

PREVIEW

Chapter 1

The Quick Learner's Guide to Oracle Fusion Web Application Development

Please Note: All prepublication sample material is for review purposes only.
Content may change without prior notification.

A data control is an implementation of the contract that exists in Oracle ADF between the proprietary APIs of a business service and the consistent set of APIs that are exposed through the ADF model to the web application developer working with the ADF binding layer.

In this first chapter we briefly introduce the technologies of the Oracle Fusion development stack that are used throughout the book in the examples and content. The intention is to provide you with a crash course in Oracle Fusion development that gets you, as a quick learner, started immediately. In detail, this chapter covers

- An overview of Oracle Fusion application development in Oracle JDeveloper 11g
- An architecture overview of the Oracle Application Development Framework
- A quick learner overview of ADF Business Components
- A quick learner introduction to JavaServer Faces
- A quick learner introduction of the ADF Faces Rich Client UI framework

More books about Oracle ADF

For an entry level introduction to Oracle Fusion application development we recommend the purchase of *Oracle JDeveloper 11g Handbook: A Guide to Fusion Web Development* by Duncan Mills, Peter Koletzke and Avrom Roy-Faderman, published in 2009 by McGraw Hill (ISBN-10 0071602380). It is a tutorial driven developer guide that requires no prerequisites on the reader side. Both books, the *Oracle JDeveloper 11g Handbook* and the one you hold in your hand, were written at the same time with the authors making sure only a reasonable overlap to exist.

Another book to recommend for Oracle WebCenter customers is *Oracle WebCenter 11g Handbook: Build Rich, Customizable Enterprise 2.0 Applications* by Frederic Desbiens, Peter Moskovits and Philipp Weckerle. Oracle WebCenter 11g is a powerful Web 2.0 portal framework within Oracle Fusion Middleware that is build on top of Oracle ADF. The book is also published by McGraw Hill (ISBN-10 0071629327).

The Oracle product documentation for ADF and the ADF Faces Rich Client component framework ship in two books, which are available online in HTML and for download in a PDF

format. You can access these books from the **Documentation | JDeveloper** menu entry at otn.oracle.com.

- Oracle® Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11g Release 1
- Oracle® Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g Release 1

Building Oracle Fusion Applications

The Oracle Fusion development stack is the technology choice that Oracle Applications, the group within Oracle that builds Oracle standard business solution based on Java EE and Service Oriented Architecture (SOA), uses to build their next generation of Rich Enterprise Application (REA) software. The Fusion development stack has been chosen with service orientation in mind and includes technologies like ADF Business Components, ADF Faces Rich Client, Data visualization components (DVT), Web Services, BPEL, ESB and XML.

Oracle ADF

In Oracle JDeveloper 11g, the Oracle Application Development Framework has grown out of a series of Java EE frameworks that are integrated through the ADF model layer. The ADF core binding layer is proposed for standardization in JSR-227 “A Standard Data Binding & Data Access Facility for J2EE“, to formalizes the value and method binding of UI components to functionality exposed on Business Services. While JSR-227 is the core of Oracle ADF, the term “ADF” has become a synonym for declarative Java and Java EE development in Oracle JDeveloper. Oracle ADF consists of the following components

- **ADFv** The view layer bindings that exist for ADF include JavaServer Pages with Struts, JavaServer Faces, ADF Faces Rich Client and Apache Trinidad. Using the mobile development support in JavaServer Faces, ADF applications also display on devices like iPhone and PDA.
- **ADFc** The controller component in ADF is an extension of the JavaServer Faces navigation model and promotes modularization and reuse. In addition ADFc provides declarative transaction handling and clearly defined process boundaries.

- **ADFm** The binding layer and model, represented by Data Controls and the binding container object. ADFm is built on JSR-227 and abstracts the view layer model access from the implementation details of the underlying business service.
- **ADFbc** ADF Business Components has a special role within the list of supported business services in that it incorporates the ADF APIs in its Application Module. ADF Business Components is the preferred business service within the Oracle Fusion development stack.
- **ADFdi** Desktop integration with Microsoft Office 2007 that allows developers to access the server side ADF binding layer from Excel workbooks.

Figure 1-1 shows the ADF architecture for the Oracle Fusion developer stack. The view layer is based on JavaServer Faces 1.2 and includes ADF Faces Rich Client components (ADF Faces RC), Data Visualization Tools (DVT), JavaServer Faces mobile support and the desktop integration client, which provides MS Excel access to the ADF binding layer. The controller used with ADF Faces RC and DVT is ADF Task Flow, an enhanced navigation handler built on top of the JavaServer Faces navigation model. Oracle mobile support is based on the Apache Trinidad open source JavaServer Faces components and uses the JavaServer Faces navigation handler as its controller. The role of the controller is to handle page and activity navigation and also to maintain the application state.

The ADF binding layer exposes a consistent API to the user interface layer, hiding the implementation details of the business service the application accesses to read and write data from data services like RDBMS or XML. The binding layer accesses attributes and business methods that are exposed on the business services through ADF Data Controls. Data Controls are business service implementation dependent and map service specific APIs to those understood by ADF. A variety of prefabricated Data Controls exist in ADF that can be used with EJB 3.0/ JPA services, Web Services, POJO services, ADF Business Components and many more. Those relevant to ADF Fusion developers are shown in Figure 1-1

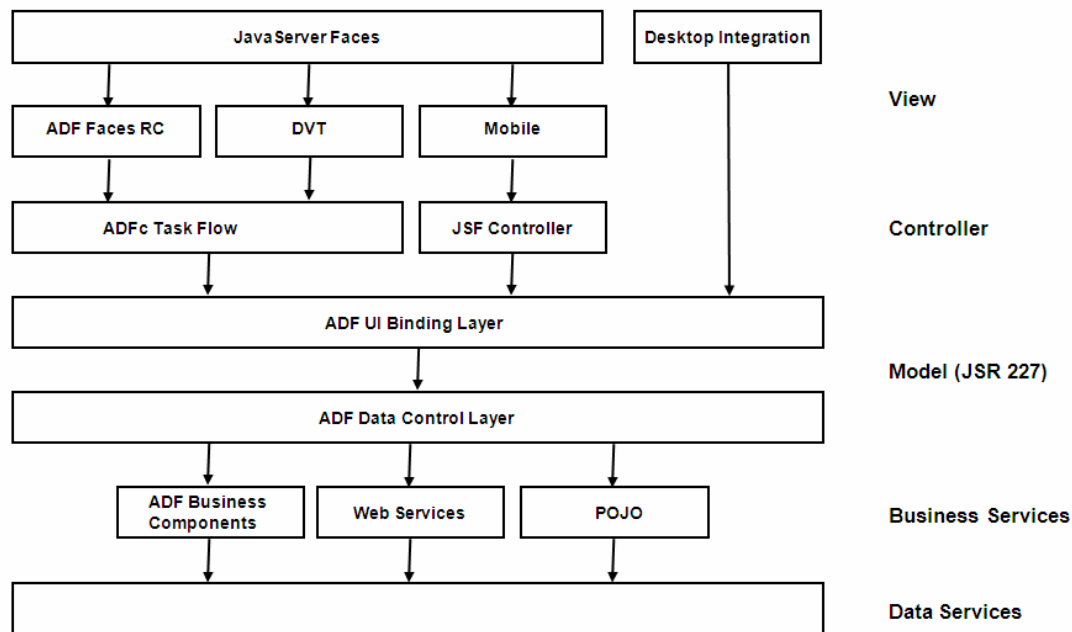


Figure 1-1: ADF Fusion architecture

Building a Fusion application workspace in Oracle JDeveloper

Oracle JDeveloper 11g is the Oracle SOA and Java EE integrated development environment (IDE). To build a Rich Enterprise Application with the Oracle Fusion stack, developers start their new application project with the “Fusion Web Application (ADF)” application template. Assuming a fresh install of Oracle JDeveloper 11g, click on the **New Application** entry in the Application Navigator, or use the **File | New** menu option and select the “Fusion Web Application (ADF)” template from the list of available templates in opened dialog. The template creates the following projects and folders

- **Model** A project that is pre-configured for building the ADF Business Components business service. “Model” is only the default name used and may be changed in the template wizard.
- **ViewController** A project that is pre-configured for using ADF Faces Rich Client and the ADF Controller. “ViewController” is only the default name used and may be changed in the template wizard.
- **.adf\META-INF** A folder created on the file system for the application that contains ADF specific configuration files like “adf-config.xml”. The file content of this directory is

accessible in JDeveloper from the **Descriptors | ADF META-INF** folder in the Application Navigator’s **Application Resources** accordion panel.

- **Src\META-INF** A folder created on the file system for the application that contains deployment descriptor files. The file content of this directory is accessible in JDeveloper from the **Descriptors | META-INF** folder in the Application Navigator’s **Application Resources** accordion panel.

Additional projects, if needed, are created from the **File | New** menu option that opens the “New Gallery” dialog. Select the **General | Projects** node to choose from a list of pre-configured project types.

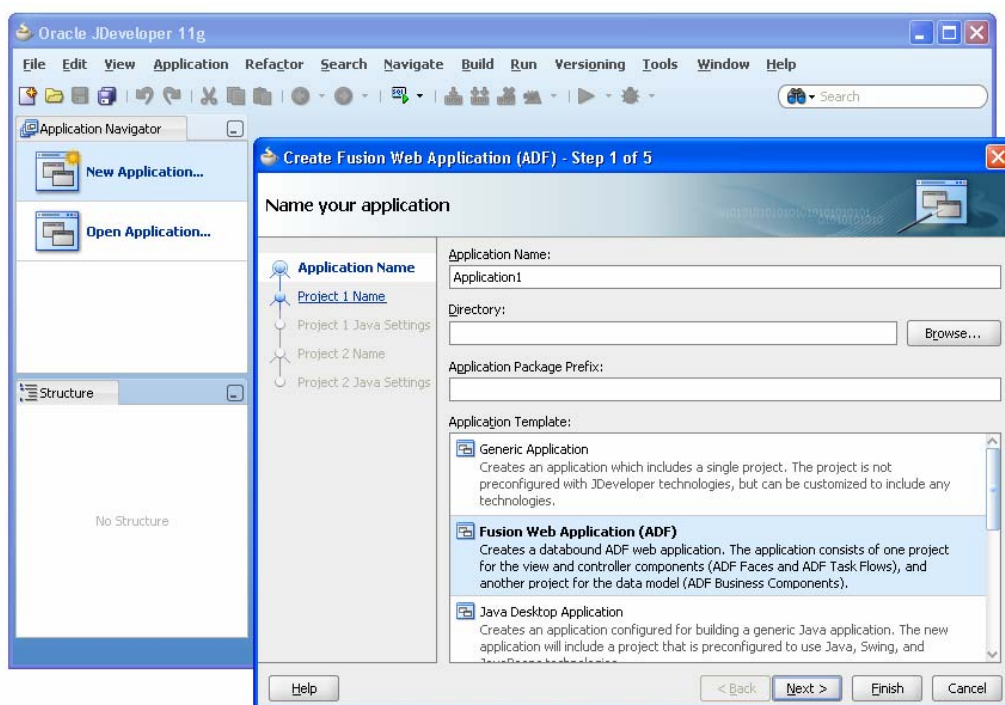


Figure 1-2: Building a new application in Oracle JDeveloper based on the Fusion web application template

ADF Business Components for quick learners

Oracle ADF Business Components is the business services layer of choice in Oracle Fusion application development. Compared to other persistence layer, ADF Business Components provides exceptional on-board intelligence and declarative development gestures that makes it a perfect match for those who seek for an end-to-end 4GL Java EE development architecture.

ADF Business Components provides a foundation of lightweight Java classes and XML metadata configuration files for building database centric business models that include business logic, validation, queries, transaction handling and data access. The XML files are created for each of the ADF BC components, like View Object, Entity Object and Application Module. The name of the configuration file matches to the name of the component. For example, the configuration for an “EmployeesView” View Object is stored in a file called “EmployeesView.xml”. If custom business logic is required then additional Java classes can be created that extend the component base framework class.

The ADF Business Components framework does not create user interfaces and purely focuses on the backend application logic and data management. It provides options for developers to define UI hints and properties that the web application developer uses to display labels, tool tips and data formats. Using the Oracle ADF binding layer, ADF BC client APIs are exposed on the ADF data control palette, a hierarchical view of the data model objects and functionality, in Oracle JDeveloper 11g. Figure 01-3 shows the core ADF BC elements that developers work with when building Fusion web applications and their relation to each other. The core elements include

- **Application Module** One or more modules that expose the defined data model based on the defined View Objects and their relations. The root Application Module represents the transaction context on which commit and rollback operations are executed.
- **View Object** Definition of the data access layer that provides data based on SQL queries, programmatic Java and stored procedure access and static lists. SQL query based View Object reference one or many entity objects to perform DML operations.
- **Entity Object** Java object that represents a row in a database table. It exposes method for DML and the object lifecycle like create and remove.
- **Associations** Entity object associations define the relation between two entity objects. The association can be read from define database constraints or manually defined.
- **ViewLinks** Relations that are based on entity associations or defined manually, providing master-detail coordination between those View Objects.

