Oracle Application Development Framework — Development Guidelines Manual

Contents

Chapter 1, Introduction to J2EE Application Development in JDeveloper

The Oracle Application Development Framework (Oracle ADF), available with Oracle JDeveloper 9.0.5.1 and later, offers developers greater flexibility and openness when deciding how to implement the layers of a J2EE enterprise application. Oracle ADF brings “data control” abstraction to back-end business services (data sources) and generalizes Oracle’s existing data binding objects to support it. These features of Oracle ADF give developers a consistent and pluggable model layer to the J2EE application architecture.

This chapter provides an overview of the ways in which Oracle ADF supports J2EE application development in JDeveloper.

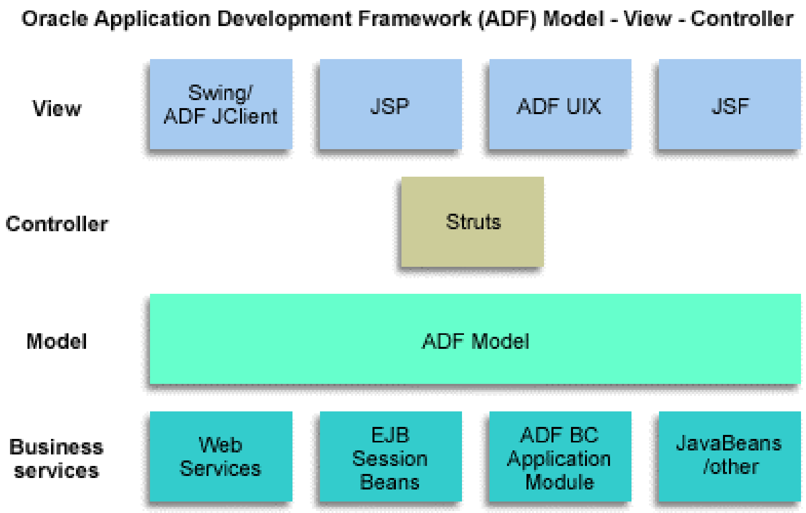
Summary

Introduction

This document addresses the enterprise application developer who wants to use JDeveloper to implement enterprise business solutions for the J2EE platform (Java 2 Platform, Enterprise Edition). Enterprise applications are built from various components that derive from standards-based technologies. As a J2EE-compliant development tool, JDeveloper supports building enterprise applications using these same standard technologies:

- JavaServer Pages technology to handle the presentation of the user interface
- Servlet and JavaBeans technology for the Apache Struts framework to manage the application flow
- Enterprise JavaBeans to manage application state and define the business logic

In addition, JDeveloper extends the J2EE paradigm by giving developers alternatives to the standard technology stack. Developers can elect to work with a wide variety of technologies, as illustrated by the following diagram.



The purpose of this document is to show how developers can combine technologies to suit their particular application needs. Recommendations are made in the interest of ease of development and recognized best practices that exploit the design time of the JDeveloper IDE.

The remainder of this chapter describes the application development choices JDeveloper provides in more detail. These choices comprise best practices that can improve application reliability, increase your productivity, and decrease the overall time to deployment.

Where to find additional information:

- For information about JDeveloper IDE features that support team development, testing, and production deployment to the J2EE platform, see the JDeveloper help system. These topics are beyond the scope of the present document.
- For a list of J2EE-related learning resources, see Related Information links at the end of this chapter.

Considering the Application Architecture

The enterprise application is typically deployed across multiple servers to achieve its distributed component architecture. On the client side, the part of the application that users interact with, the application can be browser-based or it may be a standalone client based on Java Swing components. JDeveloper supports two databound web application user interface technologies and one standalone client:

- Standard JSP pages and HTML elements
- Oracle's own ADF UIX pages with its own set of ADF UIX user interface components
- Standard Java/Swing components with Oracle's own ADF JClient bindings

The JDeveloper IDE for building enterprise applications provides equal support for all styles of application clients. Screen designers, whether creating web pages or Swing forms, work with databound UI components through a single, easy-to-use design time tool, known as the *Data Control Palette*.

