` collection to the page from the **Data Controls** (in step 3). The **HrComponentsAppModuleDataControl** data control was added to the list of the available data controls once we added the `HRComponents` ADF Library JAR to the project. We indicated to include navigation controls to the form (step 4), so that we can navigate the employees.

In step 5, we added an Input File (`af:inputFile`) ADF Faces component to the page to allow for the selection of an image file, which will be associated with the current employee's picture. This component provides a button to allow for the selection of a file. At runtime, when you click on the button, a native operating system file open window is presented that allows you to choose a file. Once a file is selected and uploaded to the server, the Input File component allows you to update the file by presenting an Update File dialog:



Note how in step 5 we created a managed bean binding for the `af:inputFile` component called `employeePicture`. This binding creates an `oracle.adf.view.rich.component.rich.input.RichInputFile` attribute called `employeePicture` on the managed bean that will allow us to manipulate the Input File component in step 9 from within the managed bean code.

In step 6, we added a button to allow us to upload the file selected. The actual file upload is done in the `onEmployeePictureUpload` action listener (in step 9), which is also added in this step.

To be able to save the image data to the `Employees` collection current employee (this is done in the `onEmployeePictureUpload` action listener in step 9), in step 7 we ensured that the `Picture` attribute is bound to the page. This is the attribute that is associated to the `PICTURE BLOB` column of the `EMPLOYEES` table in the `HR` schema.

We also added, in step 8, an action binding for the `Commit` operation. We will implicitly execute this action binding in step 9 to commit the changes once the image has been copied to the `Picture` attribute of the current employee displayed on the page.

In step 9, we implement the code to upload the employee picture to the server. Using the `employeePicture` binding we call `getValue()` to get the `UploadedFile` model. We determine the file contents by calling `getContentType()` on the `UploadedFile`, and we check for a valid image content type. A valid image type starts with `image`, for a JPEG image for instance is `image/jpeg`. For a valid image file, we proceed by reading its contents from the `UploadedFile` model input stream and writing it to an `oracle.jbo.domain.BlobDomain`. Once the image contents are written to the `oracle.jbo.domain.BlobDomain`, we set the `Picture Employees` collection attribute to it. We do this by utilizing the `ADFUtils` helper class `setBoundAttributeValue()` method. The `ADFUtils` helper class was introduced in the *Using ADFUtils/JSFUtils* recipe in chapter 1. Finally, we commit the changes to the current `Employees` row by programmatically executing the `Commit` action binding. Again, we the use `ADFUtils` helper class and the

`findOperation()` method to locate the action binding. This method returns an `oracle.binding.OperationBinding`, which we use to call its `execute()` method to execute the action binding. This results in committing the changes to the database.

There's more...

You can find additional information on the `af:inputFile` ADF Faces component on the *Using File Upload* section in the *Web User Interface Developer's Guide for Oracle Application Development Framework*.

See also

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

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 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
- Using JasperReports

Introduction

Baking (or managed) beans are isolated pieces of Java code referenced by the JSF pages used in a Fusion Web Application (ADF) through Expression Language (EL) and 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 Fusion Web Application (ADF) 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` and `pageFlow` scopes which are ADF specific.
- `adfc-config.xml` – The unbounded task flow definition. 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 – Managed bean definitions are accessed only by the specific task flow.`

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 handy when for instance you want to raise a warning popup message each time you attempt to leave the current record. 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 approach throughout your application dealing with pending uncommitted transaction changes. The `CommonActions` framework was introduced in the *Using a generic View Controller actions framework* recipe in chapter 1.

Getting ready

The functionality implemented by this recipe will be added to the `ADFUtils` helper class introduced in the *Using ADFUtils/JSFUtils* recipe in chapter 1. This class is part of the shared components workspace.

How to do it...

1. Open the shared components 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 Business Components (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 whether 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 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 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.

Using an af:popup to handle pending changes, chapter 8.