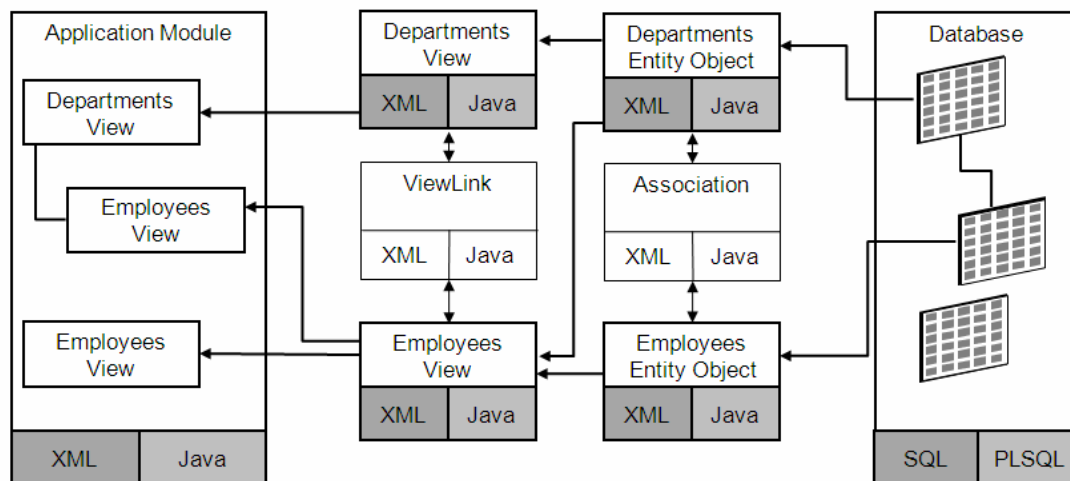


Figure 1-3: ADF Business Components component architecture

Note: In depth coverage of ADF Business Components is subject of the “Oracle JDeveloper 11g Handbook: A Guide to Fusion Web Development”, published by McGraw Hill, and the “Oracle® Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g Release 1” available online at otn.oracle.com

Maturity through history

Oracle ADF Business Components is a road tested Java EE persistence framework that has existed within Oracle JDeveloper since 1999. Originally named Oracle Java Business Objects (JBO), and still showing this legacy in the “oracle.jbo” package naming, Oracle ADF Business Components is used by Oracle internal and external customers from the early days on. The Oracle ADF Business Components framework gained importance with its adoption in the Oracle Applications developer stack, where since then it is used as the model layer for Java based commercial self service applications. Today, when building Oracle Fusion applications, Oracle Applications uses the same version of Oracle ADF Business Components that ship with Oracle JDeveloper and the Oracle Fusion Middleware, which demonstrates Oracle confidence in the quality of the stack and improved productivity features. Basing big parts of its own business on the ADF Business Component technology, Oracle adds even more road testing to this software layer, eating more of its own dog food than probably any other vendor or open source project.

Framework classes

Default ADF Business Component implementation base classes exist for all component interfaces in the oracle.jbo.server package. The base classes are referenced in the metadata that is generated for the created data source objects in an application. Application developers can extend the default framework classes to expose additional helper methods or to customize the default framework behavior. Extended framework classes are configured in Oracle JDeveloper either on the project level, in which case only new objects created in this single project will be based on the custom classes, or on the IDE level, in which case any ADF Business Component project uses the custom classes.

Project level configuration

To configure custom framework classes for a project, select the project root folder in the Oracle JDeveloper Application Navigator view and choose **Project Properties** from the context menu. In the project properties editor, expand the **Business Components** entry and select the **Base Classes** entry. Use this dialog to replace the listed base classes with the custom sub classes.

IDE level configuration

To configure custom framework classes for all ADF Business Components projects that created in an IDE, choose **Tools | Preferences** from the menu and expand the **Business Components** node. Select the **Base Classes** entry and replace the listed base classes with the custom sub classes.

It is worthwhile to explore the various options that exist in the tool preferences to refine the ADF Business Components settings. For example, the **Packages** entry should be used to locate the different class types in their own sub package folder for better readability.

Creating Business Components model

Starting from a model project created by the “Fusion Web Application (ADF)” template, choose the **New** option from the context menu to open the Oracle JDeveloper “New Gallery”. Expand the **Business Tier** node and select the **Business Components** entry. As shown in figure 1-4, Oracle JDeveloper 11g provides various options to create business

component objects. Selecting one of the options expands the item area, displaying the item description.

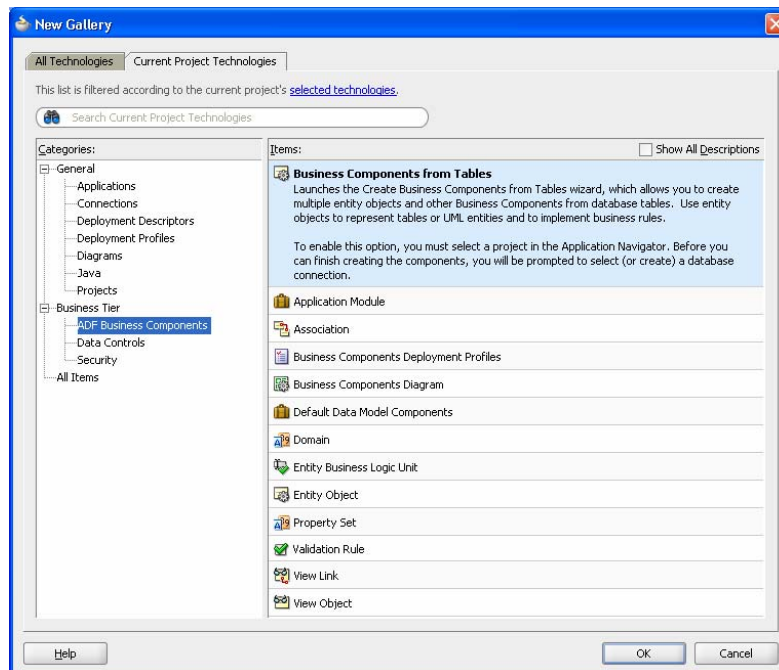


Figure 1-4: ADF Business Component creation options in the Oracle JDeveloper New Gallery

Select the **Create Business Components from Tables** entry and press Ok. In the first dialog, create a database connection to the RDBM instance and schema that holds the application database tables, views and synonyms. The database connection is referenced in the ADF Business Components Application Module and defined for the application. To change the database connection properties, you can open the “Application Resources” accordion panel in the Application Navigator and expand the “Database” node. Use the **Properties** option in the context menu you open with the right mouse button to edit the connect information.,

Note: The database connection used by the ADF Business Component project is also accessible from the model project. Open the project properties by selecting **Project Properties** from the context menu and choose the **Business Components** entry. Press the green “plus” icon to create a new database connection, or the “pencil” icon to edit the existing database connection.

Creating Entities

In the first wizard page, query the table, views and synonyms to build entity object representations for. By default, the entity object name is chosen the same as the selected database object, but can be changed using the **Entity Name** field. Entity associations are automatically created for database objects that have a foreign key relationship defined. The database object name is transformed in a Java friendly format, which is to use camel case letters. For example, a table name of "PERSON_DETAIL" creates an entity name of "PersonDetail".

Creating View Objects

The second page in the wizard shows the list of created Entity Objects to build default View Objects for. The names of the View Object can be changed using the **Object Name** field. View Links are automatically created for View Objects that are based on entities with defined associations.

Step 3 of the Business Components wizard creates read-only-view objects that are not based on an Entity Object.

Creating an Application Module instance

The fourth wizard page allows application developers to associate the created View Objects with an existing or new Application Module instance that exposes the View Objects as a part of its data model.

ADF Business Component diagram

The last, optional, step of the creation wizard allows developers to create a diagram for further ADF Business Component modeling that includes the created objects.

Entity Objects

An Entity Object (EO) provides an object-oriented representation of the data it caches from the database, such as a table or view. It exposes access methods for its attributes that correspond to table or view row columns and maps the database SQL types to standard Java types or ADF Business Component domain types. In addition it holds domain business logic for consistent validation of business policies and rules. An entity object holds a copy of the data object queried from the database, as well as a local copy to work with. The ADF Business

Component base class that implements the default entity behavior is `EntityImpl` and is referenced in the generated entity metadata of an application entity, like "Employees".

Creating entity objects

Entity objects are created in a "bulk" operation with the "Create Business Components from Tables" wizard explained earlier or individually by selecting the **Entity Object** entry in the **Business Tier | ADF Business Components** New Gallery section. The **Create Entity Object** wizard steps the developer through the creation and configuration of the new entity

- **Name** Developer option to specify the name and Java package for the new entity, the base entity if any, shared property sets to use and the data source as database schema or ADF Business Component service interface
- **Attributes** Developer option to create additional entity attributes or remove existing attributes from entity
- **Attribute Settings** Provides options to configure each attribute with query, update and refresh flags, to change the attribute data type, for example to use `DBSequence` if the field value is added by a database trigger on insert.
- **Java** Allows developers to generate Java implementation classes that extend the default framework classes. If there is no immediate need for such a Java class, for example to override the default entity methods, like the `beforeCommit` and `beforeRollback` methods, then the class creation can be deferred to later.
- **Generate** Allows developers to create a default View Object and configure the View Object with an Application Module

Editing entity objects

Existing entity objects are edited in Oracle JDeveloper editors and the Property Inspector [ctrl+shift+i]. Open the editor by double click on the entity entry in the Application Navigator, or choose **Open <Entity>** from the context menu. The editor shows a categorized view of the entity object configuration options. You use the icons on top of the editor dialog to create, remove or edit configuration options. In addition, for each selected configuration options, the Property Inspector shows additional options. The editor window and the Property Inspector window are synchronized when both are open.

<Entity>Impl

Business Service developers can build a custom implementation class for an entity that extends the configured base framework class and that is referenced from the entity metadata. The entity implementation class allows developers to expose typed attribute setter and getter methods, create attribute validation methods, access child collections in a master-detail relation, retrieve the entity object key or to modify the entity create and DML behavior. All changes to the default framework behavior applied in an <Entity>Impl class apply only to the entity the class is built for.

To create a custom entity implementation class, select the Entity object in the Application Navigator view of the ADF Business Component project and choose **Open <Entity>** from the context menu. In the Entity editor, select the **Java** menu option and click the “pencil” icon on top. This opens the **Java Options** dialog to create entity related Java classes, including an option to create the implementation class. For a list of public methods that can be overridden, choose **Source | Override Methods** from the opened Java class in the Oracle JDeveloper code view.

Other custom entity classes that can be created through the **Java Options** dialog are

- **<Entity>DefImpl** the entity definition object that describes the structure of the entity at runtime and that manages the instances of the entity object it describes. The DefImpl class is a singleton and changes applied at runtime to this object are reflected by all instance. The class exposes methods to create new entity instances, find existing instances by their primary key and to access entity properties.
- **<Entity>CollImpl** extends the EntityCache object that caches queried rows for a particular entity type. The entity cache is referenced from View Object queries and improves performance in cases in which multiple views access the same entity object.

Associations

An Association describes the dependency that exists between two entities. When creating an Association between two Entity Objects, you establish an active link between them. The active link is accessible from your Java code to work with the dependent entity. An association can be created from a foreign key constraint in the database or manually at design time. It is defined in XML metadata that is referenced by an instance of the

`EntityAssociation` class at runtime and maps attributes of two entities in a where clause. To edit an existing association, double click the association in the JDeveloper Application Navigator to open the editor view.

Creating associations

Associations are automatically created for dependent entity objects when running the “Create Business Components from Tables” wizard. To manually create associations, right mouse click anywhere in the project package structure and choose **New Association** from the context menu. Alternatively press the ctrl+N key pair to open the Oracle JDeveloper “New Gallery” and expand the **Business Tier** node to select the **Business Components** entry. In here choose the **Association** entry. In the association wizard, select the two entity attributes to link together and press the **Add** button. In the following, define the accessor names to be added to the source and target entity for programmatic access.

Compositions

A composition is defined as an attribute of the association and describes a relationship in that the parent entity doesn't reference the child entity but contains it. For example, using compositions, creating or updating a child entity always ensures that a parent entity exists and that it is updated before its children. The relationship between an order and an order item is a good example for a composition. An order item cannot exist without a valid order and deleting an order must cascade delete the contained order items. If the parent entity is created at the same time as the detail item, then no primary key may exist yet for the detail to reference. If the primary key becomes available after commit of the parent, then all foreign key references of the order items entity are updated with this key.