For the middle tier, several technologies are available in JDeveloper to help define the application's business logic. In the J2EE platform, Enterprise JavaBeans components provide the persistence layer for the application and manage transactions between the client and the back-end data store. However, in JDeveloper, business logic developers can choose to implement this functionality using a variety of technologies:

- Standard Enterprise JavaBeans components
- Oracle ADF Business Components technology, which implements many of the design patterns required by transactional applications
- OracleAS TopLink mapping objects based on JavaBeans components

Or when the middle tier need not support transactional operations (that is, when users can commit or roll back changes), the developer can provide data-access and method-execution operations to the view and controller layers using these technologies:

- Web services
- Standard JavaBeans components

All five of the above business services are accessible at design time through the JDeveloper Data Control Palette. Because data controls abstract business services, the Swing UI developer, the page designer, or the controller layer developer can work from a single, consistent design time regardless of the chosen technology stack.

Note: In JDeveloper, business objects that define the business logic are referred to as *business services*. The *Oracle ADF model layer objects* abstract the implementation of a business service and provide access to the application in a consistent way for all business services. As already mentioned, JDeveloper provides a range of available business services technologies that work out of the box with Oracle ADF.

The following section describes how you use these technology choices in JDeveloper to implement the application based on the Model-View-Controller architecture.

Partitioning Application Development in JDeveloper

The J2EE architectural design pattern for the interactive enterprise application is known as Model-View-Controller, or MVC. The MVC pattern is ideally suited for the kind of application that combines distributed application logic with a complex user interface. When developing applications based on the MVC pattern, the goal is to enforce separating or "partitioning" the application logic and the user interface.

In the most general terms, the model is the underlying logical representation, the view is the visual representation, and the controller specifies how to handle user input. When the data model changes, it notifies all views that depend on it. This separation of state and presentation results in these important characteristics of the enterprise application:

- Multiple views can be based on the same model. For instance, the same data can be presented in both table form and chart form. As the data model becomes updated, the model notifies both views and gives each an opportunity to update itself.

- Because models specify nothing about presentation, you can modify or create views without affecting the underlying model.

A view uses a controller to specify its response mechanism. For instance, the controller determines what action to take when receiving keyboard input. To accomplish this requires additional objects to pass information between the two layers, but the benefits are worth the effort. Having a clean separation between application layers at each tier makes it easier for the development team to divide roles and responsibilities.

In JDeveloper, the integration effort is minimized, since the design time helps establish the wiring between the model, the view, and the controller, as described in the following table.

Layer	Browser-Based Web Application	Java Client Application
Model layer	A thin data binding layer, identical for web applications and Java clients, known as the *Oracle ADF model layer*, provides access to the business objects. In the web application, instances of the model layer are created by standard Struts action classes.	A thin data binding layer, identical for Java clients and web applications, known as the *Oracle ADF model layer*, provides access to the business objects. In a Java client application, instances of the model layer are created in Java code during a panel initialization.
View layer	HTML, Struts tags, and JSTL tags comprise the view layer in the JSP web application. JDeveloper also provides an alternative view technology, known as *Oracle ADF UIX components*. Both JSP and Oracle ADF UIX have full design time support that integrates them with the model layer.	Standard Swing UI components comprise the view layer in the Java client graphical user interface. JDeveloper provides design time integration with the model layer for Swing components and specialized composite widgets (like chart controls), known as *JClient controls*.
Controller layer	The Struts action servlet that dispatches incoming requests from the view layer to the appropriate action classes in the model layer. The Oracle ADF UIX technology is also fully integrated with the Struts controller.	Standard Swing UI components serve the role of the controller layer in the Java client. Again, JDeveloper provides design time integration with Swing components.

The following section provides more detail about the technology stack available with Oracle ADF.