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 this: `#{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 go over how you can implement your own custom table selection listener. This may be handy if your application requires any additional processing 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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

Moreover, this recipe enhances the `JSFUtils` helper class introduced in the *Using ADFUtils/JSFUtils* recipe in chapter 1 which is part of the shared components workspace.

How to do it...

1. Open the shared components 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. Then, ensure that you redeploy the shared components workspace to an ADF Library JAR.

```
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 main application workspace and add the shared components and the `HRComponents` ADF Library JARs to the ViewController project.
4. Create a JSP XML page based on any pf 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 `SelectionListener`, 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 above:

```
// 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 common components workspace, where the `JSFUtils` helper class is defined, was redeployed into an ADF Library JAR. Then, in step 3, we added the newly deployed common components ADF Library JAR into the ViewController project of our application. We also added the `HRComponents` ADF Library JAR to the ViewController project, since 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 JSPX page and we dropped in it the `Employees` collection, part of the `HrComponentsAppModule` in the `HRComponents` ADF Library JAR, as an ADF Table (`af:table`) component. 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 by calling `getSource()` on the selection event and then calling `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...

This recipe demonstrated how to implement the table selection by invoking the default selection listener `makeCurrent` programmatically via a JSF method expression invocation. To do the analogous task with Java code instead, 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*.

See also

Using ADFUtils/JSFUtils, chapter 1.

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

Using a custom af:query listener to allow 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 JSF method expression indicated. 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* recipe in chapter 1.

The recipe also uses the shared components and `HRComponents` workspaces, which were created in the *Breaking up the application in multiple workspaces* in chapter 1 and in the *Overriding remove() to delete associated children entities* recipe in chapter 2 respectively.

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

How to do it...

1. Open the main application workspace and ensure that both the shared components 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 shared components 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 JSPX page and we 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`, which 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 as a Data Control 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**, which also allows us to create and configure, if needed, a new managed bean. We simply called our custom query listener `queryListener` and we 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 the specific criterion 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 `QueryEvent` event passed by the ADF framework to the query listener. The `getCriterionList()` method returns a `java.util.List` of `AttributeCriterion`, which we iterate over checking 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 the *Using ADFUtils/JSFUtils* in chapter 1. 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*.

See also

Using ADFUtils/JSFUtils, chapter 1.

Breaking up the application in multiple workspaces, chapter 1.

Overriding remove() to delete associated children entities, chapter 2.

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 yourself the `af:query` component's search functionality. 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 by the `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 shared components and `HRComponents` workspaces, which were created in the *Breaking up the application in multiple workspaces* in chapter 1 and in the *Overriding remove() to delete associated children entities* recipe in chapter 2 respectively.

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

How to do it...

1. Open the shared components workspace and locate the `ExtApplicationModuleImpl.java` custom Application Module extension class. Add the following `resetCriteria()` method 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 shared components 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 main application 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 shared components workspace to an ADF Library JAR.