View Objects

The hierarchy of View Objects represents the business service data structure and the business logic that is contained in the entity objects to the application client. It exposes a business service client interface to the user interface developer to bind UI components to. In general terms, a View Object manages the client data for display and update operations. View Objects that are based on entities provide DML access to the application client, whereas a non-entity View Object is read only. Other data access options for a View Object are

programmatically, in which case Java is used to read in the data, and static, in which case the View Object shows a static list of values like imported from a character delimited file.

A View Object can be linked with another View Object to describe a master-detail relationship that at runtime is synchronized by the ADF Business Components framework.

View Objects use SQL to query the data from the database. Developers can modify the default View Object query by adding a where clause, group by clause or adding bind variables that allow a query by example.

Creating a View Object

To create a View Object, use the “Create Business Components from Tables” wizard or one of the following options

- **Create View Object for an entity** Select an entity object in the Oracle JDeveloper Application Navigator and choose **New Default View Object** from the context menu
- **Create a View Object from the New Gallery** Use **File | New** or **ctrl+N** to open the New Gallery in Oracle JDeveloper. Select the **View Object** entry in the **Business Tier | ADF Business Components** node
- **Create a View Object anywhere in the project** Select a package node in the project and choose **New View Object** from the context menu.

The **Create View Object** wizard steps the developer through the creation and configuration of the new entity

- **Name** The name category allows developers to define the View Object name and package location in where the XML definition files and optional generated Java classes are located. In addition, developers choose the type of View Object as SQL based, programmatic or static
- **Entity Objects** SQL query type View Object can be based on one or many entity objects. If a View Object is based on many entities then a link is created based on existing entity associations, using inner or outer joins to display the data.
- **Attributes** Displays the available list of attributes exposed by the View Object entity or entities for the developer to choose from. Additionally, the developer may create transient

attribute that don't have an equivalent in the entity object. Such attributes are used for example to display calculated fields

- **Attribute Settings** Each exposed attribute can be configured for its update and refresh behavior, if it is queryable and the default value it should display. The default value may be chosen as a static literal, or a Groovy expression.
- **Query** The SQL query that is used to retrieve the result set. The possible SQL modes are "Normal", for queries that are created by the framework based on the selected entities and objects, "Expert" for queries that are fully hand edited by the application developer, and "Declarative" for queries that are defined by the framework and that are further filtered through the use of declarative query builders
- **Bind variables** To dynamically filter the returned result set, developers can use Oracle style named bind variables to pass the condition for the query clause. Bind variables are exposed in the "ExecuteWithParams" operation of the View Object but can also be set using the View Object API.
- **Java** Like Entity Objects, View Objects are defined in XML that at runtime is mapped to a framework class. Custom implementations of the runtime classes can be created for the View Object and for the View Object rows to expose typed user interfaces for the contained attributes, to change the default framework behavior and to provide default values in Java.
- **Application Module** View Objects are exposed on an Application Module that developers can create or reference in the last wizard dialog.

<View Object>Impl

Developers can build a custom implementation class for a View Object that extends the framework `ViewObjectImpl` base class. A custom View Object implementation class allows developers to override or hook into the default framework functionality.

To create a custom View Object implementation class, select the View Object in the Application Navigator view of the ADF Business Component project and choose **Open <View Object>** from the context menu. In the View Object editor dialog, select the **Java** menu option and click the "pencil" icon on top. This opens the **Java Options** dialog to create an application specific View Object implementation class. For a list of public methods to override in the

ViewObjectImpl class, choose **Source | Override Methods** from the opened Java class in the Oracle JDeveloper code view.

Other custom entity classes that can be created through the **Java Options** dialog are

- **<ViewObject>DefImpl** a custom runtime object that describes the View Object structure based on the XML metadata defined at design time. All custom implementations extend the ViewDefImpl base class.
- **<ViewObject>RowImpl** an object that provides access to a View Object row and that extends the ViewRowImpl framework class. ADF Business Components instantiates one object of this class for each record that is returned by a View Object query. The RowImpl class allows developers to access row data and their child collections.

Named View Criteria

Named View Criteria are pre-defined and reusable where clause definitions that are exposed for declarative use in the ADF DataControl. Developers use named View Criteria to build search forms, to expose filtered View Object instances on the Application Module and to apply them programmatically at runtime to restrict the returned result set.

To create a named View Criteria, in the Oracle JDeveloper Application Navigator, select the View Object to own the criteria and open it in the editor. Choose the **Query** criteria and press the green plus icon to open the **Create View Criteria** dialog. A View Criteria can exist of several groups, which are criteria rows that are AND'ed or OR'ed together. To create a new criteria within a group, press **Add Item**. Select a View Object attribute and an operator, as well as whether the where clause should be based on a literal value or a bind variable. If the bind variable doesn't exist, it can be created within the same dialog. After the View Criteria is created, it shows in the ADF DataControl palette under the View Object node's "Named Criteria" entry.

Operations

View Objects expose a set of operations to the client application to navigate within the exposed collection, to execute the query, to create and delete rows and to search within the row set. All operations are also exposed on the ADF binding layer for developers to declaratively bind UI components to. The list below describes some of the commonly used operations

- **Create / CreateInsert** Create a new row in the View Object. The “CreateInsert” operation also adds the new row to the current transaction, after which it is initialized. Using the “Create” operation only creates the row but doesn't add it to the transaction.
- **Create with Parameters** Create a new row with default values declaratively set for some of the View Object attributes. Adding this operation from the ADF Data Control palette to a page, creates a operation binding entry that developers can add “NamedData” items to that reference a ViewObject attribute name and a source for the value. A usecase for this operation is to create a copy of a selected row.
- **Delete** Removes the current selected row from the collection.
- **Find** Calling this operation sets the View Object into query mode, also known as “Find mode”. All View Object attributes take search criteria values so that after a call to “Execute”, the result set is filtered. The “Find” operation has been used in previous versions of ADF to build search forms but lost its importance in Oracle JDeveloper 11g with the introduction of the ADF Faces RC `af:query` component and the recommended use of named View Criteria.
- **SetCurrentRowWithKey** Makes the row current that is defined by its String serialized row key.
- **RemoveRowWithKey** Removes the row that is identified by its serialized row key.

Client methods

It is best practices when working with ADF Business Components to not type cast to a business service implementation class on the client layer. So instead of type casting the handle to a View Object to its `ViewObjectImpl` class, you should generate and use the client interface with the public method defined. For example, lets assume a public method `printHelloWorld(String s)` to exist in the `DepartmentsViewImpl` class of the `DepartmensView` object. To create the client interface class and expose the method for the application client to use, select the View Object in the Application Navigator and choose **Open DepartmentsView** from the context menu. In the opened editor view, select the Java category and click on the “pencil” icon next to the **Client Interface** section. In the opened **Edit Client Interface** dialog, select the “printHelloWorld” method entry and move it into the selection list. This creates the

client interface and exposes the method on it. The type casting on the user interface to execute the method now is to “DepartmentsView” and no longer the implementation class.

Bind variables

Bind variables are used in View Criteria and the where clause of a View Object query. They are named value holders that developers can access from the client to provide the filter criteria. Bind variables can be created when creating the View Criteria and View Object, but can also be created later using the View Object editor.

View Links

View Links, like associations, define the relationship between two View Objects. They are created through manual mapping of a View Object attribute to the equivalent attribute of a dependent View Object or based on an existing entity association.

Creating View Links

View Links are automatically created for dependent View Objects when running the “Create Business Components from Tables” wizard. To manually create a View Link, right mouse click anywhere in the project package structure and choose **New View Link** from the context menu. Alternatively press the ctrl+N key pair to open the Oracle JDeveloper “New Gallery” and expand the **Business Tier** node to select the **Business Components** entry. In here choose the **View Link** entry. In the View Link wizard, select the two entity attributes to link together and press the **Add** button, or use existing entity associations. In the following dialogs, optionally define accessors methods to be generated in the View Object implementation classes, modify the generated query and configure the View Link to display in an Application Module.

Application Modules

An Application Module is a container for View Objects, View Links and other Application Modules. It is defined in XML metadata at design time and encapsulates the Business Components data model and methods. As shown in figure 1-5, the root Application Module is the root of the data model and provides the transaction context for all the objects contained in it. View Objects are added as instances to the data model with a hierarchical structure built based on existing View Links. Application Modules are exposed as a data control in the Data Controls panel.

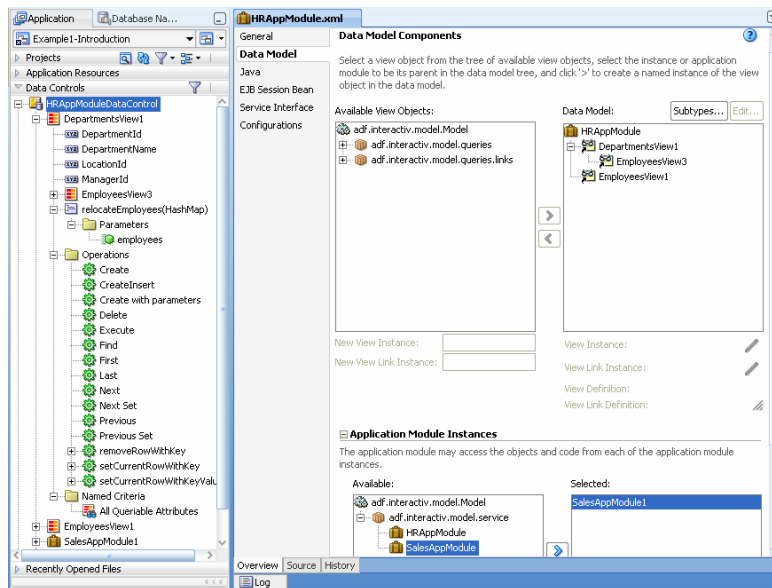


Figure 1-5: ADF Business Components data model and its exposure in the ADF Data Control palette

<Application Module>Impl

Developers build a custom implementation class for an Application Module to easier access View Object instances and to expose public client methods. The Application Module implementation extends the `ApplicationModuleImpl` framework class and is referenced from the Application Module XML metadata.

To create a custom Application Module implementation class, select the Application Module in the Application Navigator view of the ADF Business Component project and choose **Open <Application Module>** from the context menu. In the Entity editor dialog, select the **Java** menu option and click the “pencil” icon on top. This opens the **Java Options** dialog to create Application Module specific Java classes, including an option to create the implementation class. For a list of public methods that can be overridden, choose **Source | Override Methods** from the opened Java class in the Oracle JDeveloper code view.

Other custom classes that can be created through the **Java Options** dialog are

- **<Application Module>DefImpl** a custom object that stores the runtime metadata for the Application Modules. The object is needed to load an Application Module and the components it contains from XML metadata. If no custom object is created, then the framework uses the `ApplicationModuleDefImpl` base implementation class

Defining the Data Model

Open the Application Module by double clicking on its entry in the Application Navigator. Select the **Data Model** category to build the data model from the list of “Available View Objects”. Selecting a View Object from the list of available View Objects and adding it to the “Data Model” creates a new instance of the View Object, with an added number to the View Object name. Selecting the View Object instance and pressing the **Edit** button allows to assign a named view criteria to the instance, assuming a view criteria is defined for this View Object. As shown in figure 01-06, the View Object instance hierarchy of the data model displays in the Data Controls palette with all the View Object operations and client methods defined for the View Object.

Root Application Modules may have nested Application Modules configured that share the parent module transaction context.

Building client interface methods

Similar to client interfaces that can be exposed on View Objects, Application Modules can also expose public methods to the client. To write and expose a public method, you first create the Application Module implementation class. Open the Application Module in the editor and select the **Java** category. Click the “pencil” icon to open the **Select Java Options** dialog. Check the option that creates the Application Module impl class and close the dialog. Open the Application Module implementation class and add the public method to expose to the client. In the Application Module editor, click the “pencil” icon next to the **Client Interface** section. In the **Edit Client Interface** dialog, move the public method entry to the selected list to create the interface.

Testing

ADF Business Components services can be tested using the integrated generic Java client. To launch the tester, select the Application Module in the JDeveloper Application Navigator and choose **Run** from the context menu.

Oracle ADF for quick learners

The ADF model exists of two layers, data controls and bindings. Data controls abstract the implementation of a business service and map the service attributes and methods to consistent access patterns in ADF. Data controls are not used directly from the application view

layer or controller and instead accessed by Oracle JDeveloper at design time and the ADF binding layer at runtime. The ADF binding layer is also referred to as the “data binding” layer and exposes methods, attributes and query result sets that exist on the business services layer, and that are published through the data control interface, to the view layer. The binding layer shapes the data control access to the functionality needed on the current displayed page.