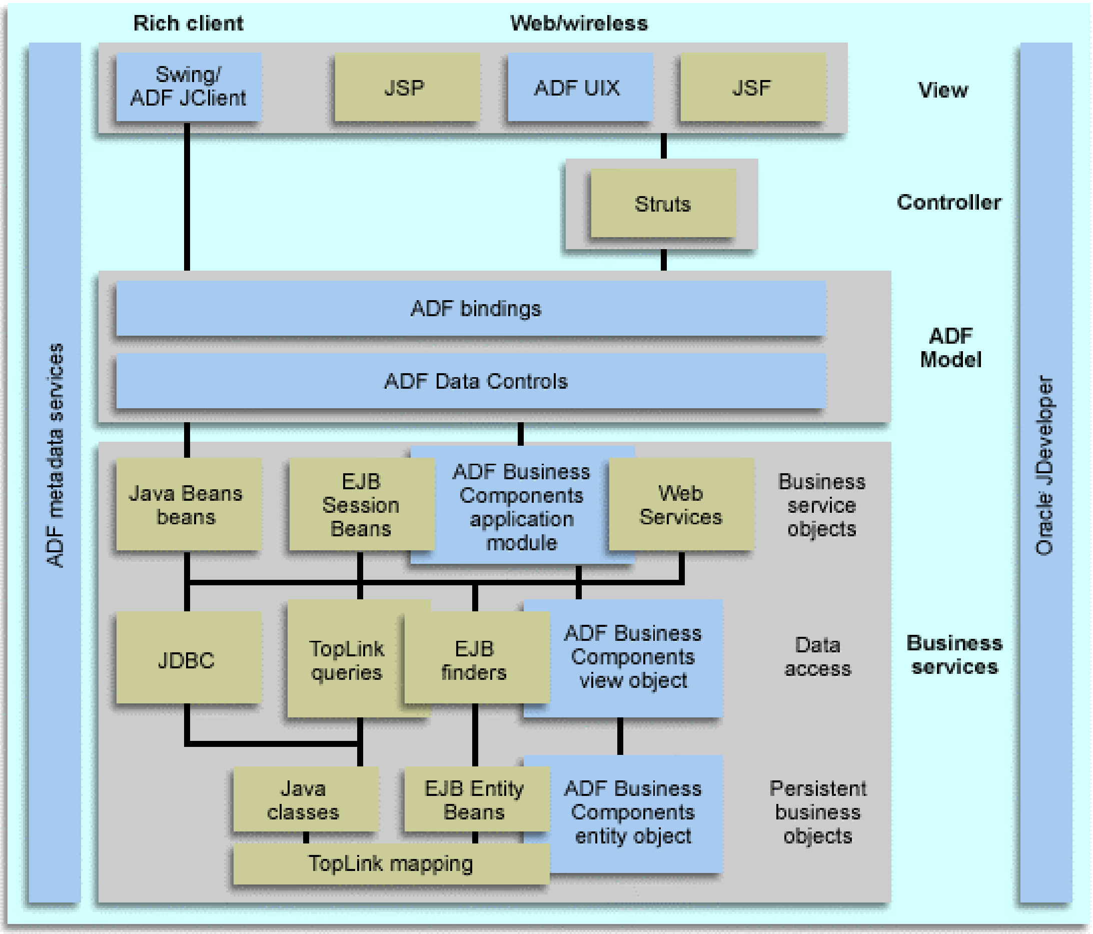
Speeding Development with Frameworks in JDeveloper

Frameworks make sense for developers because developers can write code based on well-defined interfaces. This is largely a time-saving benefit, but it also makes sense in a J2EE environment because J2EE frameworks provide the necessary infrastructure for the enterprise application. In other words, J2EE frameworks make the concepts expressed in the J2EE design patterns more concrete.

One example of a J2EE framework already familiar to many web application developers is the framework available from Apache Software Foundation, known as *Struts*. Web application developers currently work with the *Struts framework* to manage the flow of their application. Simply, JavaBeans technology and declarative definition files define the Struts framework. The framework provides the web application view layer with a single, centralized point of access for request handling.

Another such framework, also provided with JDeveloper, is the Oracle Application Development Framework (Oracle ADF). In its entirety, Oracle ADF provides you with the means to easily create the business services, the model, the controller, and the view layers. The full stack of Oracle ADF looks like this:

Oracle Application Development Framework (ADF) Application Architecture



The salient point when using the Oracle ADF framework is that you may use it in its entirety or you can elect to use just the Oracle ADF model layer, just the Oracle ADF UIX view, or just the Oracle ADF BC business services. In fact, JDeveloper places no restrictions on the Oracle ADF framework technology usage. For example, JDeveloper makes it especially easy to integrate a non-Oracle ADF business service with the Oracle ADF UIX view technology. For the complete set of applicable J2EE technologies, see the illustration at the beginning of this document.

The glue that makes customization of the technology stack possible is the Oracle ADF model layer, represented in the diagram as the Oracle ADF data controls and data bindings. Chapter 3, ADF Business Components in Depth describes the objects of that layer in detail. For now it is sufficient to understand that the Oracle ADF model layer places no restrictions on which view technology or which back-end business service to use. Further, because the Oracle ADF model layer is designed by Oracle as a thin integration layer, the Oracle ADF model objects provide data binding and other services without degradation of performance to the application.