In step 3, we added a method called `resetEmployees()` to the `HrComponentsAppModule` Application Module implementation class that 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 `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 implementation of the query operation listener where the presence of the `reset` operation binding cannot be guaranteed, use the `QueryModel reset()` method to accomplish the reset of the `af:query` component only. The `reset()` method in this case is called for all system saved searches, which are retrieved by calling `getSystemQueries()` on the `QueryModel`. This is shown in the code snippet below:

```
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.

Overriding remove() to delete associated children entities, chapter 2.

Using a custom af:query listener to allow execution of a custom Application Module operation recipe, chapter 8.

Using a session scope bean to preserve session-wide information

Session-wide 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the shared components and `HRComponents` workspaces, which were created in the *Breaking up the application in multiple workspaces* in chapter 1 and in the *Overriding remove() to delete associated children entities* recipe in chapter 2 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.

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 = :inEmployeeId`.
3. Add a bind variable called `inEmployeeId`. 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 a 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 main application workspace and add both the `HRComponents` and the shared components 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 UserInfoAppModule userInfoAppModule;
private String firstName;
private String lastName;

public SessionInfoBean() {
    userInfoAppModule = (UserInfoAppModule)ADFUtils
        .getApplicationModuleForDataControl(
            "UserInfoAppModuleDataControl");
}

public String getFirstName() {
    if (firstName == null) {
        firstName = userInfoAppModule.getFirstName();
    }
    return firstName;
}

public String getLastName() {
    if (lastName == null) {
        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 we 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`, we 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, i.e. `getLastName()` 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 dependant shared components ADF Library JARs to the main application's ViewController project.

Finally, in steps 9 and 10, we added a session-scoped managed bean, called `SessionInfoBean`, to the main application 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 the *Using ADFUtils/JSFUtils* recipe in chapter 1.

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.

Overriding remove() to delete associated children entities, chapter 2.

Using an af:popup during long running tasks

For long running tasks in your application, a popup 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 we will raise a popup for the duration of the task to alert us about the fact that this operation may take awhile. We will hide the popup 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

How to do it...

1. Open the main application 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 a `metaContainer` facet to the `af:document` tag and an `af:resource` tag inside the `metaContainer` facet. 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 popup 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 popup should look similar to this:

```
<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...

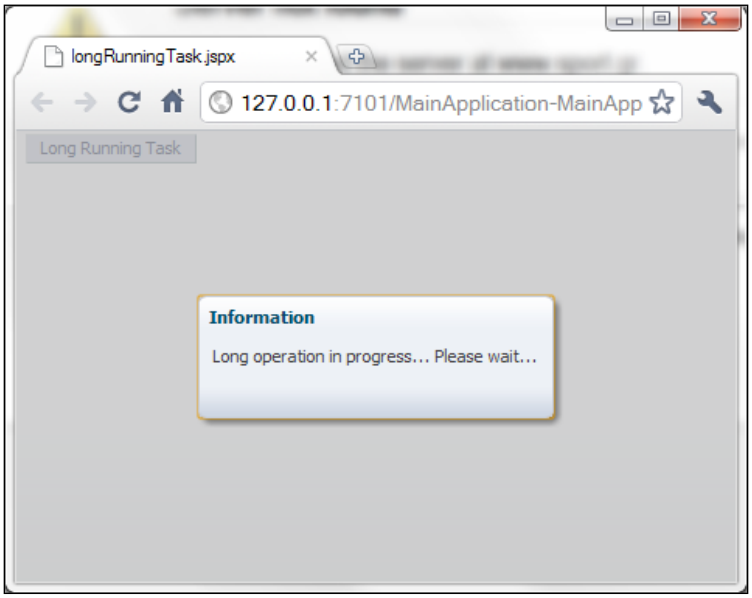
In steps 1 and 2 we created a JSPX page called `longRunningTask.jspx` and we added a button component to it. When pressed, the button will initiate a long running task through an action listener. The action listener is added to the button at steps 3 and 4. It is defined to a method called `longRunningTask()` 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 popup component (the popup 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 popup and we remove the busy state listener once the event completes.

Finally, in step 7, we added the `longRunningPopup` popup to the page. The popup is raised by the `busyStateListener()` as long as the event is busy (for 5 seconds). We made sure that the popup's `contentDelivery` attribute was set to `immediate` to deliver immediately the popup content 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 popup is raised for the duration of the long running task (the action listener in the managed bean). The popup is hidden once the long running task completes.



See also

Breaking up the application in multiple workspaces, chapter 1.

Using an af:popup to handle pending changes

In 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 the *Using a generic View Controller actions framework* recipe in chapter 1, to raise a popup message window informing you whether you want to commit the changes. We will add the popup window to the `TemplateDef1` page template definition that we created in the *Using page templates* recipe in chapter 1.