Maturity through history

A common safe harbor thesis in software development is to wait for a second version of a Java EE or open source framework before adopting it. Before Oracle ADF, Oracle provided a Swing and JavaServer Pages binding for Oracle ADF BC so applications developers had it easy to build desktop and web applications in Java and Java EE. In early 2004, Oracle released Oracle ADF, as a generic binding layer to link UIX, Swing and JavaServer Pages interfaces to their Java EE models, which included Enterprise Java Beans (EJB), Plain Old Java Objects (POJO) and Web Services. The Oracle JDeveloper version for this initial release was Oracle JDeveloper 9.0.5. Since then, a new version of the ADF binding has been released with each version of Oracle JDeveloper. Therefore, from in-official counting, Oracle ADF in Oracle JDeveloper 11g is the fourth version of this technology. Since 2004, the Oracle ADF developer community has constantly grown, a proof point for the value add and maturity of this technology. Also grown has the support for business service models and view layer technologies, which now includes JavaServer Faces, mobile and MS Excel.

Introduction to Data Controls

A data control is an implementation of the contract that exists in Oracle ADF between the proprietary APIs of a business service and the consistent set of APIs that are exposed through the ADF model to the web application developer working with the ADF binding layer (figure 01-6). As a web application developer you don't need to know about the classes that comprise a data control as you are exposed to the ADF binding layer only. Knowledge about the data control internals is important for data control developers and to some degree for business service developers that expose a service as a data control.

The `oracle.adf.model.binding.DCDataControl` class is the abstract base class in ADF that infrastructure developers extend to build specific business service data controls for the Oracle ADF model. Implementations of the `DCDataControl` class manage

connections to data providers, handle exceptions raised by the business service and manage the iterator binding to RowIterators in the data provider.

ADF Data Control functionality

Dependent on the business service to map the ADF binding layer to, the ADF model provides the functionality explained below

- **Iterator service** ADF provides an implementation of the iterator pattern that can be used with any business collection. Data controls optionally can use built-in operations for the implementation, like first, next, previous, last in ADF BC to navigate the iterator.
- **Find** The ADF model provides functionality to set a business service into a query by example (QBE) mode.
- **Business object properties access** The ADF model allows access to properties of the business object through Java bean setter and getter methods.
- **Method invocation service** Business service operations and methods are exposed through the data control for the user interface developer to invoke through the binding layer. Iterator may reference the result set of such a method invocation to display the returned data in a collection model used for example with tables and trees. Methods referenced by an iterator are implicitly invoked by the ADF model.
- **Transaction services** ADF does not handle transactions but notifies the business service about logical transaction events like commit and rollback. Custom data controls may respond to the default commit and rollback operation that are exposed on the data control palette, by calling the equivalent built-in functionality on the business service. Not all data controls support this functionality though.
- **Collection manipulation** The ADF model provides default operations to work with the collection exposed by an iterator. Application developers can use these operations to add, modify and remove objects from a collection. The Data control's responsibility is to notify the business service about these changes if an equivalent operation is available.
- **Lifecycle notification** Data controls receive lifecycle event notification from ADF, for the developer to passivate or activate business service user state.

The above functionality, and more, is exposed through the Oracle ADF model for declarative use. The data control developer and the business service technology that is used determine how complete the mapping of the standard ADF services to functionality of the business service is.

ADF Business Components Data Control

The ADF Business Component data control extends `DCDataControl` and is implemented by the `oracle.adf.model.bc4j.DCJboDataControl` class. It exposes ADF Business Component root Application Modules as data controls in the ADF Data Control palette. Nested Application Modules, View Object instances, View Object attributes and operations are exposed in a hierarchical structure to the business application developer.

Oracle Fusion developers use ADF Business Components as the business service and don't need to bother how to expose the service as a data control because Application Modules automatically show in the Oracle JDeveloper 11g data control panel. ADF BC business service developers control the functionality that is exposed by the data control through configuration in the ADF Business Component **Data Model** panel of the Application Module. At runtime, the ADF Business Component data control, `DCJboDataControl`, provides an Application Module instance for each application request. From an Application Module perspective, the data control serves as the client that interacts with the business service. All exceptions raised by the ADF Business Component framework that aren't handled in the business service itself are passed on to the data control, which forwards them to the ADF error handler.

To work with the ADF Business Components data control in a view layer project, ensure a project dependency to the model project exists in the view project properties. Double click the view project node and select the **Dependencies** option. If the model project doesn't show in the list of dependencies, then click the "pencil" icon to build it. If the ADF Business Component data control doesn't show in the Data Control panel of the Application Navigator, use the right mouse context menu to refresh the panel.

Creating Data Controls for non ADF BC services

For business services like EJB, web service or POJO, application developers need to explicitly create the data control metadata definition files to be used with the default data control implementation classes in ADF. To do so, they select the EJB session bean, the Web

Service or the POJO in the Oracle JDeveloper Application Navigator and choose **Create Data Control** from the context menu.

An alternative option exists for web services, which is to select **File | New** from the Oracle JDeveloper menu and then choose **All Technologies | Business Tier | Web Services** to launch the “Web Service Data Control” wizard to provide the online or local reference to the service WSDL file.

The DataControl definition is stored as XML in the DataControls.dcx file that maps the generic data control implementation class to the session façade or service endpoint. Default data control implementations exist in ADF for the standard business services mentioned above. The generated data control metadata describe the entities, the attributes and methods of the referenced service instance that ADF should work with.

Note: The web service data control allows invoking any Web service of either SOAP runtime or WSIF. The web service can be invoked purely from the data described in the metadata file at design time

The Oracle JDeveloper Data Control palette

The Data Control palette in Oracle JDeveloper 11g exposes the ADF Business Component Data Model to the application developer for declarative use.

As shown in figure 1-5, the following information is exposed to the application developer

- **View Objects** are collections of row objects that developers work with to build input forms, tables, trees, tree tables and navigation controls. A View Object that has a view link defined to a child collection shows the detail View Object in its expanded view hierarchy. The “DepartmentsView1” collection instance in figure 01-5 has a detail View Object instance “EmployeesView3” defined. Dragging both collections to an ADF Faces page, say as a form and table, creates a master-detail behavior in which the table is refreshed with the change of the row currency in the form.
- **View Object attributes** are exposed in the expanded View Object hierarchy and can be dragged onto a page as input components, output components and select choice components. A common usecase for dragging an attribute to a page is to replace a form text input field, which you delete from the form, with a select list component.

- **Operations** are built-in methods available on the Application module and the View Object. Operations are added as command actions to a page. Operations, like *setCurrentRowWithKey*, that require an input argument display a dialog to the application developer to provide values or value references.
- **Methods** exposed on the ADF Business Component client interface of a View Object or Application Module are dragged as command components or input forms. The method arguments are exposed as attributes that can be dragged as UI input components to a page if the argument type is a simple object. If the argument type is a complex object then a managed bean property is required that returns the complex object.
- **View Criteria** are named query predicates for the View Object they are exposed under. Dragging a View Criteria on a page allows to declaratively build search forms.
- **Application Modules** show as root nodes in the Data Control palette, or nested controls for Application Module instances that are nested in a parent instance from which they inherit the transaction context. Figure 1-5 shows an instance of “SalesAppModule” as a nested Application Module.

What happens when you drag and drop content from the Data Control palette

Dragging an entry of the Oracle JDeveloper data control palette to an ADF Faces page, leads to the following

- A context menu option is shown for the application developer to select a UI component to which the selected entry should be data bound
- If this is the first ADF element that is added to the view project, then the `web.xml` configuration file is updated with filter references to the `ADFBindingFilter` and Oracle Data Visualization Tools (DVT) filters that are used to graphically visualize data on a page. In addition, the `DataBindings.cpx` registry file is created.
- If this is the first ADF bounded component added to the JavaServer Faces page, then a metadata file is created to hold the ADF binding information for this page. The metadata file name is defined as the name of the JSF page followed by “PageDef.xml”.
- The ADF metadata file is updated with a reference to the collection, attribute or operation added to the page

- Expression Language references are created to link the ADF metadata in the page specific ADF binding file to the user interface component.

Note: The PageDef metadata file is also referenced to as “ADF binding definition” and “page definition”.

The role of the ADF binding layer

Oracle ADF data controls expose attributes and functionality of the business services for the web application developer to bind user interface components to. The components however don't bind directly to the data control, but, as shown in figure 1-6, to the ADF binding layer. The ADF binding layer consists of metadata that is generated for each ADF enabled page and a set of generic framework classes. The ADF UI binding layer exposes the information of the data controls configured in the binding context. The ADF binding layer exposes a consistent API that is independent of the technology used to build the business service, for the application developer to build the user interface.

The Oracle Application Development Framework is designed for declarative and visual application development. Fusion web applications that use ADF Faces Rich Client and Oracle DVT components for rendering the user interface bind the UI component properties to the ADF binding through Expression Language, an easy to learn and use scripting notation introduced later in this chapter.

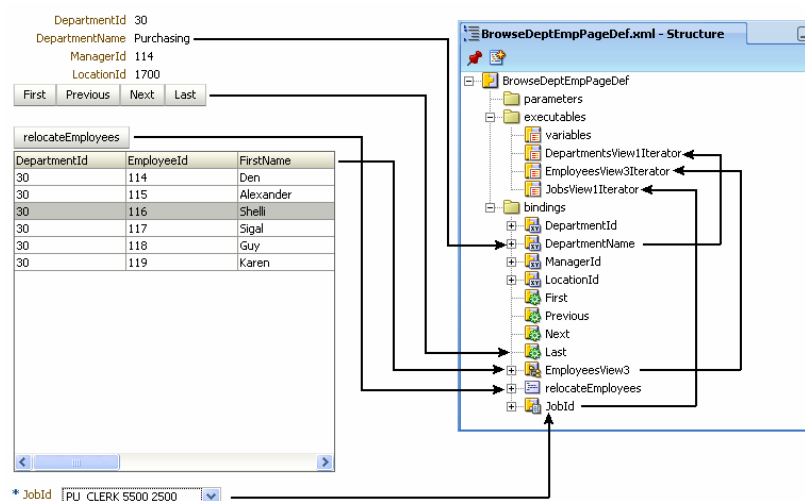


Figure 1-6: ADF binding metadata in Oracle JDeveloper Structure window linked to UI components on the page

Figure 1-6 shows the ADF binding entries for a master-detail page that renders the master view as a form and the detail as a table. As shown in the image, user interface components, like table or select list, are linked to entries in the binding page definition. Each entry in the **bindings** category holds a reference to an entry in the **executables** category. An iterator in the executables category is a reference to a RowIterator exposed on the data control. The “relocateEmployees” button is bound to the “relocateEmployees” method binding through Expression Language that invokes the method on the binding layer when the button is pressed.

```
{bindings.relocateEmployees.execute}
```

In addition to the default functionality that is configured when dragging an entry from the data control palette to the page, developers may need to access the binding definitions from Java, which is what the next sections will cover.

Programming against the ADF binding layer

All binding metadata that you work with at designtime, are used to configure a framework object instance at runtime. Figure 01-7 shows the hierarchy of the core ADF binding classes from the perspective of the ADF Faces Rich client view layer. ADF Faces RC components access the ADF binding layer through specialist binding classes that work as an abstraction between the UI component model APIs and the generic ADF binding classes they extend. As shown in Figure 1-7, ADF Faces specific binding classes have a “FacesCtrl” naming prefix and extend from the generic binding layer classes, which are the classes that developers work with when accessing the ADF binding from Java. The role of the ADF Faces specific binding classes is to map the ADF binding API to the interface expected by the UI component model and to expose functionality contained in the binding to Expression Language access.

