13
Building Business Services with EJB

So far our discussions have been centered on the usage of **ADF Business Components** (**ADF BC**) and building a user interface for ADF BC-based business services. In this chapter, you will learn how Oracle ADF will helps you to declaratively build user interfaces for **Enterprise Java Beans** (**EJB**) based services.

Following topics will be discussed in this chapter:

- The architecture of a Fusion web application, using EJB as a business service
- Generating a business service layer by using EJB
- Exposing an EJB service through data control
- Building a data bound edit page
- Oracle ADF binding architecture for EJB
- How does ADF Model data binding work in the Java EE application?
- Customizing error handling for the EJB services

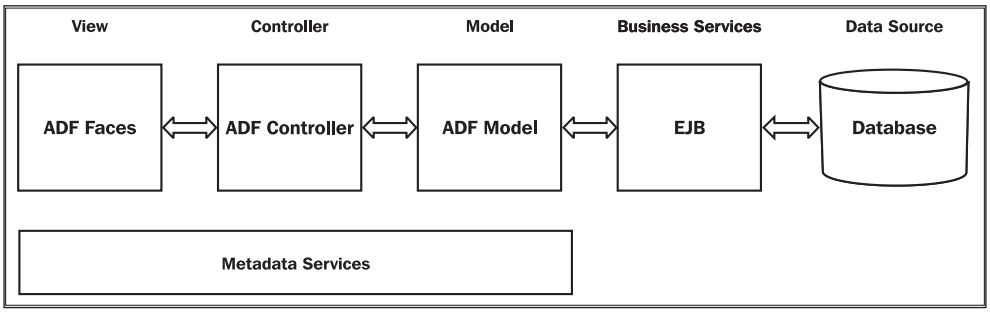
Introduction

The core **Java Enterprise Edition** (**Java EE**) technology stack has been improved considerably in the recent past. The business service development has become much easier with annotations, resource injection, EJB 3.x, and **Java Persistence API** (**JPA**). However, if you take a closer look at the implementations of a Java EE application, you may still see a large amount of repetitive code in different layers. In the next sections, you will learn the offerings from the ADF framework and JDeveloper IDE, for declaratively building a fully functional UI for EJB services without writing even a single line of glue code.

Architecture of a Fusion web application, using EJB as a business service

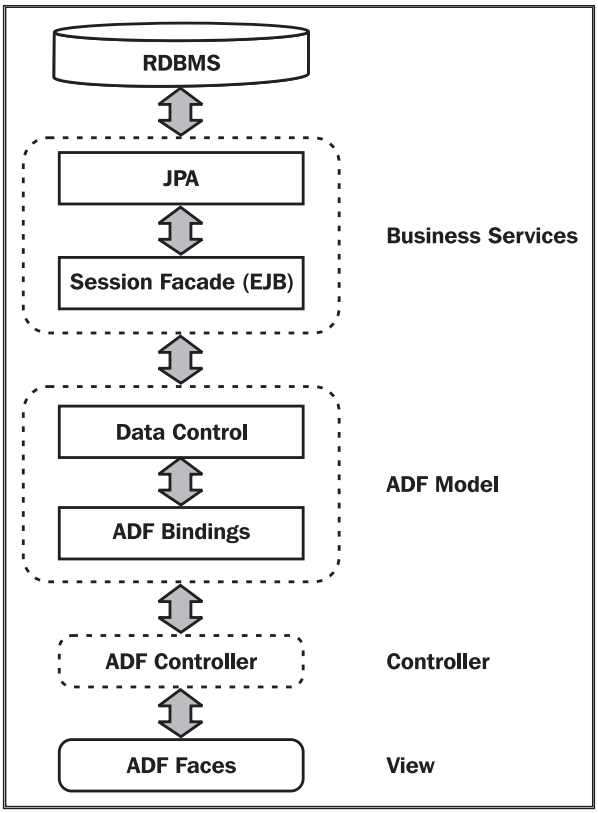
Oracle ADF, along with JDeveloper IDE, provides a visual and declarative UI development experience for the business services built by using EJB. This eases your job as a developer by freeing you from the complexity of the underlying technology stack used for building business services. Before we dig into the details of offerings from Oracle ADF for building an EJB-based application, let us see what makes Oracle ADF flexible enough to plug in different technologies.

We will start with architecture of an ADF web application by using EJB.
The following diagram depicts the basic architecture for such application:



In fact, architecture-wise, you may not see many differences between an ADF BC application and an EJB application. The view layer and controller remain the same between both the implementations. As you see in the diagram, the view layer is bound to EJB services through the ADF Model. In other words, the ADF Model decouples the view layer from the business service implementation. The advantage of this loose architecture is that it abstracts the technology used for building a business service from the UI layer and provides consistent development experience for UI developers across various business service implementations.

The ADF Model layer plays a very important role in keeping the UI layer neutral to the underlying business service implementation. The following diagram illustrates how the ADF Model glues the view layer with business services:



The ADF Model implementation is split into two parts, as follows:

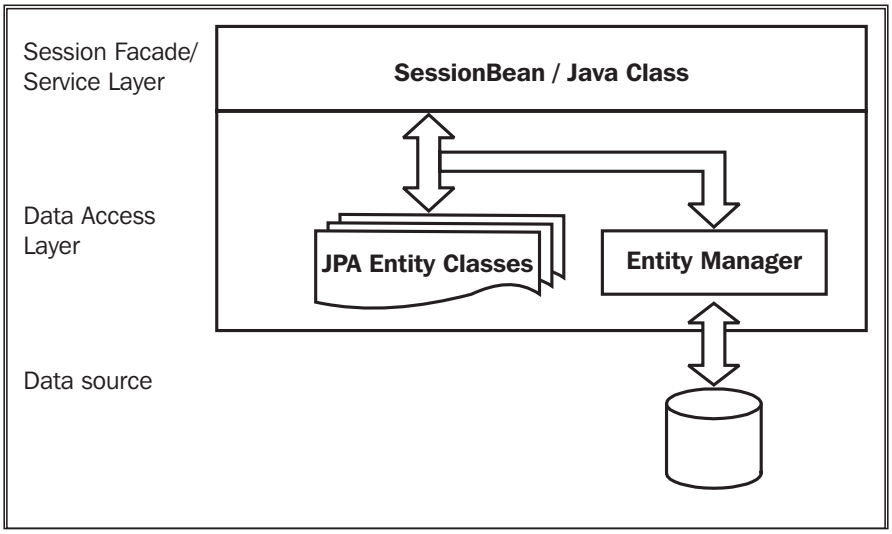
- **ADF bindings**: This layer provides a declarative data binding facility abstracting the data returned by the business services and method definitions. This provides a generic way to bind the view layer to the business services declaratively.
- **Data control**: The ADF data control acts as a smarter proxy for your business service layer. This layer abstracts the implementation technology of a business service by using standard metadata interfaces.

The separation of the ADF Model into two parts makes the binding framework more extensible and reusable. ADF offers many data controls out of the box to support various technologies such as ADF Business Components, EJB, web service, **Java Management Extensions** (**JMX**), and **Plain Old Java Object** (**POJO**).

In the next section, we will see how JDeveloper and ADF binding eases the UI development effort when the application uses the EJB services.

A quick look at the building blocks of an EJB model project

The following diagram displays the building blocks of a typical EJB-based business logic built by using the JDeveloper IDE:



Let us take a quick look at the core components of the EJB model project:

- **Session bean**: The session bean forms the service layer for an EJB-based business service implementation. There are two types of session bean—stateless and stateful. A stateless session bean does not maintain the state for a specific client, whereas a stateful bean maintains the a state until the client finishes using it.