Getting ready

We will modify the `TemplateDef1` page template definition and the `CommonActions` actions framework. Both reside in the shared components workspace, which is deployed as an ADF Library JAR and it was introduced in the *Breaking up the application in multiple workspaces* recipe in chapter 1.

Furthermore, we will utilize the `HRComponents` workspace, also deployed as an ADF Library JAR. This workspace was introduced in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

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

How to do it...

1. Open the shared components 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 popup identifier to `CreatePendingChanges`. Add an embedded **Dialog** (`af:dialog`) component to the popup 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 like this:

```
<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 into 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 shared components workspace into an ADF Library JAR.
6. Open the main workspace application and ensure that both the shared components 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 that we added a **Continue** button, which when pressed saves the uncommitted changes. This is done through the buttons 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 there are indeed pending changes, it calls `onCreatePendingChanges()` to handle them. Otherwise, it calls `onContinueCreate()` to continue with the row creation action.

- `onCreatePendingChanges()` – The default implementation displays the `CreatePendingChanges` popup.
- `onContinueCreate()` – It is called either directly from `create()` if there are no pending changes, or from the `CreatePendingChanges` popup upon pressing the **Continue** button. Implements the actual row creation by calling the methods `onBeforeCreate()`, `onCreate()` and `onAfterCreate()`.
- `onBeforeCreate()` – It is called to handle any actions prior to the creation of the new row. The default implementation invokes the `Commit` action binding.
- `onCreate()` – It is called to handle the creation of the new row. The default implementation invokes the `CreateInsert` action binding.
- `onAfterCreate()` – It is called to handle any post creation actions. The default implementation does nothing.

In step 5 we redeploy the shared components workspace to an ADF Library JAR. Then, in step 6, we add it along with the `HRComponents` ADF Library JAR to the main application'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 used to raise a popup, prepends the popup 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 popup 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 popup message window indicating that there are pending unsaved transaction changes and committing the changes, as it is implemented in this recipe it applies specifically to the new row creation action. Additional functionality will need to be added for the other actions in your application, such as for instance navigating to the next, previous, first and last row in a collection.

See also

Determining whether the current transaction has pending changes, chapter 8.

Using page templates, chapter 1.

Using a generic View Controller actions framework, chapter 1.

Breaking up the application in multiple workspaces, chapter 1

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 that could be created optionally by JDeveloper.

In this recipe, we will show how 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 the *Breaking up the application in multiple workspaces* recipe in chapter 1.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in chapter 2.

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 it shown in the code fragment below:

```
<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 this:

```
<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 code fragment below:

```
<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 main application 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 we dropped (step 3) the `Employees` collection on the page as an **ADF Read-only Table** component (`af:table`). By doing this, JDeveloper also created the underlying iterator and tree bindings. Then, in step 4, we switched to the page bindings and changed the `EmployeesIterator` range size to our desired page size. Note that this page size is indicated in the managed bean created in step 6 by the constant definition `PAGE_SIZE` (it is 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 disable condition methods implemented by the `CollectionPaginationBean` managed bean (implemented in step 10).

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

- `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 we subtract from it the last page's rows. 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 **Next** 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.

Overriding remove() to delete associated children entities, chapter 2.

Using JasperReports

JasperReports is a third party library that can be interfaced with ADF in order to provide reporting capability to your Fusion Web Application. Reports are designed using accompanying software called **iReport Designer** and saved in an XML format on the server. Your ADF application subsequently, using the JasperReports Java libraries, opens the pre-designed XML report and displays it through the Jasper-supported viewer **JasperViewer** application. Using the JasperReports libraries the report may also be saved in a PDF or Excel format and opened subsequently by your application in those formats.

In this recipe we will design a report for the Employees in the HR schema using the iReport Designer tool. We will then demonstrate how to open the report from a Fusion Web Application and display it using JasperViewer.