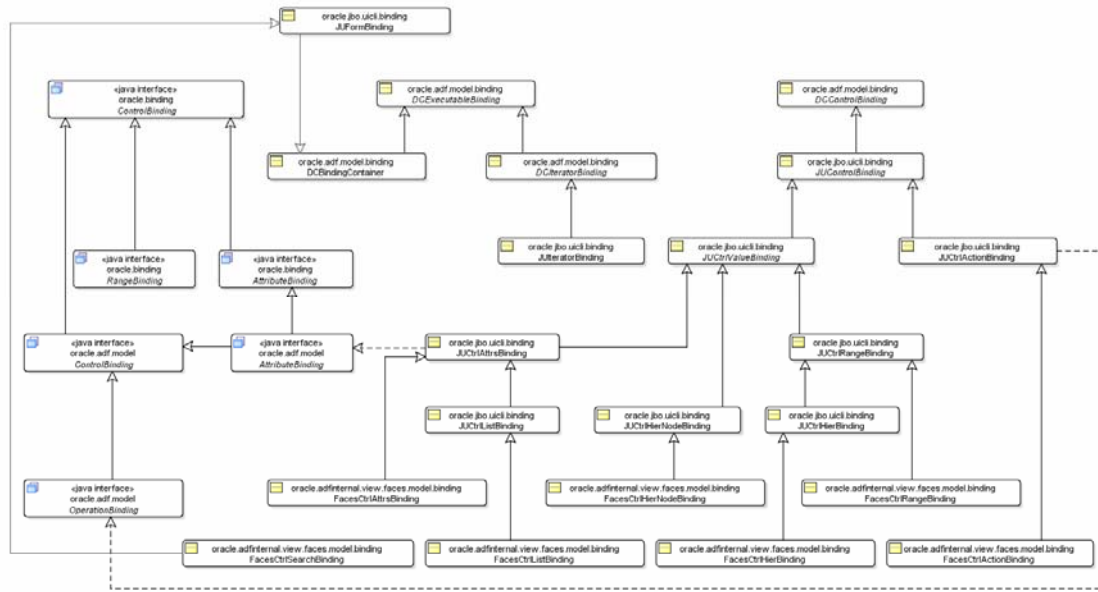


Figure 1-7: ADF binding class diagram

Note: Fusion application developers that work declaratively with ADF and ADF Faces are not directly exposed to these framework classes but use them implicitly in the Expression Language references that are created in Oracle JDeveloper. To know about the binding classes and how to use them is only important for developers that want to access the binding layer from Java. Sooner or later this includes all of us.

The framework classes introduced in the following are the most commonly used classes when accessing the ADF binding layer from Java.

Note: The ADF Faces component framework and the role of the Faces binding classes are discussed in the ADF Faces Rich Client section later in this chapter

DCControlBinding

The `oracle.adf.model.binding.DCControlBinding` class is the base class that is extended by all component binding classes. It defines access to the current `RowIterator`, the `BindingContainer` and the `DataProvider` instance. A control binding can be an attribute binding, like “DepartmentName” in figure 1-6 or a tree binding like “EmployeesView3”.

BindingContainer and DCBindingContainer

In figure 1-6, the “BrowseDeptEmpPageDef” root container is exposed by the “bindings” object at runtime. Different options exist to access the “bindings” object at runtime and below is the option we consider best practices.

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

The `BindingContext` class is an interface that exposes generic methods to access the binding layer. Application developers that prefer working with type safe methods can cast the `bindings` instance to `DCBindingContainer`.

```
import oracle.adf.model.binding.DCBindingContainer;
...
DCBindingContainer bindingsImpl = (DCBindingContainer) bindings;
```

As with many implementation classes in Java, the `DCBindingContainer` class exposes more methods than defined by the `BindingContainer` interface.

DCIteratorBinding

In Fusion applications, the abstract `DCIteratorBinding` class is used to access the rowset iterator of a View Object. The `DCIteratorBinding` class handles the events generated from the associated `RowIterator` and sends the current `Row` to individual control bindings to display current data. To access the current row of an iterator from Java, you use

```
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.jbo.Row;
...
DCIteratorBinding dciter = null;
//access the iterator by its ID value in the PageDef file.
dciter = bindingsImpl.findIteratorBinding("DepartmentsView1Iterator");
Row currentRow = dciter.getCurrentRow();
```

The iterator ID in the binding definition is “DepartmentsView1Iterator”, as shown in figure 1-6, and is looked up in the `DCBindingContainer` instance.

OperationBinding

Method bindings, like “relocateEmployees”, or action bindings like “Next” or “Previous” exposed on a View Object are instance of `JUCtrlActionBinding` that can be casted the `OperationBinding` interface when accessed from Java. The following code is used to invoke the “Next” operation shown in figure 1-6

```
import oracle.binding.OperationBinding;
...
OperationBinding nextOperation = null;
nextOperation = bindingsImpl.getOperationBinding("Next");
Object result = nextOperation.execute();
```

The return value of the execute method operation is the result returned by the custom method, or null if there is no return value.

Other operations, like “ExecuteWithParams” or “CreateWithParameters”, that are exposed on the data control palette for a View Object require input arguments to be passed to it. The arguments may be added through `ExpressionLanguage` references, or provided in Java before invoking the method execution. The “relocateEmployees” method in figure 1-6 expects a single argument of type `Map<Number, Number>`, which is provided as follows

```
import oracle.binding.OperationBinding;
import oracle.jbo.domain.Number;
...
OperationBinding relocateEmployees = null;
relocateEmployees = bindingsImpl.getOperationBinding("relocateEmployees");
HashMap<Number,Number> emps = new HashMap<Number,Number>();
emps.put(new Number(103),new Number(90));
emps.put(new Number(104),new Number(90));
emps.put(new Number(105),new Number(100));
relocateEmployees.getParamsMap().put("employees", emps);
relocateEmployees.execute();
```

The method arguments are provided in a `Map`, which is accessed from a call to `getParamsMap` on the `OperationBinding` reference. The argument name is the same name as defined in the method signature, which can be looked at in the data control palette that exposes the method and the `pageDefinition` file that contains the method binding.

AttributeBinding

Attribute definitions in the ADF binding definition like “DepartmentName” in figure 01-6 are instances of `JUCtrlAttrsBinding` that can be casted to the `AttributeBinding` interface when accessed from Java. Attribute bindings are referenced from components like `InputText` and `OutputText`.

```
import oracle.binding.AttributeBinding;
...
AttributeBinding departmentName =
    (AttributeBinding) bindings.get ("DepartmentName");
String oldValue = (String) departmentName.getInputValue();
String newValue = oldValue+"_new";
departmentName.setInputValue(newValue);
```

JUCtrlListBinding

The list binding is used by select components and bind a single row attribute of an iterator to a secondary rowset that provides the selected value. In figure 01-6, the “JobId” list is bound to the “JobsViewIterator”. A selection from the list updates the “JobId” attribute of the `EmployeesView3Iterator`. To access the list binding from Java, you use the `JUCtrlListBinding` class as shown below

```
import oracle.jbo.uicli.binding.JUCtrlListBinding;
import oracle.jbo.domain.Number;
...
JUCtrlListBinding listBinding = null;
listBinding = (JUCtrlListBinding) bindings.get ("JobId");
ViewRowImpl selectedListRow = null;
selectedListRow = (ViewRowImpl) listBinding.getSelectedValue();
String jobIdValue = (String) selectedListRow.getAttribute ("JobId");
Number maxSalary = (Number) selectedListRow.getAttribute ("MaxSalary");
```

The list binding value, when accessed from Java, is the row in the secondary rowset that the user selected to update the bound attribute. Compared to plain HTML, in which lists cannot represent objects but only indexes and string values, this means a huge advantage of the ADF binding framework.

JUCtrlHierBinding

The `JUCtrlHierBinding` class is used to bind tree, tree table and table components to the ADF model. Dependent on the component you work with, the tree binding works with a single node or multiple node hierarchy. Chapter 9 in this book covers the work with collections to populate trees, tree tables and tables. To access a tree binding, like “EmployeesView3” in the example shown in figure 01-6, you use the following Java code

```
JUCtrlHierBinding hierBinding = null;  
hierBinding = (JUCtrlHierBinding) bindings.get("EmployeesView3");
```

JavaServer Faces for quick learners

JavaServer Faces 1.2 has been chosen as the view layer of choice in Fusion web application development of Oracle JDeveloper 11g. JavaServer Faces (JSF) is a Java EE standard since 2004 and, in its current version 1.2, part of the Java EE 5 platform. JSF is designed to simplify web application development and changes the programming model for building web pages from HTML markup development to the assembly of reusable UI components that are declaratively linked to their data model and server-side event handlers.

Architecture overview

As shown in figure 1-8, a JSF web application request is always dispatched by the front controller, which is an instance of `FacesServlet` that is configured in the `web.xml` descriptor file. The request is routed to the page that is identified by the `viewId` in the request. Components on a page are represented on the server as memory object within a virtual component tree. A component may reference a managed bean to execute client logic, like a method that is invoked by pressing a button. The visual presentation of a JSF page that is downloaded to the client in response to a request is handled by render kits. A render kit is a specialized set of Java classes that produce the display code for a specific device.

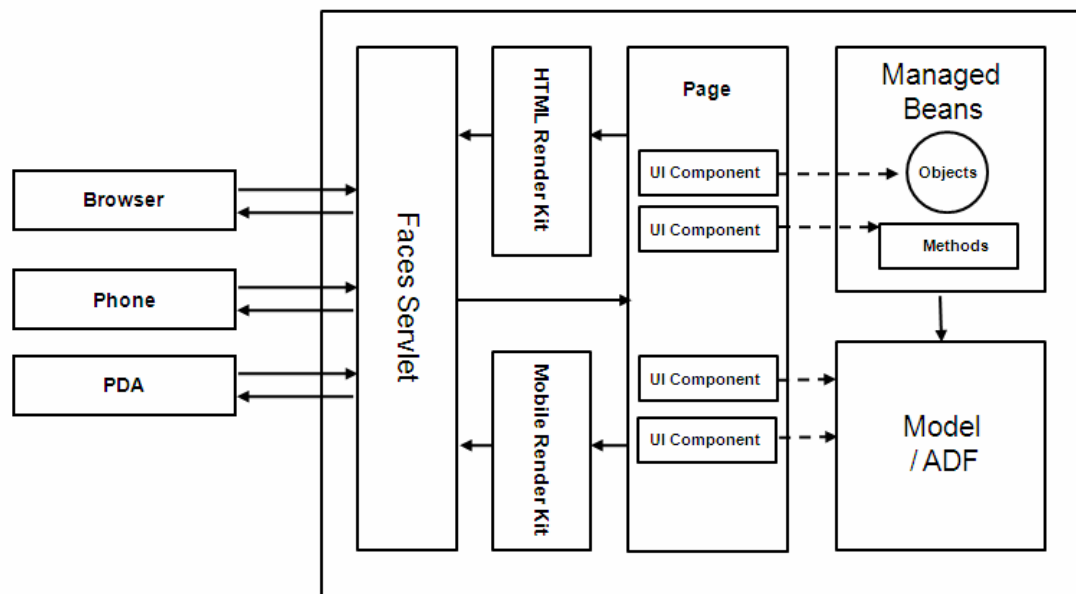


Figure 1-8: JavaServer Faces architecture diagram

Note: Managed beans are explained in detail later in this chapter

Building blocks

Building JavaServer Faces applications, application developers work with the following core building blocks

- **UI Component** A user interfaces component, like a button or a input text field is a reusable component that exposes its attributes as properties to the developer. Attributes are either set declaratively, using the Oracle JDeveloper Property Inspector, or in Java, referencing the component instance and calling the setter and getter method for the attribute to set or read. Components are displays agnostic so they can display on many different devices and not only browsers.
- **Managed Bean** A managed bean is a Plain Old Java Object with a no argument constructor that is configured to be managed by the JSF framework. Configuring a managed bean with JSF makes its public methods and properties available to Expression Language for use on the UI component attributes. The instantiation and dismissal of a managed bean is handled by JavaServer Faces so that there is no need for developers to maintain a beans lifetime.

- **Expression Language** A scripting notation that accesses public methods and properties on JavaBeans. Expression Language (EL) is used to bind user interface components to the data model, which can be a managed bean or the Oracle ADF binding. Expression Language is a key to “easy” in Java EE application.
- **Navigation** JavaServer Faces provides an integrated controller that enables developers to declaratively specify page to page navigation by defining named navigation cases. Navigation cases describe the page navigation starts form, its destination and the action or method that invokes this navigation.
- **Lifecycle** The lifecycle of markup driven technologies like JavaServer Pages (JSP) is request, compile and render in where no chances are given to developers to respond to individual steps in the process after the incoming request. In JavaServer Faces, there exists six steps in which incoming requests are processed, each of which the developer can listen and respond to. Chapter 3 of this book details the JavaServer faces lifecycle and puts it into the context of an ADF Fusion application.

Expression Language

Expression Language accesses memory objects, or objects exposed by the JavaServer Faces configuration, using the following syntax:

```
#{<bean>.<method>}
```

The <bean> reference could be to a managed bean or the standard servlet scopes: request, session and application. In Oracle ADF, additional scopes exist with PageFlow, view and backingBean, which also are EL accessible. The <method> reference either accesses a pair of getter and setter methods, if the access is from a component attribute that is not a listener, or a method, if the reference is from a command component or a listener attribute to execute an action.