- **JPA entities**: The JPA entities allow data management without wiring down SQL or JDBC. You can consider an entity class as a table in a relational database, and an entity instance as a row in that table.
- **Entity manager**: The entity manager manages the entities in an application. The entity classes managed by an entity manager are specified through a persistence unit (configured using `persistence.xml`). It is the persistence unit that stores information about the underlying data store (database) to be used by the entity manager.

Building user interfaces for EJB

The ADF binding layer for EJB bridges the gap between the UI and business service layer, and provides a visual and declarative UI development experience for EJB developers. In this section you will learn the following:

- Using JDeveloper IDE to build EJB business services
- Visually building UI for EJB business services

Generating the business service layer by using EJB

You might have used EJB for building applications in the past. If you are an experienced Java EE developer, in this section you will not learn anything new from the Java EE stack. However, you will definitely see how JDeveloper accelerates the pace of the Java EE application development.

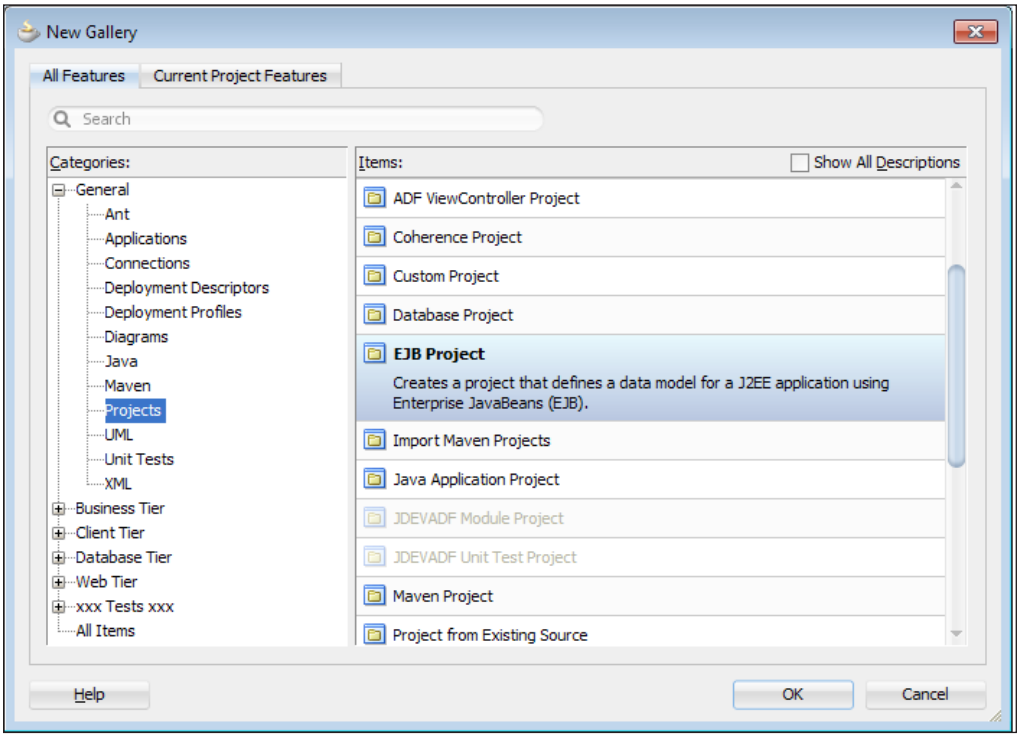
If you are starting from scratch, first you may need to build a Fusion web application, which can be extended to hold EJB services later. Note that you are not limited to any specific application template for using an EJB model. You can choose any template that meets your use casily. You can also add required libraries as and when required.

If you have an application ready and you want to add EJB services to it, open the application in JDeveloper.

To add an EJB project to an existing application, perform the following steps:

1. Choose **File | New** from the main menu toolbar.

2. In the **New Gallery** window, click on the **All Features** tab. Expand the **General** node and select **Projects | EJB Project**. Click on **OK** to continue. This is shown in the following screenshot:



3. In the **Create EJB Project** window, specify the project name and directory, and click on **Next** to continue.
4. In the **Project Java Settings** screen, specify the default package for the project and click on **Next**.
5. In the **Configure EJB Settings** section, specify **EJB Version** and its details. Mostly you may leave the default values set by the IDE in the **Create EJB Project** window as they are, unless you really want to override the settings.
6. Click on **Finish** to generate the model project template.

Generating entities from the database tables

Once you have created a template for the EJB model project, you can start adding model components to the project. To add entities from the database tables, follow these steps:

1. Right-click on the model project and select **New**.
2. In the **New Gallery** window, expand **Business Tier**, and select **Entities | Entities from Tables**.
3. In the **Create Entities from Tables** window, optionally override the persistence unit created by JDeveloper in the **Persistence Unit** screen. Click on **Next** to continue.
4. In the **Type of Connection** screen, select **Online Database Connection**, and click on **Next**.
5. In the **Database Connection Details** screen, create a new database connection. Alternatively, you can select an existing connection from the **Connection** drop-down list. Click on **Next** to continue.
6. In the **Select Tables** screen, click on the **Query** button to list the tables from the database. Select the desired tables for generating entities by shuttling them to the right-hand side list. Click on **Next** to continue.
7. In the **General Options** screen, specify **Package Name** and **Entity Class Options** as appropriate, and click on **Next**.
8. In the **Specify Entity Details** screen, optionally modify **Entity Details** populated by the IDE for each table displayed in the **Table Name** drop-down list. Click on **Next** to continue. Alternatively, you can click on the **Finish** button to complete the entity creation, accepting the default values set by the IDE.
9. If you decide to continue with the wizard by clicking on **Next** in the last step, IDE will display the **Relationships** screen. In the **Relationships** screen, you can pick up each entry in the **Potential Relationships** list and, if needed, override the default values set for the accessor filed (getter methods) names in source and destination entity objects.
10. Click on **Finish** to finish generating entities.

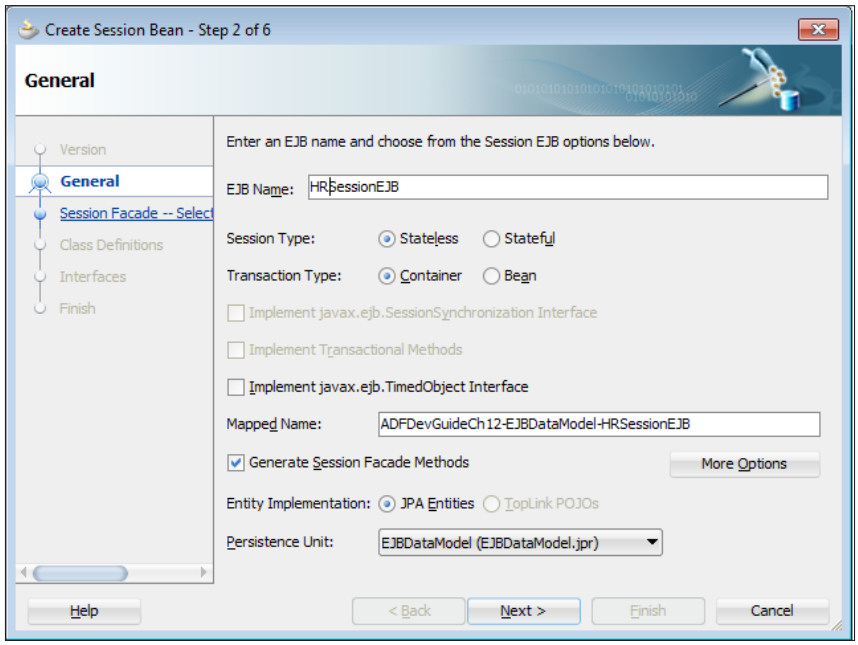
Building a service layer for the EJB model

Once the entities are added to a project, the next step is to expose the business functionalities through the appropriate service interface (session facade) to be used by a client. You can either use a session bean facade or Java facade to generate a service layer.