Getting ready

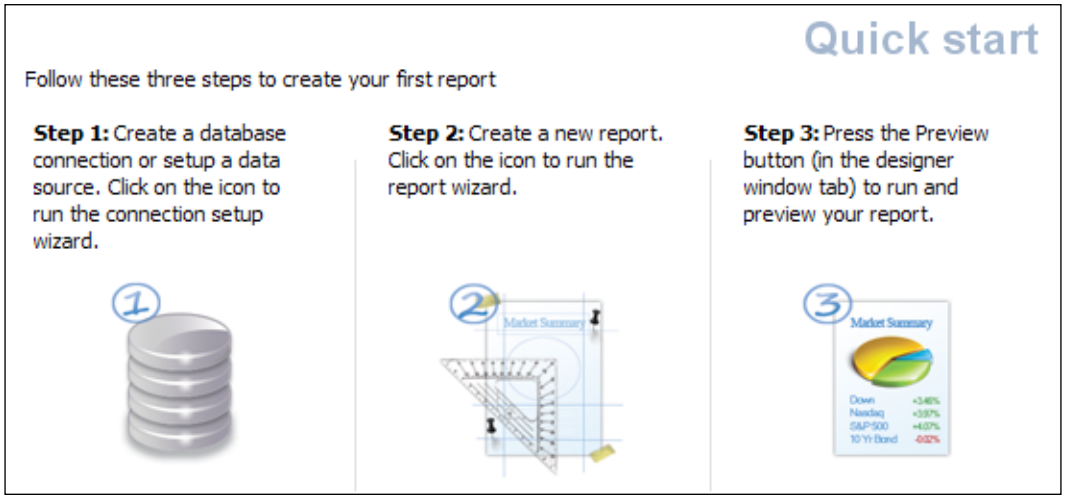
You will need to download and install the latest versions of both the JasperReports Library and the iReport Designer. At the time of this writing, these can be downloaded from `http://sourceforge.net/projects/jasperreports/` and `http://sourceforge.net/projects/ireport/files/` respectively. The version of the JasperReports Library and the iReport Designer used for this recipe is 4.1.2.

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* recipe in chapter 1.

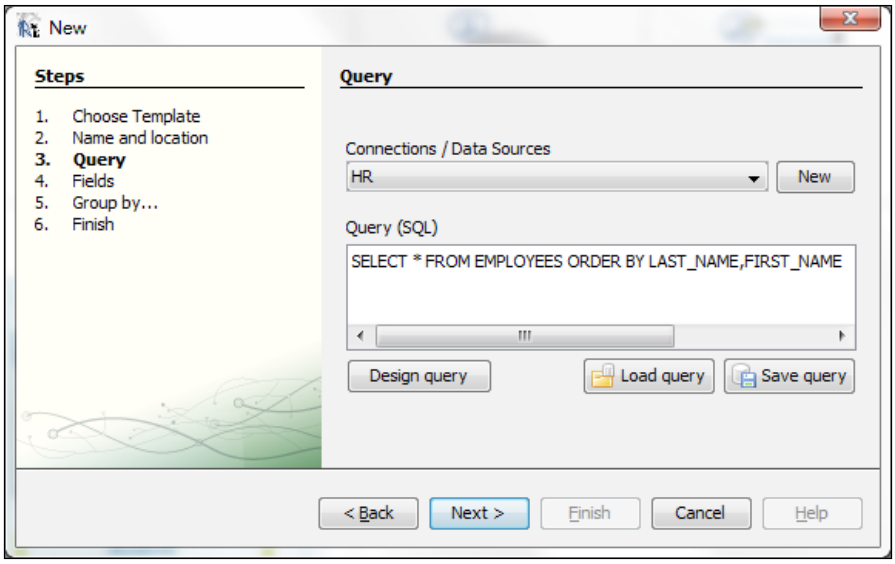
A database connection to the `HR` schema is required to create a report.

How to do it...