The following section describes how the JDeveloper design time helps you to build seamless enterprise applications easily and transparently.

Development Methodology in JDeveloper

Oracle ADF and the JDeveloper design time support two high-level concepts of development: iterative development and division of labor. These concepts lay the foundation for the Oracle ADF best practices described in subsequent chapters.

Consider the demands of iterative development before examining roles-based development in more detail.

Iterative Development and Visual Tools

Developers approach a business problem by analyzing and dividing the task into its constituent parts. The team then proceeds to create the application’s components, including model data abstractions, screens that the user will interact with, and code to manage the application flow. The development phase frequently progresses iteratively. That is to say, as the application grows, and the constituent parts become better identified, developers expect to be able to add new components to the application, for example, to address new functionality.

The finished result, created from diverse components, must function as a coherent whole. To allow you to work in this fashion, the tools of the IDE must be both highly visual and interactive. The benefit of getting immediate feedback that an addition was successfully integrated across layers is vital to creating robust applications. JDeveloper serves this need with a rich assortment of visual tools.

Tools such as Oracle ADF Business Components wizards, the Struts Page Flow Modeler, the JSP Visual Editor, and the Data Control Palette assist in iterative development because you can create an object in one layer and allow others to work with it in their own layers. For instance, the JSP or Oracle ADF UIX page created by the page designer will be accessible to the developer managing the application’s flow control in the Struts Page Flow Modeler. Similarly, the objects of the business services are exposed to the Oracle ADF model layer through the Data Control Palette, which is refreshed to display items used to create data bindings for the UI components.

Roles and Code Integration

An additional demand for building enterprise applications is the need to support the development roles suggested by the model-view-controller paradigm. The model layer designer, for example, should decide what to expose from the model data, but the UI developer need not be concerned with how this is performed. Likewise, page flow control should be tightly integrated with the actual pages of the application, but the page designer should not have to write Java code for this purpose.

In the realm of web application development, where the model-view-controller paradigm is known as *Model 2* to distinguish it from the early notion that pages might contain the flow control logic, the need for experts to work by roles has real merit. At this time, the Struts framework provides the integration mechanism that allows page designers to refer to page handler classes (Struts actions) in a declarative fashion. Similarly, in JDeveloper, the visual editor together with the Data Control Palette (the design time for working with the Oracle ADF model layer) simplifies binding UI components or Struts actions to data and methods. In the case of web pages, the result is easy-to-read binding expressions that appear as part of the markup language of the page itself (HTML elements, Struts form tags, or Oracle ADF UIX elements).

Overall, the JDeveloper design time permits developers to work across the model-view-controller areas, without requiring detailed knowledge of how the integration is accomplished.

Proceeding with Application Development in JDeveloper

To quickly become familiar with the full technology stack provided in Oracle ADF, continue reading this document. Here are some specific suggestions that may begin to address your application development questions:

1. Read the Chapter 2, Business Services and the Oracle ADF and determine which business services the business logic developers will be most comfortable creating when exposing the model data to the application. Each business service, including Oracle's own transactional technology, Oracle ADF Business Components, provides its own unique set of features and benefits.
2. Determine the style of application you want to create: one that uses JSP pages, Oracle ADF UIX pages, or Oracle ADF JClient for Swing forms. More information about each of these technologies can be found in Chapter 6, Overview of Oracle ADF Data Binding in View Technologies dealing with the view layer.
3. If your application will be browser-based, and you are uncertain about whether or not to work with the Struts framework, read Chapter 5, Overview of Oracle ADF Integration with Struts on the integration of Oracle ADF and the controller layer. JDeveloper supports Model 1–style application development when the Struts controller is not required. However, both Model 1–style and Model 2–style applications have full integration support with the Oracle ADF model layer for working with data and methods.

4. In your browser, run the Quick Tour of the JDeveloper IDE for an overview of the available enterprise application development tools. The Quick Tour is accessible from the JDeveloper help system *Getting Started* topics.
5. In JDeveloper, create an application workspace based on a template that meets your application needs. The JDeveloper help system describes this technology-scoping feature in the *Working with Application Design Tools* topics.
6. Investigate typical use cases for your application in the end-to-end procedural documentation. The JDeveloper help system provides this information in the *Working with Oracle ADF* topics.