Generating a session bean service facade

To generate a session bean facade for the EJB model project, follow these steps:

1. Right-click on the EJB model project and select **New | New Gallery**.
2. In the **New Gallery** window, expand **Business Tier** and select **EJB | Session Bean** in the dialog.
3. In the **Create Session Bean** dialog, enter **EJB Name** and specify **Session Type** as **Stateless** and **Transaction Type** as **Container**. Select the appropriate **Persistence Unit**. This step is displayed in the following screenshot. Click on Next to continue the wizard.



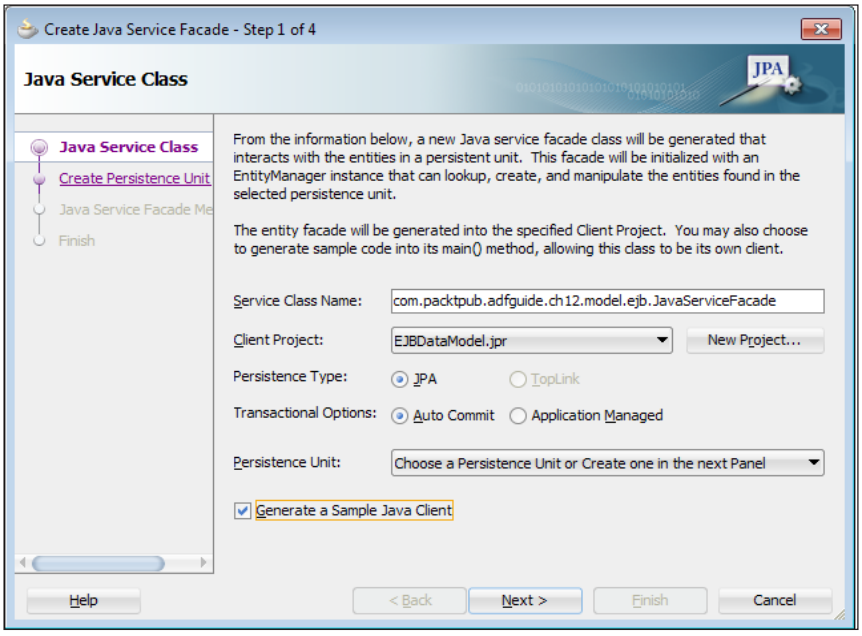
4. In the **Session Facade** screen, select the core façade methods and the CRUD service methods for each entity. Click on **Next** to continue. Alternatively, you can click on **Finish** to complete the creation of the session bean, accepting the default settings for the bean.

5. In the **Class Definition** screen, enter the bean class and directory. Click on **Next** to continue.
6. In the **Interface** screen, select how you want to expose the service to the client. Choose **Remote Interface** to expose the services to be used by a remote client. Then, choose **Local Interface** in all other cases.
7. Click on **Finish** to generate the session bean facade.

Generating a Java service facade

As mentioned earlier, session bean is not the only option to generate service layer for the JPA model. If you don't want to use EJB in your application, use the **Java Service Facade** option. To generate a Java service façade for the JPA entities, follow these steps:

1. Right-click on the EJB model project and select **New | New Gallery**.
2. In the **New Gallery** window, select **EJB** in the **Business Tier** section and then select the **Java Service Facade** item.
3. In the Create Java Service Class dialog, enter the Java service class name in the **Service Class Name** section. Then, select **Client Project**, where you want the IDE to generate the source file for the Java service facade. Then, choose **Persistence Type** for the JPA model as **JPA** or **Toplink**. Then, choose **Transactional Options** as **Auto Commit** or **Application Managed**. This is shown in the following screenshot:



You are required to specify **Persistence Unit** for the `javax.persistence.EntityManager` in this screen. You can either specify an existing persistence unit name from the project or opt for creating a new persistence unit in the next screen. Click on **Next** to continue the wizard.

4. If you have opted to create a new persistence unit in the previous step, then the next screen would be the **Create Persistence Unit** screen where you can enter the persistence unit name and specify JDBC connection. Click on **Next** to continue.
5. In the **Java Service Facade Methods** screen, select the methods that you want to expose to the client. Click on **Finish** to generate data control.

If you are a beginner or not familiar with the Java EE and JPA programming model, the options that you see in the Create Service wizard may confuse you. In reality, you do not need to worry much about these configuration options. Mostly the default values populated by the IDE in the **Create Java Service Class** wizard screen work for you, so that you really don't need to bother overriding these values while generating the service layer.

Exposing an EJB service through data control

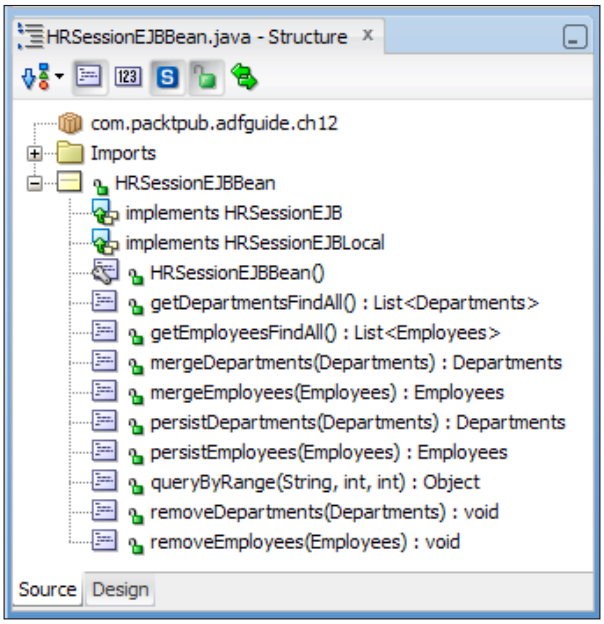
Once you have finish building the EJB services, the next step is to bind the services with the UI layer by using ADF binding features. ADF uses data control to decouple bindings in the UI layer from business service implementation. To generate data control for accessing services exposed in a session bean, right-click on the appropriate session bean implementation class in the application navigator and choose **Create Data Control** in the context menu. In **Choose EJB Interface**, select **Local** and click on **OK**.