Expression	Accesses
#{applicationScope.firstname}	Accesses an attribute with the name “firstname” in the application scope. If the attribute doesn’t exist on a first write attempt, it gets created. Because the scope is a shared across application instances, it

	should not be used for attributes specific to a user instance.
<code>#{sessionScope.firstname}</code>	Accesses an attribute with the name “firstname” in the session scope. If the attribute doesn’t exist on a first write attempt, it gets created. Because the scope lasts for the duration of the user session, it should not be used for page scoped attributes.
<code>#{requestScope.firstname}</code>	Accesses an attribute with the name “firstname” the request scope. If the attribute doesn’t exist on a first write attempt, it gets created.
<code>#{firstname}</code>	Searches the memory scopes for an attribute with the name “firstname” starting from the smallest scope. Since this expression is ambiguous as there is no guarantee in which scope the attribute is found, we discourage the use of it.
<code>#{cookie.lastVisited.value}</code>	Searches the browser cookies for a cookie named “lastVisited”. Other cookie attributes that can be accessed from EL include <ul style="list-style-type: none"> • domain • maxAge • path • secure
<code>#{param.firstname}</code>	The expression accesses a parameter on the request URL with the name “firstname”
<code>#{mybean.firstname}</code>	Accesses a bean property “firstname” that is exposed from a managed bean through its setter and getter methods. Managed beans have unique names within the JavaServer Faces configuration so that there is no need to use their scope within the expression.

It is the context in which EL is used that allows developers to assume whether the accessed resource is a property or a method. The access below is to a property since the reference is from a component “value” attribute

```
<af:inputText label="First Name" value="#{mybean.firstname}"/>
```

The access here is to an action listener, a method in a managed bean, since it is referenced from a listener attribute

```
<af:inputText label="Label 1" id="it1" [...] valueChangeListener="#{mybean.onValueChange}"/>
```

EL expressions cannot take arguments, which sometimes is limiting to the developer, forcing him or her to implement a work around that may require coding.

Building expressions in Oracle JDeveloper 11g

Oracle JDeveloper 11g provides an “Expression Builder” assistant that helps you to declaratively define expressions for JavaServer Faces component attributes and ADF binding attributes. To open the EL builder, select a JavaServer Faces component in the JDeveloper visual editor or the Structure window and open the property inspector. In the JDeveloper property inspector, click the arrow icon next to the property field, for example the *value* property. Choose **Expression Builder** or **Method Builder** from the context menu. This brings up the Expression Builder dialog from where you can browse and select the object to bind the attribute to.

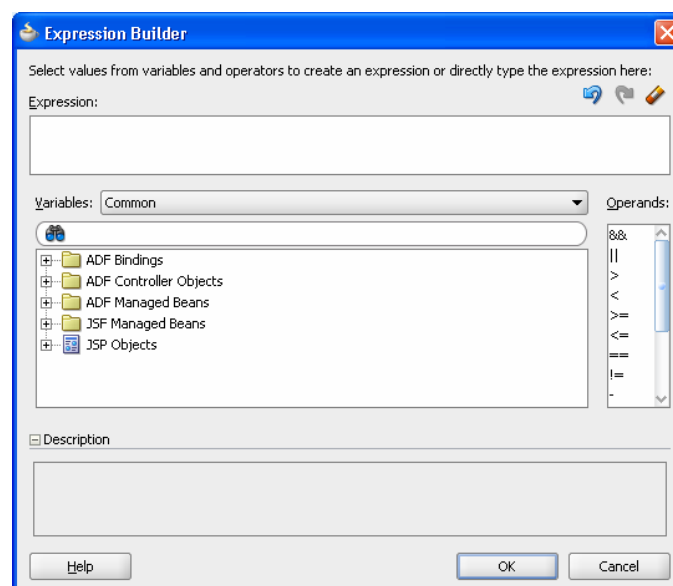


Figure 1-9: Expression Builder in Oracle JDeveloper 11g

The Expression Builder in Oracle JDeveloper 11g exposes the following categories, as shown in figure 1-9

- **ADF Bindings** contains references to the ADF binding layer of the current page and the ADF binding context, which exposes all configured binding definitions. Referencing the ADF binding layer in Expression Language is the key to simplicity in Fusion web application development
- **ADF Controller Objects** ADF Task Flows extend the JavaServer Faces navigation model and expose a `ControllerContext` object for EL access that allows developers to access information about the current displayed view, exceptions, the Task Flow URL for remote access, the train model and many more.

Note: ADF Task Flows are covered in detail in chapters four, five and six

- **ADF Manages Beans** ADF managed beans are beans that are configured in the in the Task Flow configuration, not in the standard JSF `faces-config.xml` file. The node contains sub-nodes that represent the available scopes, making it easy to prefix the bean reference with the name of the scope it is defined in.
- **JSF Managed Bean** shows the managed beans that are defined in the `faces-config.xml` file, also categorized by the scopes they are defined in.
- **JSP Objects** allows access to the objects exposed by the JavaServer Faces and Servlet APIs. It exposes the `FacesContext` object and gives developers access to the Servlet request header, cookies, Servlet initialization parameters and request parameters.

The Difference Between \$ and # in Expression Language

Expression Language isn't new in Java EE and is used as well in JavaServer Pages and the Java Standard Tag Library (JSTL). Though JavaServer Pages 2.1 and JavaServer Faces 1.2 use a unified Expression Language, there exist differences. The most obvious difference between EL used in JSP and JSF is that the EL expression in JSP 2.1 starts with a "\$" character whereas EL in JSF starts with a "#". Beside of this visual difference, there also exists a difference in the behavior.

EL that uses the "\$" syntax executes the expressions eagerly, which means that the result is returned immediately when the page renders. Using the "#" syntax defers the expression

evaluation to a point defined by the implementing technology. In general, JavaServer Faces uses deferred EL evaluation because of its multiple lifecycle phases in which events are handled. To ensure the model is prepared before the values are accessed by EL, it must defer EL evaluation until the appropriate point in the life cycle.

Context and Application objects

The JavaServer Faces framework classes application developers use the most are `FacesContext`, `ExternalContext` and `Application`. Both objects provide developers access to the JavaServer Faces framework, like the access to the current displayed view, and the Java EE environment, like the http session, request and response object and many more.

- **FacesContext** The static `FacesContext` class provides the request context information for the application instance and grants developers access to application wide settings, messages to be displayed to the end user, lifecycle methods and the displayed view root. The following methods are most commonly used with the `FacesContext`
 - `getCurrentInstance` returns a handle to the `FacesContext` instance
 - `addMessage` used to add a `FacesMessage` which content is displayed to the user
 - `getELContext` one way of programmatically creating an Expression Language reference is through the `ELContext` object, which is accessible from the `FacesContext`
 - `getViewRoot` returns `UIViewRoot`, the parent component of the page that allows developers to programmatically register lifecycle listeners, retrieve view root information and traverse the page component structure.
 - `renderResponse` a method that is used to short circuit the request lifecycle to immediately render the page response. This method is used whenever developers don't want the model to be updated or all components on a page to be validated. (See chapter 3 for more details)
- **ExternalContext** The external context wraps some of the external Servlet functionality and information, like the Servlet context, cookie maps, the remote user, the request locale, page redirect and many more. If developers don't find what they are looking within the set

of method exposed on the `ExternalContext` then using a call to `getResponse` or to `getRequest` return instances of `HttpServletResponse` and `HttpServletRequest` object. The `ExternalContext` object is accessible from the `FacesContext`.

- **Application** The `Application` object is also exposed from the `FacesContext` instance and grants developers access to application wide settings like the `NavigationHandler` instance to invoke navigation from listeners and component that are no command components, the default locale setting, the list of supported locales, configured message bundles and many more.

Configuration files

JavaServer Faces is configured through the `faces-config.xml` file and the `web.xml` file, which both are located in the `WEB-INF` directory of the web project.

faces-config.xml

The `faces-config.xml` file is the only configuration file available for all framework and application specific configurations in JavaServer Faces. Though multiple copies of `faces-config.xml` may be created in an application, or added from the `META-INF` directory of an attached library, at runtime only one instance of the configuration exists with its content accessible from the `Application` instance exposed on the `FacesContext`. The `faces-config.xml` file is visually configured in Oracle JDeveloper 11g. To open the visual dialog, double click the `faces-config.xml` file reference in the `ApplicationNavigator` and choose the **Overview** or **Diagram** tab.

web.xml

The `web.xml` file contains the configuration of the `FacesServlet` class that is mapped to the “/faces/” context path to server all requests that include this path with the request. If a project uses multiple faces configuration files then the additional files are configured through the “`javax.faces.CONFIG_FILES`” context parameter.

Managed bean and backing beans

Want to confuse a developer? Interview him or her about the difference between managed beans and backing beans in JavaServer Faces. In a short summary, backing beans are special uses of managed beans.

Managed beans

A managed bean is a JavaBean that is configured in the `faces-config.xml` file and that is managed by the JavaServer Faces framework. Managed beans are optional elements and can be used to hold presentation logic, navigation logic, state, execute Java methods or define event handlers. JSF instantiates managed beans when they are referenced from Expression Language. JavaBeans that should be configured as managed beans must have a no argument constructor. If a bean needs initialization parameters then these are configured through managed bean properties.

The example configuration below defines a managed bean “BrowseBean” with a managed property “bindingContainer” that makes the ADF binding layer available to the bean at runtime.

```
<managed-bean>
  <managed-bean-name>BrowseBean</managed-bean-name>
  <managed-bean-class>
    adf.interactiv.view.BrowseBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>bindingContainer</property-name>
    <property-class>
      oracle.binding.BindingContainer
    </property-class>
    <value>#{bindings}</value>
  </managed-property>
</managed-bean>
```

The `BindingContainer` instance is referenced from the request scope “bindings” object using Expression Language. The `BrowseBean` JavaBean contains a property “bindingContainer” with its associated setter and getter methods.

```
import oracle.binding.BindingContainer;
...
BindingContainer bindingContainer = null;
public void setBindingContainer(BindingContainer bindingContainer) {
    this.bindingContainer = bindingContainer;
}
public BindingContainer getBindingContainer() {
    return bindingContainer;
}
```

As soon as the managed bean is instantiated by the JavaServer Faces framework, the binding container becomes available within the JavaBean to work with.

Backing Beans

A backing bean is a special case of managed bean that has a one-to-one relationship to a single JavaServer Faces page. Backing beans expose setter and getter methods for the component contained on the page. By default, Oracle JDeveloper does not create a backing bean when creating a new JSF page because in many cases it is sufficient to work with a managed bean that is not strictly associated with a single page. Backing beans are good to use with smaller pages, like a login screen, that you need to have Java access to the contained components for. They produce unnecessary overhead though if used with large and complex pages, unless you manually delete all the component references that are not used in your Java client programming.

To create backing beans for each new page in Oracle JDeveloper, you need to enable the “Automatically Expose New UI Components in a New Managed Bean” option that displays after expanding the “Page Implementation” header in the JSF page creation dialog.

To create backing beans for existing JavaServer Pages, open the JSF page in the visual editor and choose **Design | Page Properties** from the tool menu. Check the “Auto Bind” option and create or select an existing managed bean to store the UI component references in and press Ok.

While backing beans appear attractive to use, there is a downside with it. The generated setter and getter entries in a backing bean are created for all components on a page, adding unnecessary weight to the size of the managed beans.

Backing Bean and Managed Bean analogy

A good analogy is to look at a managed bean as the backpack you take with you for hiking. When you go hiking, there is a lot of stuff you want to take with you. However, you know about their weights and therefore carefully decide about the items you want to put in and the items you better leave at home. A backing bean that gets generated by JDeveloper is like your partner packing the backpack for you. Because he or she doesn't know what exactly you need for the trip - but wants to make sure you have everything you need - they put in everything that comes into their mind. You then have to carry the weight over the hills. Because generating a backing bean creates setter and getter methods for all components on a page, including those you don't use in your Java programming, the "backing bean backpack" grows big and heavy. If using managed beans instead, you are packing the backpack yourself with better control over its weight.

Managed bean scopes

Managed beans have a scope, which is the lifetime between their instantiation by the JavaServer Faces framework and their dismissal for garbage collection. As a rule of thumb, always use the smallest managed bean scope possible. The standard scopes are

Scope	Description
None	The managed bean does not store any state and is released immediately. Use beans in no scope for helper classes that don't need to keep any memory.
Request	The managed bean lives for the duration of a request and stores its internal state until the request completes. This also is the only scope available for backing beans.
Session	The managed bean lives until the user session is invalidated or expires. This scope should be used carefully only for beans that carry information that is of use anywhere in the application. For example, a bean holding user information would be a good candidate for the session scope.
Application	The managed bean is available to all instances of an

	application. This scope could be used to define static lists of data used by all instances.
--	---

Managed beans can reference other managed beans using managed properties with the ADF binding reference example. The limitation for this reference is that a managed bean cannot have a managed bean property referencing a bean of a smaller scope. This means that a bean in request scope can reference a bean in session scope, but a bean in session scope cannot reference a bean in request scope. The exceptions to this rule are beans with no scope, which can be accessed from all beans scopes.