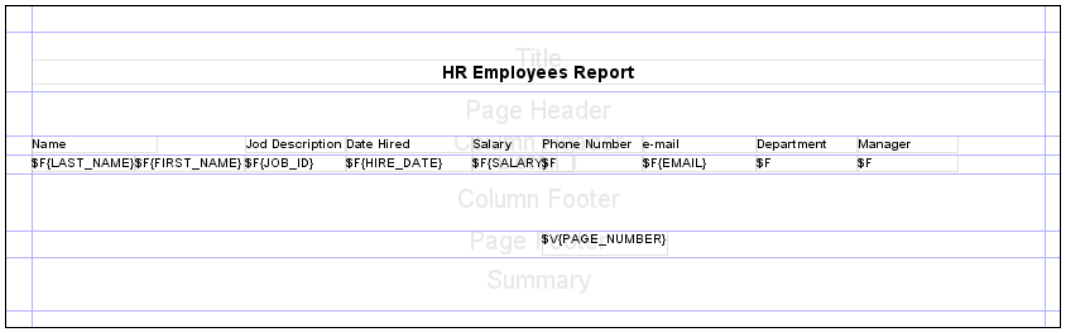
1. Start the iReport Designer tool by running the `ireport` application in the `bin` directory of the iReport Designer installation directory.
2. Open the iReport Designer **Options** dialog using the **Tools** > **Options** menu. In the **Classpath** tab click on the **Add JAR** button and add the Oracle JDBC library JAR to the iReport Classpath. The Oracle JDBC library JAR is called `ojdbc6.jar` and is located in the `wlserver_10.3/server/lib` directory under the Middleware Home directory.
3. Using the **Quick start** steps in the **Welcome Window**, click on **Step 1** to create a database connection for the `HR` database.



4. In the **Datasource** page select **Database JDBC Connection** and click **Next**. In the **Database JDBC connection** page, specify the database connection information. Click **Test** to test the connection and finally save the connection by clicking on the **Save** button.
5. Click on the **Quick start Step 2** to create a report. Select one of the blank templates and click on the **Launch Report Wizard** button. In the **Name and location** page specify the name and location on the server of the report. We will call the report `EmployeesReport`. Click **Next** to proceed.
6. In the **Query** page, with the data source created in step 2 selected, specify the SQL query used to gather the report data. Use the following query: `SELECT * FROM EMPLOYEES ORDER BY LAST_NAME, FIRST_NAME`. Then click **Next**.



7. In the **Fields** page, specify the `EMPLOYEES` table fields that you want to include in the report. Click **Next** when done.
8. Do not specify any groups in the **Group by...** page and click **Next**. Finally, in the **Finish** page, click on the **Finish** button to complete the creation of the report. The report should open in the designer.
9. Using the **Report Inspector**, complete the definition of the report by dropping the report fields from the **Fields** tree. Your report definition in the designer should look similar to the one shown below. You may preview the report by clicking on the **Preview** button. When done, ensure that the report is saved.



10. Open the main application workspace. Using the **Project Properties** > **Libraries and Classpath** dialog, add the following JasperReports library JARs: `jasperreports-4.1.2.jar`, `poi-3.6.jar`, `iText-2.1.7.jar` and `commons-digester-1.7.jar`.