What happens when you generate data control for a session bean

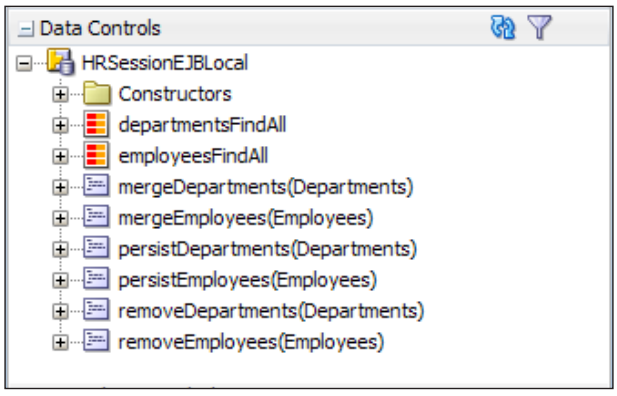
When you generate data control for a session bean implementation, JDeveloper IDE generates a `DataControls.dcx` file in the same folder where the session bean resides. The `.dcx` file records metadata for invoking service methods defined in a session bean from a client. A typical data control descriptor file for an EJB will contain an `<AdapterDataControl>` entry describing the data control and the target bean definition. The following is an example for a data control descriptor for the `HRServiceSessionEJB` bean:

```
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
version="11.1.2.60.81" id="DataControls"
Package="com.packtpub.adfguide.ch12.model.ejb">
  <AdapterDataControl id="HRServiceSessionEJBLocal"
    FactoryClass=
      "oracle.adf.model.adapter.DataControlFactoryImpl"
    ImplDef=
      "oracle.adfinternal.model.adapter.ejb.EjbDCDefinition"
    SupportsTransactions="false"
    SupportsSortCollection="true" SupportsResetState="false"
    SupportsRangesize="false"
    SupportsFindMode="false" SupportsUpdates="true"
    Definition=
    "com.packtpub.adfguide.ch12.model.ejb.HRSessionEJBLocal"
    BeanClass=
    "com.packtpub.adfguide.ch12.model.ejb.HRSessionEJBLocal"
    xmlns="http://xmlns.oracle.com/adfm/datacontrol">
    <CreatableTypes>
      <TypeInfo FullName=
        "com.packtpub.adfguide.ch12.model.ejb.Employees"/>
      <TypeInfo FullName=
        "com.packtpub.adfguide.ch12.model.ejb.Departments"/>
    </CreatableTypes>
    <Source>
      <ejb-definition ejb-version="3.0"
      ejb-name="HRServiceSessionEJB"
      ejb-type="Session"
      ejb-business-interface=
      "com.packtpub.adfguide.ch12.model.ejb.HRSessionEJBLocal"
      ejb-interface-type="local"
      initial-context-factory=
      "weblogic.jndi.WLInitialContextFactory"
      DataControlHandler=
      "oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler"
      xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
    </Source>
  </AdapterDataControl>
</DataControlConfigs>
```

Once you have finish building the data control for a session bean, the data control pallet will display the business services and data collection exposed by the session bean implementations. For example, consider the **HRSessionEJBBean** class shown in the following screenshot:



When you generate the data control for `HRSessionEJBBean`, the **Data Controls** panel will expose the business service methods and data collections defined in `HRSessionEJBBean` as follows:



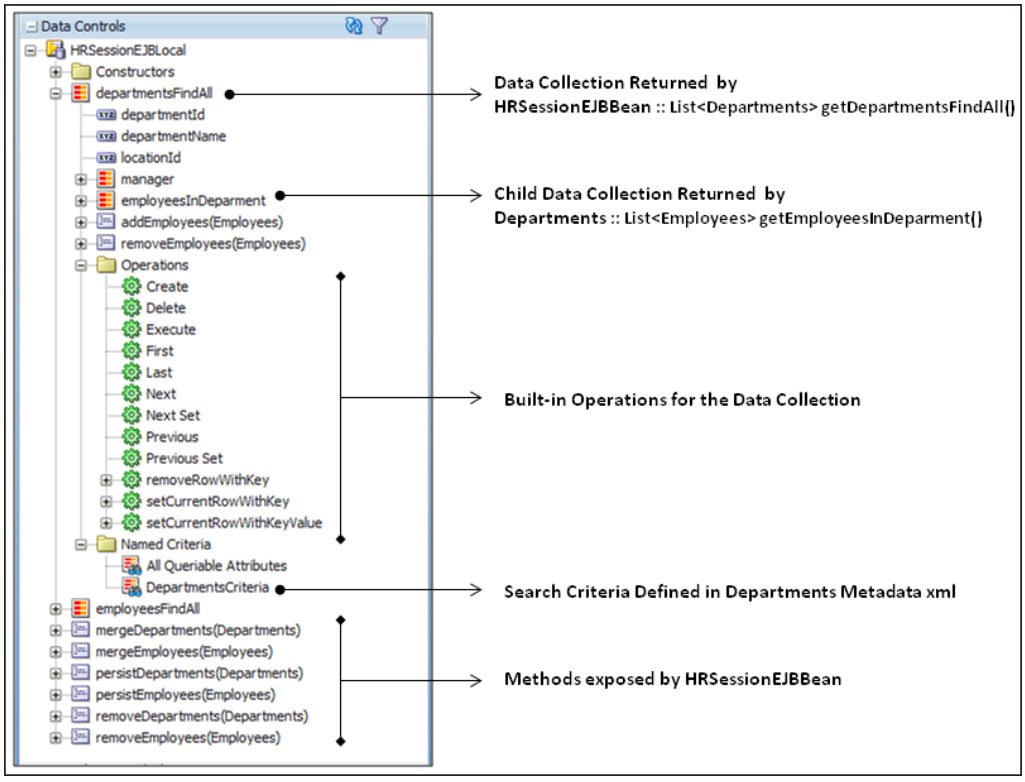
The data control generated for the session bean allows you to declaratively create a UI by dropping appropriate items on to the page.

A closer look at the Data Controls panel

The **Data Controls** panel exposes the business service methods and data collections defined in the bean implementation for building a data bound UI. The **Data Controls** panel displays the following items:

- **Data collection**: The data control tree displays the data collection retuned by the getter methods as data collection nodes. If an entity in the data collection contains a relation to another entity, the data control tree will display the detailed data collection nested under its master data collection node.
- **Named criteria**: The **Named Criteria** node under the data collection represents a declarative query model, which can be used for building a search form.
- **Business service methods**: This represents custom business methods defined in the bean.
- **Built-in operations**: The **Operations** node under the data collection displays the built-in operations that are provided by the framework.

Let us take an example to understand the items exposed by the **Data Controls** panel. The following screenshot displays the data control generated for `HRSessionEJB`:



In the preceding screenshot, the **Data Controls** panel displays two data collections `departmentFindAll` and `employeesFindAll`. If you are curious to know how IDE generates these data collections, here are the details.

When you create a data control for a bean, the **Data Controls** panel displays the getter (accessor) methods, returning collection in the bean as data collection. If the method in the bean, which returns the collection takes a parameter, the **Data Controls** panel displays it as a method. The methods in the data control can be bound to the command components on the UI.

The data control offers declarative means to manipulate elements in the collection when used in the UI. Under each data collection node in the **Data Controls** panel, you can see the **Operation** node that contains built-in operations for navigation over collection and operations for basic data manipulations such as **Create**, **Delete**, and **Execute**. These operations can be dropped as action components on the UI.

Under the data collection node in the **Data Controls** panel, you can see a **Named Criteria** node. The named criteria represents the declarative query model, which can be dropped as a search component in the UI. This node will contain an **All Queriable Attributes** criteria by default and additional criteria that you add explicitly for the entity present in the collection.

Generating the data control structure XML file for the bean in the data collection

When you create a data control for a bean that returns a collection, by default, the attributes listed under the data collection and their properties are derived from the entity Java class present in the model project. The ADF framework allows you to optionally override the default properties for each attribute present in the entity by generating a structure XML file. The structure XML file for an entity may contain attribute definitions, validations, accessor methods for associated entities, and named criteria. Note that when you create a data control, IDE will not generate any structure XML files for entities used in the underlying service implementation, by default.

To generate a structure XML file for a bean in the data collection, perform the following steps:

1. Open `DataControls.dcx` file, which owns the collection in the overview editor.
2. Select the data collection node for which you want to generate a structure XML file, right-click on it, and select **Edit Definition** icon displayed in the **Data Controls** panel.