Creating managed beans in Oracle JDeveloper

In Oracle JDeveloper, managed beans can be created and configured in the `faces-config.xml` file or an ADF Task Flow configuration file in one of the following ways

- a double click on a command component like button or link within the visual page editor. This opens the dialog to create and configure a managed bean.
- selecting the **Edit** option in the context menu of an action or listener property, like `ValueChangeListener` or `ActionListener`, in the JDeveloper Property Inspector, This also opens a dialog to configure the managed bean.
- creating a JavaBean using the **File | New** menu option to open the new Gallery. Opening the `faces-config.xml` file or ADF Task Flow configuration to manually register the bean class as a managed bean.
- selecting the `faces-config.xml` file or the Task Flow configuration XML file, for example `adfc-config.xml`, in the Oracle JDeveloper Application Navigator and open the Structure window [ctrl+shift+s]. Select the root node and choose **Insert inside FacesConfig | managed-bean** or **Insert inside Task Flow | ADF Task Flow | Managed Bean**, based on which configuration file is selected.
- for Oracle Task Flows only, open the visual Task Flow diagrammer and drag and drop the **managed-bean** entry from the **Source Elements** category of the JDeveloper Component Palette [ctrl+shift+p] into it.

Recommended best practices for working with managed beans is to implement `java.io.Serializable` when creating the bean classes. Also a good practice to follow is to extend a managed bean class from a base class, which allows you to easily add common methods.

Managed beans that are no beans

Instances of `java.util.HashMap` can be configured as a managed bean to provide a static list of values to a managed bean instance

```
<managed-bean>
  <managed-bean-name>bookmarks</managed-bean-name>
  <managed-bean-class>java.util.HashMap</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <map-entries>
    <map-entry>
      <key>JDeveloperForum</key>
      <value>http://forums.oracle.com/forums/forum.jspa?forumID=83</value>
    </map-entry>
    <map-entry>
      <key>JDeveloperHome</key>
      <value>http://otn.oracle.com/products/jdev</value>
    </map-entry>
  </map-entries>
</managed-bean>
```

This creates a managed bean as an instance of `HashMap` that can be accessed from Java and Expression language.

```
{bookmarks [ 'JDeveloperForum' ] };
```

But didn't we say that Expression Language cannot have arguments? Yes, we did. But there exists an exception to the rule, which is for managed beans that are instances of `HashMap` and for resource bundles. In the `HashMap` example case shown above, the expression resolves to `bookmarks.get('JDeveloperForum')` at runtime.

Note: Developers sometimes create subclasses of `HashMap` and override the `get()` method to do something different from what the default behavior of a `HashMap` does. This works around the existing limitation in EL to not allow arguments.

Events

The JavaServer Faces event model is based on JavaBean events in where events are represented by specific classes that pass information about the event to interested listener components. There are two types of events in JavaServer Faces: component events and lifecycle events.

Component events

Component events are events that a component sends to the server to notify interested listeners about a user interaction like button press or value change. Input components, for example, expose a *valueChangeListener* attribute for developers to register a Java listener to handle the event.

```
<h:inputText id="lastname"  
  valueChangeListener="#{EmployeesBacking.onLastNameValueChange}"/>
```

The input text component references a managed bean method from its *ValueChangeListener* attribute. This registers the managed bean method as an event listener that is notified whenever the submitted component value differs from the previously displayed value. The managed bean method has the following signature

```
public void onLastNameValueChange(ValueChangeEvent valueChangeEvent) {  
  // Add event code here...  
}
```

The method receives an instance of the *ValueChangeEvent* object that contains information about the new value, the old value, the UI component instance that raised the event. The method code can now handle the event or queue the event for processing later during the request lifecycle. If the *immediate* attribute of the input text component is set to “true” then the event is raised early in the request lifecycle, allowing developers to handle it and, if needed, prevent further processing of the request by a call to *renderResponse* on the *FacesContext*.

Another option to add a value change listener is to write a custom Java class that implements the *javax.faces.event.ValueChangeListener* interface and to reference it by its absolute package and class name from the *type* attribute in a *f:valueChangeListener* tag. To use the *ValueChangeListener* tag, add it as a child component to the input component about which value change the class should receive a

notification. The `ValueChangeListener` interface defines a single method `processValueChange` that takes the `ValueChangeEvent` object as an input argument.

Navigation

Navigation in JavaServer Faces is defined in the form of navigation rules stored in the `faces-config.xml`. As shown in figure 1-10, page navigation rules in Oracle JDeveloper are composed declaratively by linking together two page references with a “JSF Navigation Case” component from the Component Palette. Any edits that are performed in the visual diagrammer, the Property Inspector or the Structure Window are written as metadata in the `faces-config.xml` file.

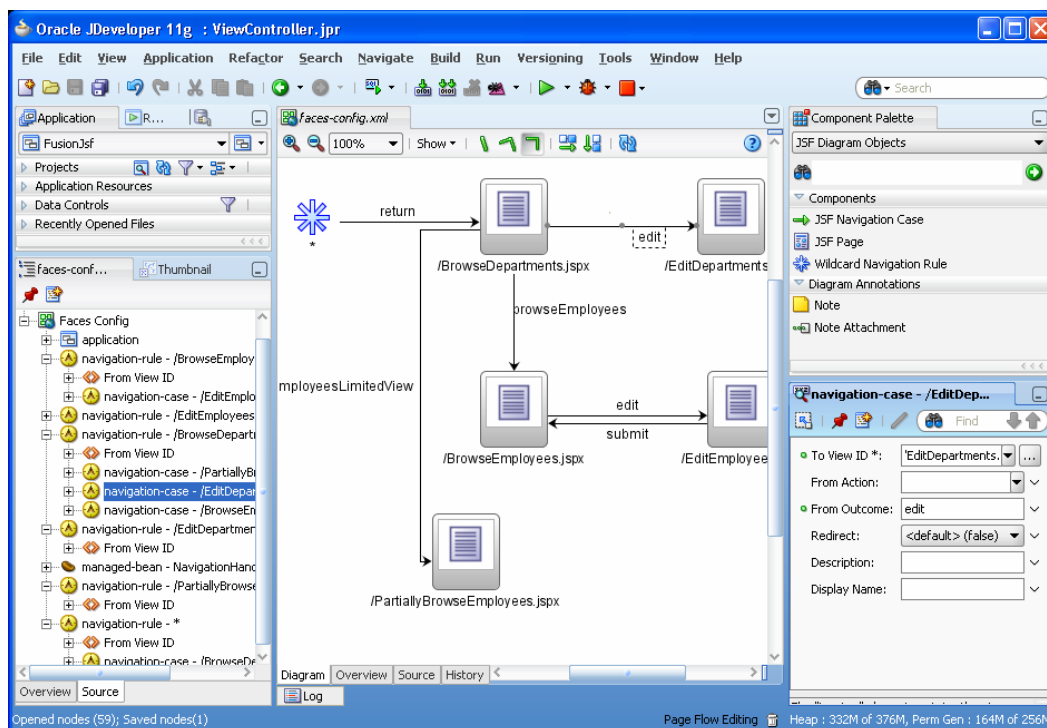


Figure 1-10: Visual editing support for navigation JSF navigation rules in JDeveloper 11g.

```
<navigation-rule>
  <from-view-id>/BrowseDepartments.jspx</from-view-id>
  <navigation-case>
    <from-outcome>employeesLimitedView</from-outcome>
    <to-view-id>/PartiallyBrowseEmployees.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>
      #{NavigationHandlerBean.accessBrowseEmployees}
    </from-action>
  </navigation-case>
</navigation-rule>
```

```
</from-action>
<from-outcome>browseEmployees</from-outcome>
  <to-view-id>/BrowseEmployees.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

The navigation rule shown above is defined for the `BrowseDepartments.jsp` page and contains two navigation cases. If the outcome of a command component, like button or link, matches the “`browseEmployees`” or “`employeesLimitedView`” outcome then JSF routes the request to the configured destination page.

```
<af:commandButton text="BrowseEmployees"
                  action=" employeesLimitedView"/>
```

Instead of hard coding the action string, The `af:commandButton` *action* attribute may also reference a managed bean method that dynamically returns a valid outcome strings to perform routing decisions in Java.

Navigation case may use the “`from-action`” element in addition to or instead of the “`from-outcome`” element to configure navigation, as shown in the example above. If the “`from-action`” element is used without a “`from-outcome`” element, then any action string returned from the referenced managed bean method performs navigation to the defined destination. If the “`from-outcome`” element is included in addition, as shown in the example above, then navigation is performed only if the returned method outcome is “`browseEmployees`”. Using the “`from-action`” element allows developers to define multiple navigation cases with the same outcome name.

```
<af:commandButton text="BrowseEmployees"
                  action="#{EmployeesBean.accessBrowseEmployees}"/>
```

The method signature of the referenced bean looks as follows

```
public String accessBrowseEmployees() {
    if (...) {
        return "employeesLimitedView";
    }
    else {
        return "browseEmployees";
    }
}
```

The lookup order in which JavaServer Faces performs navigation based on the existence of the from-action and from-outcome elements is

1. elements specifying both from-action and from-outcome
2. elements specifying only from-outcome
3. elements specifying only from-action

If no navigation case outcome matches the action string defined on a command component or returned by a managed bean method, then the navigation is returned to the same page.

A special navigation case is the use of wild cards that can be accessed from any page that matches to the defined wild card pattern. Wildcards are useful for navigation that needs to be defined on many pages, like a logout or home command that navigates the request to a logout page or the home page

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>return</from-outcome>
    <to-view-id>/BrowseDepartments.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Note: ADF task flows that you use when building ADF Fusion web applications do not use the `faces-config.xml` file to define the navigation but `adf-config.xml`. Chapter 4 and 5 discuss ADF task flows and how to work with them.

ADF Faces Rich Client for quick learners

Oracle ADF Faces RC is the chosen JavaServer Faces based view layer technology for Oracle Fusion. It provides more than 150 high end Ajax enabled UI components and the infrastructure that allows application developers to build future safe Rich Internet Applications (RIA). ADF Faces Rich Client shields developers from the complexity that exists in Ajax for the orchestration of JavaScript, CSS, DHTML and XML to render cross browser functional highly interactive and dynamic web user interfaces. Oracle fusion developers work with a single API, which is the JavaServer Faces programming interface.

In addition to more than 150 Ajax enabled JavaServer Faces UI and graphical data visualization components and operations, ADF Faces also contains a client-side programming model and lifecycle that is similar to the JavaServer Faces model but executes independently of the server. For example, field validation configured on a component is executed on the client first and if it doesn't pass this validation, no request is sent to the server.

Using the public JavaScript API in ADF Faces, developers can search and manipulate components displayed on the client without the need to program to the browser DOM (Document Object Model), which is less error prone and also simplifies cross platform application development. Using the ADF Faces client framework, application developers call getter and setter methods on client objects to read and write component properties from JavaScript. Events can be received and queued on the client, with the ability to directly call server side managed bean methods from JavaScript. Any state change that is applied to a component on the client is automatically synchronized back to the server to ensure a consistent client and server state.

A list of the non UI functionality provided by the ADF Faces Rich Client framework that simplify the work of a Rich Internet Application (RIA) developer include

- **Drag and drop framework** Tag library elements and listeners that declaratively implement drag and drop functionality for ADF Faces web applications (see chapter 14)
- **Dialog and popup framework** JavaServer Faces components that render is child components in lightweight dialogs with client and server side listeners to send notification about the launch and close event. These dialogs are also used to display context menus on tables and other components (see chapter 10)
- **Partial Page Rendering (PPR)** UI components are refreshed without page reload. PPR is the integrated behavior of components like tree, table, accordion, panel splitter and many more. Beside of the integrated component behavior, all components can be declaratively configured to become a partial event source and target, which means they can trigger other components to refresh or are refreshed in response to published events of other components. Using PPR in combination with the ADF binding layer automatically refreshes all dependent components based on data changes in the underlying iterator binding.

- **Active Data Service (ADS)** Push mechanism that refreshes ADF Faces components based on server side events (see chapter 20).
- **JavaScript client API** The ADF Faces Rich Client architecture has a server side and a client side. The client side framework consists of internal and public JavaScript objects and interfaces that developers optionally use to program client side functionality (see chapter 19).
- **Skinning** Look and feel customization framework that exposes specific CSS selectors on the ADF Faces component renderers for developers to change the default look and feel at runtime without any code changes required in the application (see chapter 16).
- **ADF binding** ADF Faces uses a specific set of binding classes to enable ADF binding as the component model and to expose its functionality through Expression Language. For example, the ADF Faces tree component ADF uses the `FacesCtrlHierBinding` class that extends the `ADFJUCtrlHierBinding` base binding class. It exposes an additional `getTreeModel` method that returns an instance of Apache Trinidad `TreeModel` class as the tree component model. Similar, operation and method bindings are exposed through the `FacesCtrlActionBinding` class.

Maturity through history

The Oracle ADF Faces Rich Client components have their origin in Oracle User Interface XML (UIX). Oracle first released UIX in 2002 for its external and internal customers to build web applications that by this time already made use of partial page rendering, a behavior that today is described as an Ajax behavior pattern. Shortly after JavaServer Faces became a Java EE standard in 2004, Oracle released more than 100 JavaServer Faces components under the ADF Faces branding. After donating the ADF Faces components to the Apache open source community, where they are further developed as part of the MyFaces Trinidad project, Oracle started a new Ajax JSF component project under the name of ADF Faces Rich Client. The base component technology of ADF Faces Rich Client is Apache Trinidad, which from an Oracle perspective is a great return of open source investment.

Two additional context classes

Oracle ADF Faces provides two context objects in addition to the default `JavaServer Faces` context objects.

The `RequestContext` object is defined by the Apache Trinidad component framework that is the basis of ADF Faces Rich Client components. The context object is accessible from EL, using `{requestContext.<method>}`, and Java, using `RequestContext.getCurrentInstance()`. The `RequestContext` allows developers

- To launch pages in external browser window dialogs
- To detect partial page requests
- To get information about the client like browser name and type of device, like browser, mobile or phone.
- To programmatically register components as partial targets, which is useful when the component needs to be refreshed at the end of a managed bean method execution
- To programmatically add partial listeners to a component and to notify interested listeners about events
- To get a handle to the help provider that is configured for providing context sensitive help
- To determine if a request is an initial page request or a postback request
- To access the page flow scope, which is a scope with a length between request scope and session scope that is available for pages launched in dialogs and bounded task flows, a concept that is introduced later in chapter 4.

The `AdfFacesContext` object mirrors the `RequestContext` object, but adds ADF Faces specific APIs, like access to the ADF Faces view scope. All public methods of the `RequestContext` are accessible through the `AdfFacesContext` object as well. The `AdfFacesContext` object is accessible from Expression Language, using `{adfFacesContext.<method>}` and from Java, using a call to `AdfFacesContext.getCurrentInstance()`.

Configuration

ADF Faces is configured automatically when choosing the ADF Faces Technology Scope in the Oracle JDeveloper project properties, or when building a new application based on the “Fusion Web Application (ADF) template”.

The three configuration files of interest are `web.xml`, `trinidad-config.xml` and `trinidad-skins.xml`.

web.xml

The `web.xml` file contains the configuration of the Trinidad servlet and resource filter mapping to the JavaServer Faces `FacesServlet`. The Trinidad servlet filter initializes ADF Faces RC for each incoming request and sets the `RequestContext` object. Additional context parameters are available for the application developer to set and configure. Context parameters are added to the `web.xml` file as a child of the “web-app” element.