11. Create a new JSP XML page called `jasperReports.jspx` using an existing quick start layout. Drop in the page a button component and adjust its **Text** property to **Employees Report**.
12. Create an action listener called `onEmployeesReport` for the button using a newly created managed bean called `JasperReportsBean`.
13. Open the `JasperReportsBean` managed bean and add the following code to the `onEmployeesReport()` method:

```
private static final String REPORT_DEFINITION =
    "/reports/EmployeesReport.jrxml";
private static final String REPORT_DATASOURCE =
    "java:comp/env/jdbc/HRConnectionDS ";

try {
    // load the report definition
    JasperDesign reportDefinition = JRXmlLoader.load(
        new FileInputStream(new File(REPORT_DEFINITION)));
    // compile the report
    JasperReport report =
        JasperCompileManager.compileReport(reportDefinition);
    // get report data from database
    InitialContext initialContext = new InitialContext();
    DataSource ds =
        (DataSource)initialContext.lookup(REPORT_DATASOURCE);
    JasperPrint print =
        JasperFillManager.fillReport(report,
           new HashMap<String, Object>(), ds.getConnection());
    // display the report using JasperViewer
    JasperViewer.viewReport(print, false);
} catch (Exception e) {
    // log the exception
}
```

14. Open the `web.xml` configuration file and the following resource definition for the `HRConnectionDS` data source:

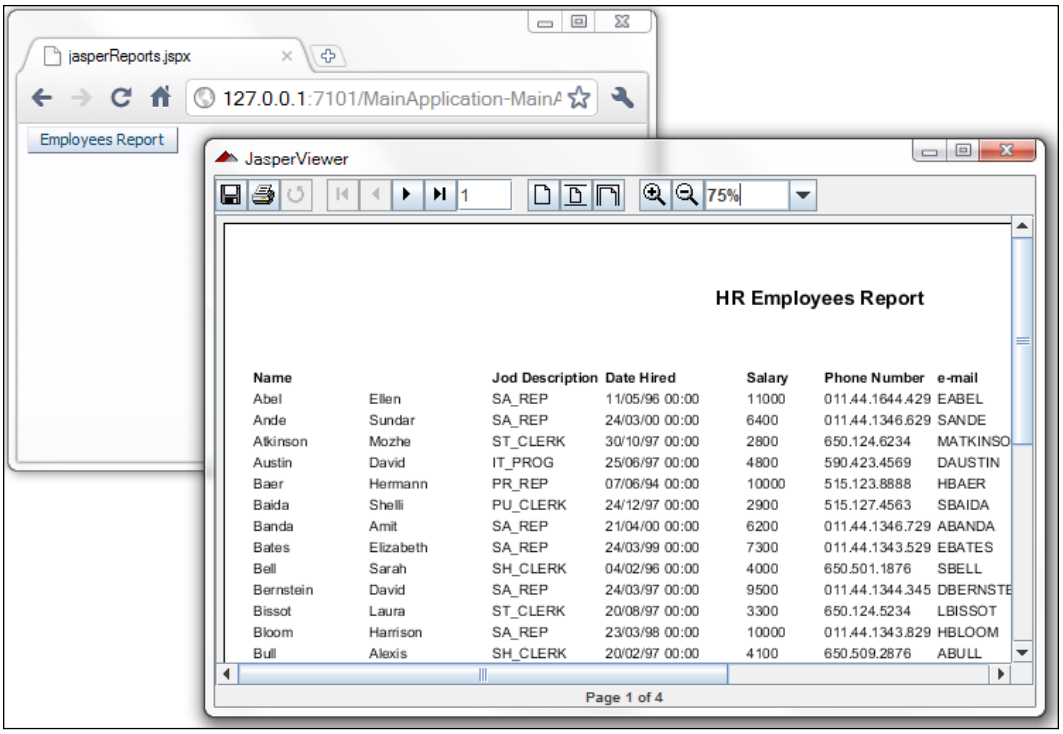
```
<resource-ref>
    <res-ref-name>jdbc/HRConnectionDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

15. Ensure that the `HRConnectionDS` data source is created in the WebLogic using the **Domain Structure** tree **Services** > **Data Sources** selection.

How it works...

In steps 1 through 9, we went through the process of defining a report using the iReport Designer application. The sample report, called `EmployeesReport`, will display a list of the all the employees in the `HR` schema. Prior to creating the actual report, we configured iReport by adding to its Classpath definition the Oracle JDBC library JAR (in step 2). The report is based on the `HR` data source created in steps 3 and 4. The actual report was created in steps 5 through 9. We used the **Report Wizard** available through the iReport **Welcome Window** to create a report based on a query that will retrieve all employees. This was done in step 6. The report was designed in step 9 using the iReport **Report Inspector** window and by dropping the report fields on the report canvas. While you design the report, you can preview it anytime using the **Preview** button. Once the report definition is complete, we saved it to a file called `EmployeesReport.jrxml`. We will load this report definition file and compile the report at runtime from our ADF application (see step 13) using the JasperReports library.

Steps 10 through 15 describe the necessary steps to interface JasperReports with an ADF Fusion Web Application and to load, compile and display a JasperReports report. We will be displaying the report from a managed bean, so the first step is to add the necessary JasperReports libraries to the applications ViewController project. These libraries are detailed in step 10. In steps 11 and 12, we created a page called `jasperReports.jspx` and we added a button to it that we will use to run the report. This is actually done through the button's action listener, implemented by a method called `onEmployeesReport()` defined in a managed bean called `JasperReportsBean`. The code in the `onEmployeesReport()` method uses the JasperReports libraries to first load the report definition from the `EmployeesReport.jrxml` report definition file and then to compile the report. It then fills the report with data originating from the `HR` database. Finally, it uses JasperViewer to display the report. This activates the JasperViewer application and displays the report as it is shown below:



Finally, in steps 14 and 15, we need to make sure that the data source used to retrieve the report data from the `HR` schema is both added as a resource definition to the application's `web.xml` configuration file and is defined in WebLogic.

There's more...

Using a JasperReports report definition the actual report may be converted to a number of other formats, such as to a PDF file or to an Excel spreadsheet for instance, and displayed using the corresponding application programs. The following code snippets show you how to convert and display a report definition to these formats.

Converting and displaying a report definition as a PDF file