When you do this, IDE generates a sparse structure XML for the entity (bean) used in the data collection, if the following conditions are met:

- The selected node in the **Data Controls** panel must be a data collection returned by an accessor method
- The collection returned by the accessor must have a valid element type specified, for example, `public List<Departments> getDepartmentsFindAll()`

The structure XML file is really sparse and does not contain any metadata when you generate it. To edit the default properties for an entity, open the structure file in the overview editor and alter the properties as appropriate. You can edit the following properties:

- Edit properties on existing attributes such as primary key, updatable, default value, UI hints, and validation rules
- Add transient attributes

Note that transient attributes added in the structure XML are managed by the ADF binding layer. The JPA entity also allows you to mark an attribute in the entity class as transient through the `@Transient` annotation, which is managed by the entity manager. Both are an acceptable approaches for defining non-persistent attributes when you use EJB with ADF.

- Add a named criteria definition

When you edit properties, they are stored in the structure XML file. The following is an example of the structure XML file created for the `Departments` entity:

```
<PDefViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="Departments">
    <PDefAttribute
    Name="departmentName">
    <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId=
"${adfBundle['model.ejb.HRDataModelEJBBundle']'model.ejb.Departments.
departmentName_LABEL']}"/>
      </SchemaBasedProperties>
    </Properties>
  </PDefAttribute>
</PDefViewObject>
```

Defining UI hints for attributes in a bean

To define UI hints for attributes in a bean, open the structure XML file of the entity (bean) in the overview editor. Click on the **Attributes** tab and then select the desired attributes for which you want to specify or override UI hints. The following table lists the commonly used properties for an attribute in a bean definition:

Property name	Description
Key attribute	This indicates if the attribute is marked as a key attribute in the bean. At least one attribute should be marked as a key attribute in a bean for the framework to function properly. For example, the ADF binding layer uses key attributes to uniquely identify each row in a collection.
Updatable	This specifies whether the attribute value is updatable by the client or not.
Queryable	This specifies if the attribute should appear in the search form generated for the All Queryable Attributes view criteria for a data collection.
Default value	This specifies the default value for an attribute for a new bean instance. The default value can be set either as static literals or Groovy expressions.
Display hint	This specifies display hints property, such as **Hide** or **Display** for an attribute.
Label	This specifies the label for the attribute.
Tooltip	This specifies the tool tip for the attribute.
Control type	This specifies the control type for the attribute. This is used by IDE to add the appropriate JSF tag when you drop the collection on to the page. This is also used by dynamically rendering UI to decide the control type for an attribute.
Category	This specifies the category of the group attributes. This is used by the dynamic rendering UI to group attributes for display.

Property name	Description
Field order	This specifies the field order hint to order how the user interface will render the attribute values within its category. This is useful to control the order of attributes in dynamic UI components.
Auto submit	This specifies the auto submit property for the attribute.

Defining validations for attributes in a bean

ADF allows you to declaratively add validation rules for entity attributes. To define a validation, follow these steps:

1. Right-click on the desired data collection in the **Data Controls** panel and select **Edit Definition**. While doing so IDE opens up the structure XML file in the overview editor. In the overview editor, click on the **Validation Rules** tab and click on the green plus icon to add validation.
2. In the **Add Validation** dialog, select **Rule Type** and specify the definition for the selected rule. Click on the **Failure Handling** tab and specify the severity and error message text.
3. Click on **OK** to save changes.

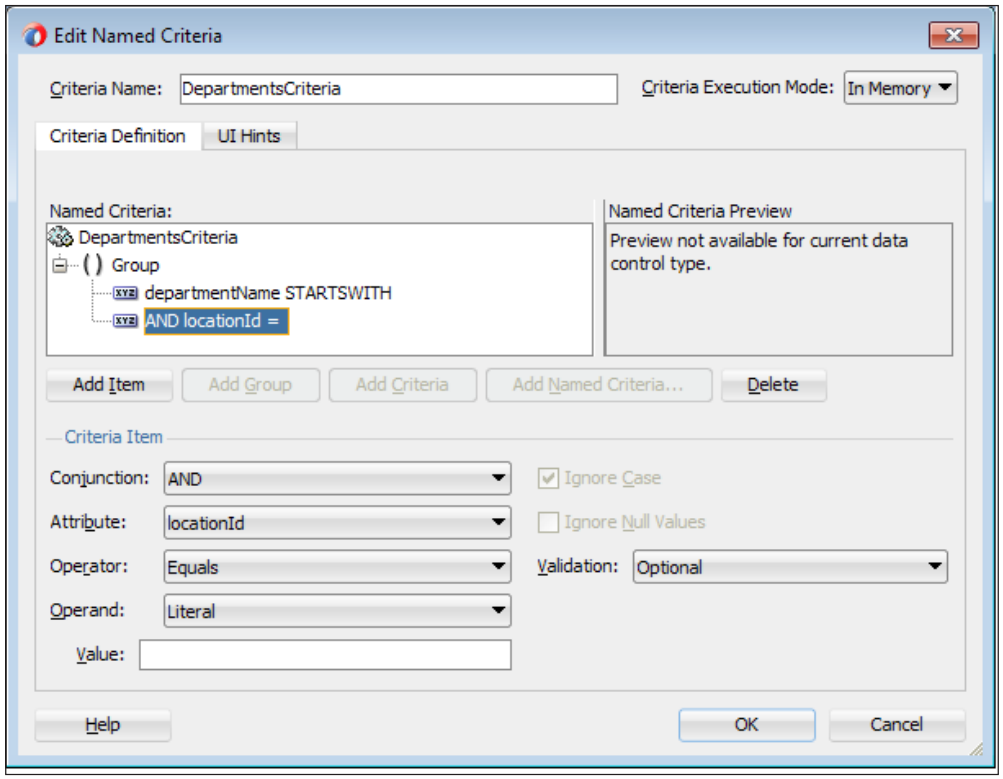
Defining named criteria

The named criteria are used to filter the data collection returned by the accessor methods. You can use named criteria to declaratively build the query-by-example search forms.

To define named criteria, follow these steps:

1. Open the structure XML file for the entity (bean) in the overview editor. Select the **Named Criteria** tab and click on the green plus icon in the **Named Criteria** section to create a new criteria.
2. In the **Create Named Criteria** screen, enter the name and specify the criteria execution mode. The default execution mode is **In Memory**. Add a criteria item, a criteria group, or other existing named criteria as appropriate, and click on **OK** to save changes.

Once you have define the named criteria for an entity, the same will be displayed under the respective data collection node in the **Data Controls** panel as a child of the **Named Criteria** node. The following screenshot displays the named criteria item definition for the `Department` bean:



We have discussed the options for defining the **Named Criteria**, available in the **Create Named Criteria** screen, during our discussions on the view objects. If you need a quick brush-up on this subject, refer to the topic *View Criteria* in *Chapter 4*, *Introducing View Object*.

Building search UI for EJB

You can use named criteria to declaratively build a search form for your JPA model. To build a search form, expand the accessor-returned data collection node in the **Data Controls** panel, then select the desired named criteria displayed below the **Named Criteria** node, and drop it on to the page by choosing **ADF Query Panel** (or other appropriate search template) in the context menu. This action adds the

`af:query` component to the page and generates the necessary search binding.

At runtime, when you click on the **Search** button, the framework generates the necessary **Java Persistence Query Language** (**JPQL**) and executes the query. The framework uses a preconfigured method present in the session bean for executing the search:

```
public Object queryByRange(String jpqlStmt, int firstResult,
int maxResults)
```

This method is added to the bean source by the IDE when it is created.