```
<context-param>  
  <param-name>parameter</param-name>  
  <param-value>value</param-value>  
</context-param>
```

A list and description of all context parameters is printed in Appendix A of the Oracle documentation “Oracle® Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11g Release 1”.

trinidad-config.xml

The `trinidad-config.xml` file is the main ADF Faces configuration file and exists in the `WEB-INF` directory of the ADF Faces view project. By default, the only configuration that exists in this file is the skin family string that determines the application look and feel. A list and description of parameters is printed in Appendix A of the Oracle documentation “Oracle® Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11g Release 1”.

All configurations in the `trinidad-config.xml` file are exposed to Expression Language and Java through the `AdfFacesContext` and the `RequestContext` object.

trinidad-skins.xml

ADF Faces allows developers to define the application look and feel in external CSS files. The `trinidad-skins.xml` file acts as the registration of custom skin definitions, including the CSS file reference, skin family name definitions and skin inheritance references. Chapter 16 of this book covers Skinning and the settings of the `trinidad-skins.xml` file in great depth.

Components

ADF Faces Components are Java EE and JavaServer Faces compliant and are not dependent of Oracle Fusion Middleware or Oracle JDeveloper 11g.

Working with ADF Faces components

Fusion web application developers add ADF Faces components declaratively add design-time using the ADF Faces JSP tag library, or programmatically at runtime, by creating and adding a Java object instance of the component.

Declarative development

Dragging an ADF Faces component from the Oracle JDeveloper Component Palette to drop it to a JavaServer Page configures the “af” XML namespace as an attribute of the `jsp:root` element.

```
xmlns:af=http://xmlns.oracle.com/adf/faces/rich
```

With this name space definition, all component markup elements that start with “af:” are handled by the ADF Faces Rich Client tag library. For example, an ADF Faces `InputText` component is added as `af:inputText` to the JSF page source, whereas a command button shows as `af:commandButton`. Developers working with the WYSIWYG visual editing environment aren't exposed to the tag elements and instead see the rendered component output.

ADF Faces Components are organized in three categories: Common Components, Layout and Operations.

- **Common component** Contains all ADF Faces components that have a visual representation for the user to read and input data, invoke actions and navigation or displays dialogs and menus.

- **Layout** Layout development in JavaServer Faces is through the nesting of panel components that each has a specific behavior in how child components are arranged on a page and where child components can be added. The layout category shows a list of all ADF Faces layout components.
- **Operations** This category contains all operational, non-visual components that can be used as a child of a visual component. Components include validators, converters and action listeners.

Programming

At runtime, every ADF Faces component on a page has a JavaBean instance representation that is created by the ADF Faces JSP tag library. To dynamically, add and bind components, this instance can be created dynamically in Java. All ADF Faces RC component classes start with “Rich”, so that an InputText component has a class of RichInputText and a Table has a component class of RichTable.

In JavaServer Faces, components can be added as the direct child of a parent component, or added to the facet of a component. To create a command button at runtime, you use the following code

```
RichCommandButton button = new RichCommandButton();  
button.setText("Delete");  
[...]  
panelGroup.getChildren().add(button);
```

ADF Faces Rich Client tag documentation

The full Oracle ADF Faces tag documentation is available online and contains a complete description of component attributes, facets and useful information about how to work with the component. To access the tag documentation, point your browser to otn.oracle.com and select **Documentation | JDeveloper** on the web site menu. The link next to “Oracle Application Development Framework API References (Javadoc)” navigates you to the API reference documents, which also contains a reference to “Oracle ADF Faces Tag Reference”

ADF Component Binding

ADF Faces components are ADF binding aware in that the ADF Faces UI framework provides specific binding classes that allow UI components to directly bind to ADF. ADF Faces component expressions that bind to ADF have the form of

```
{bindings.<binding entry>.<property>}
```

The <binding entry> corresponds to a binding entry in the ADF page binding, the “<page>PageDef.xml” file. The type of the <binding entry> can be list, action, attribute, tree and others and correspond to a specific Faces binding class.

The <property> can be a method, or a property exposed on the ADF Faces binding class. As shown in figure 01-7, all ADF Faces binding classes are contained in a private package, which means that though they are public classes, developers are not supposed to use them directly but the generic ADF binding classes. Developers that need to access ADF Faces binding specific methods from Java can do this using Expression Language and we provide code examples in chapter 9 when we discuss UI development with the ADF Faces Tree, Table and TreeTable component. The table below associates the Faces control binding class with the generic ADF binding class and where it is used.

Faces binding class	Extended ADF class	Usage
FacesCtrlAttrsBinding	JUCtrlAttrsBinding	Accesses attribute bindings in the pageDef file
FacesCtrlCalendarBinding	JUCtrlHierBinding	Represents the binding instance for ADF Calendar component. It provides CalendarModel and CalendarActivity instances for the Calendar component
FacesCtrlHierBinding	JUCtrlHierBinding	Implements the ADF Faces Component tree model interface to bind the tree component to the ADF tree binding
FacesCtrlHierNodeBinding	JUCtrlHierNodeBinding	Tree node binding object that

		adds functionality to the base binding class
FacesCtrlActionBinding	JUCtrlActionBinding	FacesCtrlActionBinding binds ADF Faces action components to ADF actions and methods that are invoked from EL through an exposed “execute” method.
FacesCtrlListBinding	JUCtrlListBinding	Provides access to the ADF list binding for SelectChoice Faces components
FacesCtrlLOVBinding	JUCtrlListBinding	Provides the model implementation for lov components
FacesCtrlSearchBinding	JUFormBinding	Used to build the model for the <code>af:query</code> component

Note: Working with the ADF binding classes from Java is an advanced topic that developers who start with ADF are not faced with. In general we recommend the use of declarative solutions over code centric solutions because they are easier to maintain and upgrade.

Partial Page Rendering (PPR)

Partial page rendering, or partial refresh, is a mechanism in ADF Faces to notify components about data change events or other triggering events for them to refresh their display without redrawing the full page. Two main Ajax patterns are implemented with PPR: cross-component refresh and single component refresh.

Single component refresh functionality is available on many ADF Faces components, like the ADF Faces table component. The table component provides built-in partial refresh functionality that allows users to scroll over displayed rows, to sort the table by a click on the

column headers, to change the table column order and to mark a row or several rows as selected.

Cross-component refresh is implemented declaratively or programmatically in that an ADF Faces RC component is registered as a partial listener of another component. The following two component attributes are used to initiate cross component partial page event to be triggered by a component

- **autoSubmit** When the `autoSubmit` attribute is set on an input component like `af:inputText` or `af:selectManyListbox` the component automatically submits the enclosing form in response to an event like a `ValueChangeEvent`. The submit is issued as a partial submit and doesn't lead to a reload of the page
- **partialSubmit** When the `partialSubmit` attribute of a command component is set to true, the page partially submits the form when the button or link is clicked. A partial submit sends an `ActionEvent` to the server without reloading the page.

To implement partial component refresh, you have a choice, ranging from declarative, to automatic and programmatic.

Declarative PPR

ADF Faces components on a page can be declaratively configured to receive partial refresh notification when an action is performed on another component. For example, selecting a value in a select box should refresh a dependent list box to show a changed list of data. To declaratively configure a UI component on a page to refresh in response to a change of another, open the Property Inspector on the component that should receive the event notification and navigate to the *PartialTriggers* property. Use the **Edit** context menu option that displays when selecting the arrow icon on the right to select the UI component that should send the change notification.

Automatic PPR initiated by the ADF binding layer

In Oracle JDeveloper 11g, the change event policy feature can be used to automatically refresh UI components that are bound to an ADF binding which data has changed in the model layer. To use this feature, you set the *ChangeEventPolicy* property of the ADF iterator binding and the attribute bindings to "ppr", which also is the default setting.

With the ADF Faces page open in Oracle JDeveloper, select the **Bindings** tab at the bottom of the visual editor or the source editor. In the “Executables” section, select the iterator that should refresh its bound UI components when changed. Open the PropertyInspector if not open and navigate to the *ChangeEventPolicy* property in the advanced section and set its value to “ppr”.

In Oracle JDeveloper 11 R1, the automatic PPR functionality is implemented on the following ADF Faces binding objects

- FacesCtrlAttrsBinding
- FacesCtrlLOVBinding
- FacesCtrlListBinding
- FacesCtrlHierBinding

Programmatic PPR

In ADF Faces, UI components can be refreshed programmatically through Java. For example, a managed bean method can be setup to listen and respond to events, like *ValueChangeEvent*, on the component that should trigger the partial refresh on another component. To partial refresh a referenced component based on an action in the source component, the managed bean executes coddle like shown below

```
AdfFacesContext adfFacesContext = AdfFacesContext.getCurrentInstance();  
adfFacesContext.addPartialTargets(<target component instance>);
```

Summary

The Oracle Application Development Framework (ADF) has become a corner stone of Oracle Fusion Middleware. ADF is built on Java EE and SOA standards and designed to simplify web application development in Java EE. The Oracle Fusion development platform is a technology choice within ADF that Oracle has chosen for building its next generation of standard business software. The choice includes the ADF binding, ADF Business Components and ADF Faces Rich Client frameworks, which are in the focus of this book. When working with Oracle ADF, we recommend you to use declarative programming over code centric programming when you have a choice. We believe however that good developers need to

understand both, declarative and code centric programming so they can identify the best solution to a problem.

If you ask us for a single recommendation of best practices to apply in all of your ADF development projects, then it is to **keep the layers separated!** This means that at any time, the business service layer should not have dependencies to the view or binding layer and the view layer should not be dependent on business services classes other than domain types. Use the view layer for UI logic and the business service for your business logic.