```
private void showReportAsPDF(JasperPrint print)
     throws JRException, FileNotFoundException, IOException {
  final String PDF_FILE = "/reports/EmployeesReport.pdf";
  final String PDF_RUNTIME = "rundll32
       url.dll,FileProtocolHandler " + PDF_FILE;
```

```
    // export report to PDF
    ByteArrayOutputStream pdfStream =
             new ByteArrayOutputStream();
    JasperExportManager.exportReportToPdfStream(
                        print, pdfStream);

    // write PDF stream to a file
    OutputStream pdfFile = new FileOutputStream(
                       new File(PDF_FILE));
    pdfFile.write(pdfStream.toByteArray());
    pdfFile.flush();
    pdfFile.close();

    // display PDF file
    Runtime.getRuntime().exec(PDF_RUNTIME);
  }
```

Converting and displaying a report definition as an Excel spreadsheet

```
  private void showReportAsExcel(JasperPrint print)
     throws FileNotFoundException, JRException, IOException {
    final String EXCEL_FILE = "/reports/EmployeesReport.xls";
    final String EXCEL_RUNTIME = "rundll32
      url.dll,FileProtocolHandler " + EXCEL_FILE;

    // export report to Excel
    JRXlsExporter exporterXLS = new JRXlsExporter();
    ByteArrayOutputStream excelStream =
            new ByteArrayOutputStream();
    exporterXLS.setParameter(
      JRXlsExporterParameter.JASPER_PRINT, print);
    exporterXLS.setParameter(
      JRXlsExporterParameter.OUTPUT_STREAM, excelStream);
    exporterXLS.exportReport();

    // write stream to an Excel file
    OutputStream excelFile = new FileOutputStream(
                    new File(EXCEL_FILE));
    excelFile.write(excelStream.toByteArray());
    excelFile.flush();
```

```
    excelFile.close();

    // display spreadsheet
    Runtime.getRuntime().exec(EXCEL_RUNTIME);
  }
```

See also

Breaking up the application in multiple workspaces, chapter 1.