Building a data bound edit page

The ADF binding framework along with JDeveloper IDE offers a visual and declarative UI development experience even for EJB business services. Once you generated data controls. The steps for building a UI for EJB are more or less the same as what we have discussed for business components in *Chapter 7*, *Building Business Services with User Interface* and *Chapter 8*, *Building Data Bound Web User Interfaces*.

To build an edit page, select the data collection in the **Data Controls** panel that returns the data you want to display and drop it on to a page by choosing **Form | ADF Form** in the context menu. In the **Edit Form Fields** dialog, delete the unwanted fields and select **Include Navigation Controls** to display navigation controls as appropriate.

Adding the create functionality

To add create functionality in a page, drop the built-in **Create** operation for the data collection from the **Data Controls** panel on to the page as suitable command components such as ADF Button. At runtime, when this component is clicked, the framework creates a new entity instance and adds it to the data collection. Note that this step just helps you to create rows in memory. To post changes to the database, you may need to call the appropriate methods as discussed under the section *Saving changes to database*.

Saving changes to the database

The new rows that are added by using the built-in **Create** operation, or changes done on the entity instances in the data collection at runtime, are not persisted in the database by default. When you use the JPA entity for building a persistence layer, you may have to call either the `merge()` or `persist()` method on `javax.persistence.EntityManager` for posting changes to the database. The `merge()` method call creates a new entity instance, copies the state from the supplied entity,

and makes the new copy managed, whereas the `persist()` method adds the supplied entity instance to the persistence context and makes that instance managed. To keep things simple, you can use `persist()` for persisting new entities and `merge()` can be used for updating existing entities.

When you generate a session bean or Java service facade for a JPA model, IDE generates the separate `merge()` and `persist()` methods for each entity present in `persistence.xml`. An example is here:

```
@Stateless(name = "HRSessionEJB", mappedName = "ADFDevGuideCh12-
EJBDataModel-HRSessionEJB")
public class HRSessionEJBBean implements HRSessionEJB,
HRSessionEJBLocal {

  @PersistenceContext(unitName="EJBDataModel")
  private EntityManager em;

  public Departments persistDepartments(Departments
  departments){
  em.persist(departments);
  return departments;
  }
}
```

Drop the appropriate entity merge method on to the page as an ADF Button for adding save functionality for an entity. While doing so, JDeveloper IDE will ask you to specify an appropriate entity instance as the value for the method parameter. Typically, the value would be the row that is being currently edited in the UI. You can specify the currently edited entity instance, using binding EL. For example, the following EL refers the `Departments` instance in `departmentsFindAllIterator` that is currently being selected in the UI:

```
#{bindings.departmentsFindAllIterator.currentRow.dataProvider}
```

The following code snippet illustrates how a save button is bound to the `mergeDepartments()` method in `HRServiceSessionEJB`:

```
<af:commandButton actionListener="#{bindings.mergeDepartments.
execute}" text="Save Departments"    id="cb6"/>
```

The method action binding present in the page definition file for the `mergeDepartments <methodAction>` is as follows:

```
<methodAction id="mergeDepartments"
  RequiresUpdateModel="true" Action="invokeMethod"
  MethodName="mergeDepartments"
  IsViewObjectMethod="false"
  DataControl="HRServiceSessionEJBLocal"
  InstanceName="data.HRServiceSessionEJBLocal.dataProvider"
  ReturnName="data.HRServiceSessionEJBLocal.
  methodResults.mergeDepartments_HRServiceSessionEJBLocal
  _dataProvider_mergeDepartments_result">
  <NamedData NDName="departments" NDValue=
  "#{bindings.departmentsFindAllIterator.currentRow.dataProvider}"
  NDType="com.packtpub.adfguide.ch12.model.ejb.Departments"/>
</methodAction>
```

Oracle ADF binding architecture for EJB

We covered the basic steps for building a UI for EJB services in the earlier sections. Now you might be curious to know how ADF binding interacts with the EJB layer. The remaining section of this chapter discusses this topic in detail.

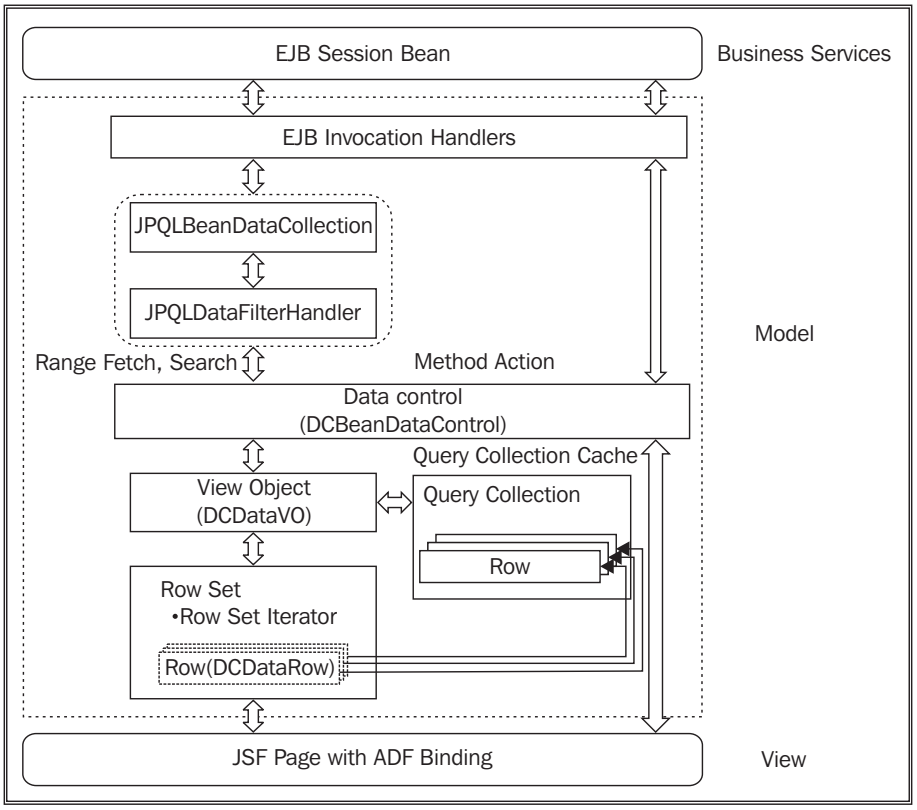
This section discusses how the ADF model wires the UI elements with data collections from the EJB-based business service methods.

When you drop a data collection from the **Data Controls** panel on to a page, IDE updates the page definition file with the `<accessorIterator>` binding entries. The `<accessorIterator>` binding manages iteration over the collection in an EJB or Java bean data control. Here is an example:

```
<iterator Binds="root" RangeSize="25" DataControl="HRServiceSessionEJB
Local" id="HRServiceSessionEJBLocalIterator"/>
<accessorIterator MasterBinding="HRServiceSessionEJBLocalIterator"
Binds="departmentsFindAll" RangeSize="25"
DataControl="HRServiceSessionEJBLocal"
BeanClass="com.packtpub.adfguide.ch12.model.ejb.Departments" id="depar
tmentsFindAllIterator"/>
```

The `MasterBinding` entry binds to the iterator that refers the master to the accessor iterator. This is used for simulating the master-detail hierarchy in the binding container.

In nutshell, the data collections returned by the getter method from EJB are wired to UI controls through iterator bindings. Now, you must be curious to know about the framework components involved in wiring the UI with the data collection returned by the getter methods defined in EJB or the Java bean. The following diagram illustrates various components used by the ADF Model to invoke EJB services:



Behind the scenes, the framework uses the row set object to aggregate the data collection returned from EJB. Each element in the row set is of type `oracle.adf.model.bean.DCDataRow`. The row set object returns a default row set iterator instance to iterate over the collection, which is used by the framework to populate the UI. Row set do not store the rows directly within them, rather they use a query collection object for storing rows. Keeping the rows in a query collection allows multiple row set iterators with the same "query filters" to re-use the same collection. Each element in the row set points to the row stored in the query collection instance. Each row in the query collection holds a data provider instance, which is a JPA entity if the underlying data model is built using JPA.

Now you may ask, who manages the row set and controls the data population logic

when the client iterates over the rows in a row set. For ADF Business Components, this is done by `oracle.jbo.server.ViewObjectImpl`. For the EJB or Java Bean data control, the ADF binding layer uses a specialized view object implementation class, known as `oracle.adf.model.bean.DCDataVO`, to do this job. `DCDataVO` makes sure that the rows are populated when a client iterates over the rows. All data requests from `DCDataVO` to the underlying Java or EJB services are routed through the `DCBeanDataControl` class that manages the connection to the data provider service implementation. To enable range fetching, sorting, and search functionality, the framework engages `JPQLDataFilterHandler` at runtime. This happens when `DCDataVO` reads the accessor methods defined in the EJB or Java services.

Pagination support in the bean data control

When you create data control for an EJB session bean, IDE generates `DataControls.dcx` for the bean. The metadata section in a data control contains entries for invoking EJB services as well as configurations parameters that decide the data control behavior at runtime when data collections exposed by EJB are accessed by the client.

DataControlHandler

When the client accesses the data collection exposed by a Java bean or EJB session bean, the binding layer engages the data control handler class specified in the data control definition file. This class can be set appropriately to enable pagination, sort, and query features on the data collection retuned by the accessor methods. The following example illustrates the `DataControlHandler` configuration for an EJB session bean:

```
<AdapterDataControl
  ...

<ejb-definition ejb-version="3.0" ejb-name="HRServiceSessionEJB" ejb-
type="Session"
  ejb-business-interface="com.packtpub.adfguide.ch12.
  model.ejb.HRServiceSessionEJBLocal"
  ejb-interface-type="local" initial-context-
  factory="weblogic.jndi.WLInitialContextFactory"
  DataControlHandler=
  "oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler"
  xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>

</AdapterDataControl>
```

The following are the two possible `DataControlHandler` implementations offered by the framework:

- `oracle.adf.model.adapter.bean.DataFilterHandler`: This supports both pagination and in-memory filtering—typically used for simple Java bean data control. If you specify `oracle.adf.model.adapter.bean.DataFilterHandler` as `DataControlHandler` in the `.dcx` file, the Java service class, on which data control is built, is required to have two more additional methods with predefined contracts:
    - `public long get<AccessorName>Size()`: A method for returning total records
    - `public List<T> get<AccessorName>(int start, int increment)`: A range accessor method that returns a paginated list whose page size is set through the value for the parameter increment passed to the method

  For example, to enable pagination for a method `public List<Departments> getDepartments()` exposed in a data control, the Java bean class should have an implementation for the following methods:

```
public List<Departments> getDepartments(){
  //Return department list
  return departmentList;
}
public List<Departments> getDepartments(int start, int increment){
  //Return departments in the range set by the caller
        return getPaginatedDepartmentList(start, increment);
}
public long getDepartmentsSize(){
  // Return total
        return getTotalDepartments();
}
```

- `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler`: This is the default `DataFilterHandler` implementation, when you generate data control for the Java service facade or session bean facade that uses the JPA data model. `JPQLDataFilterHandler` supports pagination, filtering, and query criteria out of the box.

  When you build a session EJB or Java service facade for your JPA-based model project, IDE will add the following method in your service implementation:

```
public Object queryByRange(String jpqlStmt, int firstResult, int
maxResults){ ... }
```

This method acts as a common entry point for all infrastructural service calls such as count queries, range fetching, and search and sort operations. It is the `JPQLDataFilterHandler` class that prepares the requests from the binding layer before passing it to the `queryByRange()` method defined in the bean implementation.

Disabling pagination and default data control features

If you remove the `DataControlHandler` attribute, `DCBeanDataControl` directly invokes the accessor method, by passing the handlers. The built-in support for named queries and pagination is disabled with this setting.

What you may need to know about pagination support for data collection in a bean data control

Keep the following points in mind while using the paginated data collection from a bean data control:

- Only the data collection type returned by a getter method (for example, `public List<Departments> getDepartmentsFindAll()`) supports pagination. The non-getter method does not support pagination.
- In a master-details scenario, only the master list is paginated and the details list is not paginated.

How does ADF Model data binding work in JavaEE Application?

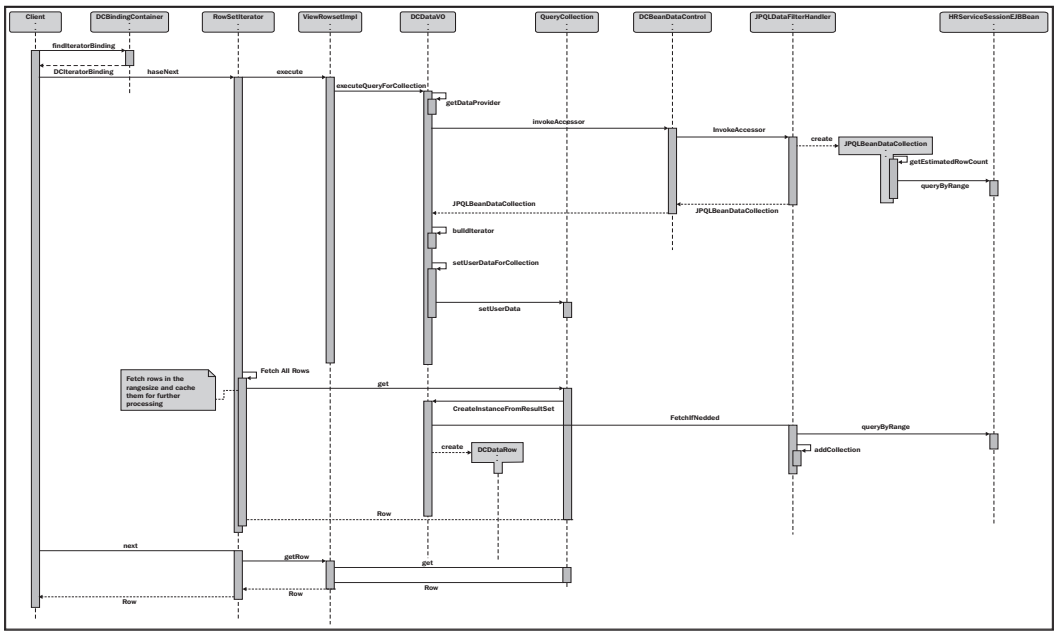
In this section, we will discuss how ADF data binding interacts with EJB-based business services at runtime and the role of various framework components in this exercise. Take a look at the following code snippet from a managed bean that reads data from a getter method (the accessor method) defined in an EJB session bean using binding APIs:

```
In managed bean code

//Get the current binding container
DCBindingContainer dcb =
  (DCBindingContainer)BindingContext.getCurrent().
  getCurrentBindingsEntry();
//Get the departmentsFindAllIterator iterator
```

```
DCIteratorBinding dcib =
  dcb.findIteratorBinding("departmentsFindAllIterator");
//Iterate over rows from the iterator
RowSetIterator rowIter = dcib.getRowSetIterator();
rowIter.reset();
while (rowIter.hasNext()) {
  Row row = rowIter.next();
}
```

The following sequence diagram illustrates the interaction between the ADF model framework components when the client executes the preceding code snippet to read the data from the data collection returned by an accessor method defined in the EJB session bean:



In this example, the client reads an iterator executable from the binding container and starts the iteration over rows by calling `hasNext()` on the `RowSetIterator` object. As this is the very first call that refers the underlying data collection and it is not yet read in the execution cycle, the `RowSetIterator` object will fire `executeQuery()` on the row set implementation class that manages the data collection for the client.

Note that the row set, by design, neither queries the datasource nor controls the data read operation; rather its role is to manage the returned collection to be used by the client. The framework deploys the `oracle.adf.model.bean.DCDataVO` object to control the execution cycle for a row set. Conceptually, this component is the same as the view object implementation that we discussed in *Chapter 4*, *Introducing View Object*. Obviously, the underlying implementation is different, than this and it is explained in the following paragraph.

The row set invokes `executeQueryForCollection(Object qcObj, Object[] params, int noUserParams)` on `DCDataVO` to read data from the underlying services. `DCDataVO` gets the appropriate data control instance, which manages the connection to the underlying business service implementation and delegates the getter method call (`invokeAccessor()`) to the data control. The framework, by default, uses `DCBeanDataControl` for invoking EJB or Java services. Now you may ask, how does the ADF binding framework know what technology (EJB or Java) is used for building business services? It is the `<AdapterDataControl>` definition present in the `DataControls.dcx` file that differentiates the technologies used for implementing services.

While preparing the request for accessing EJB, the binding layer checks if any `DataControlHandler` class has been specified for the EJB definition present in the data control descriptor XML (`DataControls.dcx`) file. If one is found, the framework routes the request for accessing EJB through the `DataControlHandler` class read from the EJB definition. Here is an example for an EJB-definition entry in `DataControls.dcx`:

```
<ejb-definition ejb-version="3.0"
  ejb-name="HRServiceSessionEJB"
  ejb-type="Session"
  ejb-business-interface="..."
  ejb-interface-type="local"
  initial-context-factory=
  "weblogic.jndi.WLInitialContextFactory"
  DataControlHandler=   "oracle.adf.model.adapter.bean.jpa.
JPQLDataFilterHandler"
  xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
```

When you generate a data control for an EJB session bean, by default, JDeveloper sets `DataControlHandler` to the `oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler` class. The `JPQLDataFilterHandler` class, under the cover, uses a special data collection model (`oracle.adf.model.adapter.bean.provider.JPQLBeanDataCollection`) class for invoking services. When `JPQLBeanDataCollection` is instantiated, it initializes the state and prepares itself for fetching the data collection. As the pagination feature requires a total record

count, the `JPQLBeanDataCollection` fires a count query during this stage. To do this, the `JPQLBeanDataCollection` class prepares the appropriate JPQL count query statement and invokes the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method defined in the service implementation class.

`DCDataVO` caches the retuned `JPQLBeanDataCollection` in the `QueryCollection` object. Once the initialization phase is over, the row set iterator initiates the fetch request for all rows. This request is routed through `QueryCollection`, which reads the cached `JPQLBeanDataCollection` data collection and starts iterating over the collection. While doing so, if the `next()` call does not result in any result and the total number of rows is more than the current rows fetched, `JPQLBeanDataCollection` invokes the `queryByRange(String jpqlStmt, int firstResult, int maxResults)` method in the service implementation and fills the collection with JPA entities returned by the query. For each entity in the collection, the framework creates the `oracle.adf.model.bean.DCDataRow` instance, passing the JPA entity instance as the data provider. When the client refers a value for an attribute in a row, the framework actually reads the value from the data provider entity instance associated with that row. In simple words, framework wraps JBA entity instances returned by the query method using Row objects to be used by the binding layer.

Customizing error handling for EJB services

The ADF Model allows you to customize the default error handling for EJB data control by extending the default `DCErrorHandlerImpl` class. You can use this feature to format the exceptions thrown by the JPA entities. We have discussed the customization of error reporting in *Chapter 11*, *More on Validations and Error Handling* under the topic *Error Handling in ADF*. Refer back to this topic, if you need a quick brush-up on this subject.

Following is an example for a custom exception handler for EJB. `CustomExceptionHandler` used in this example displays only the error message corresponding to the root exception, hiding all child exceptions from the final error display.

```
/**
 * Custom exception handler for EJB. This error handler
 * class hides unwanted details present in the exception
 * when reported to the user
 */
public class CustomExceptionHandler extends DCErrorHandlerImpl {
```

```
public CustomExceptionHandler(boolean b) {
  super(b);
}

public CustomExceptionHandler() {
  super(true);
}

/**
  * This method pre-process exception before displaying it.
  * @param bc
  * @param ex
  */
  @Override
  public void reportException(DCBindingContainer bc,
  Exception ex) {
    ex.printStackTrace();
    //Find the root cause of the exception
    ex = (Exception)getRootCauseException(ex);
    //Following method Call invokes
    //JboException::setAppendCodes(false)
    //recursively on all child exceptions
    disableAppendCodes(ex);
    super.reportException(bc, ex);
  }


  /**
   * Specify appendCode flag for  JboException to false
   * if this exception
   * should not prefix the error message
   * with Product and Message Ids. By default this flag
   * is set to true, This method sets it to false.
   * @param ex
   */
  private void disableAppendCodes(Exception ex) {
    if (ex instanceof JboException) {
      JboException jboEx = (JboException)ex;
      //Set appendCode to false if this exception
      //should not prefix the error
      //message with Product and Message Ids.
      jboEx.setAppendCodes(false);
      Object[] detailExceptions = jboEx.getDetails();
      if ((detailExceptions != null) &&
```

```
        (detailExceptions.length > 0)) {
          for (int z = 0, numEx =
          detailExceptions.length; z < numEx; z++) {
            disableAppendCodes
            ((Exception)detailExceptions[z]);
          }
        }
      }
    }

    /**
    * Get the root exception
    * @param ex
    * @return
    */
    public Throwable getRootCauseException(Throwable ex) {
      Throwable rtEx = null;
      if ((rtEx = ex.getCause()) == null) {
        return ex;
      } else {
      return getRootCauseException(rtEx);
    }

  }

}
```

You must configure the error handler class in the in the `DataBindings.cpx` file of the appropriate view controller project as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.2.60.81" id="DataBindings"
    SeparateXMLFiles="false"
    Package="com.packtpub.adfguide.ch12.view"
    ClientType="Generic"
    ErrorHandlerClass=
    "com.packtpub.adfguide.ch12.view.CustomExceptionHandler" >
 ...
</Application>
```

This chapter is intended to introduce you to the ADF binding support for EJB. To learn more about the visual development support offered by JDeveloper IDE for building EJB applications, refer to *Oracle® Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework*. To access this documentation online, go to `http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html`, navigate to **Oracle JDeveloper** and **ADF Documentation Library** and then select **Java EE Developer's Guide**. Use the search option to find specific topics.

Summary

In this chapter, we looked at the EJB binding support in Oracle ADF. Specifically, we discussed how to use JDeveloper IDE for visually building EJB services. We also discussed how to leverage ADF binding features for declaratively building a UI for EJB.

The next chapter will discuss how to secure a Fusion web application, using ADF Security.