Once you have decided how to implement your application, selecting an application template or creating a custom template in JDeveloper will streamline the IDE and set up project folders with the appropriate standard libraries. The technology scope feature is particularly useful when the development effort is underway and limiting choices to the application requirements is desirable.

It is important to note that application templates and technology scopes, once selected, do not permanently remove tools and technologies from JDeveloper, nor do they prevent you from accessing the full list of technologies should you need to alter the chosen technology stack.

Related Information

Additionally, the following resources may be helpful when you wish to read more about J2EE and JDeveloper:

- For getting started information on the Oracle Technology Network, specifically for new users, see http://otn.oracle.com/new/index.html.
- For Oracle JDeveloper production information on the Oracle Technology Network, see http://otn.oracle.com/products/jdev/index.html.
- For the Oracle JDeveloper documentation page on the Oracle Technology Network, see http://otn.oracle.com/documentation/9i_jdev.html .
- For the Sun Microsystems home page for J2EE, see http://java.sun.com/j2ee/.
- For the Apache Software Foundation home page for Struts, see http://struts.apache.org/.

Chapter 2, Business Services and the Oracle Application Development Framework

Business services are behind-the-scenes components that mediate between an MVC application and a data source (usually a database). Business services are responsible for the following:

- Retrieving data requested by the rest of the application
- Representing this data as Java objects usable by the rest of the application (object-relational ["O/R"] mapping)
- Persisting changes made by the rest of the application
- Implementing business rules, such as validation logic, calculated attributes, and defaulting logic
- Providing services that can perform large-scale batch operations on data upon request

Business services segregate the persistence and business logic of an application from the logic that governs the application's UI and control flow. Keeping persistence and business logic separate allows you to reuse them in multiple MVC applications.

This chapter provides an overview of the business service technologies that work out of the box with the Oracle Application Development Framework.

Summary

- How OracleAS TopLink Technology with POJO Provides Data Access Components
- How OracleAS TopLink Technology with POJO Provides Service Objects
- How Enterprise JavaBeans Technology with TopLink CMP Provides Persistent Business Objects
- How Enterprise JavaBeans Technology with TopLink CMP Provides Data Access Components
- How Enterprise JavaBeans Technology with TopLink CMP Provides Service Objects

The Available Business Service Technologies

JDeveloper provides tools to develop business services using multiple technologies. You can usually choose a business service technology independently of the design of the rest of your application: the Oracle ADF Model (described later) provides a common interface for all business services, allowing your view and controller to access business services almost interchangeably. All of the following technologies are supported out of the box by the ADF Model; you can also support additional business service technologies by creating your own data control classes.

Oracle ADF Business Components Technology

ADF Business Components technology is a fully-featured, XML-based framework for creating business services. ADF Business Components evolved from the Business Components for Java (BC4J) technology distributed with Oracle9*i* JDeveloper and earlier releases. The ADF Business Components runtime library handles most business service functionality, which you can customize declaratively (by changing the XML files using JDeveloper's RAD tools) or programatically (by extending library classes). Oracle ADF Business Components technology:

- Automatically handles O/R mappings and persistence for instances of its own library classes
- Allows you to make complex requests for data retrieval using SQL
- Automatically handles transaction management, including optimistic or pessimistic locking
- Provides a framework for implementing complex business logic
- Automatically implements many J2EE design patterns
- Has a powerful caching and data passivation system for increasing the performance and scalability of applications

All of the above functionality is fully customizable: if you do not like the way ADF Business Components handles O/R mappings, for example, you can override it.

Enterprise JavaBeans Technology

Enterprise JavaBeans technology is an alternative for creating business services without a framework. While EJB execution depends on an EJB container (which is part of any J2EE-compliant application server), running EJB beans requires no Oracle-specific runtime library or other Oracle-specific technology.

Enterprise JavaBeans technology:

- Generally relies upon the application server to handle O/R mappings and persistence (this is called "container-managed persistence," or CMP), although experienced EJB developers can override the application server and code persistence logic themselves ("bean-managed persistence," or BMP)
- Requires requests for data to be made using Java APIs or an EJB-specific query language called "EJB QL"
- Generally relies upon the application server to handle transaction logic (this is called "container-managed transactions," or CMT), although experienced EJB developers can override the application server and code transaction logic themselves ("bean-managed transactions," or BMT)
- Requires you to implement your own business logic
- Requires you to implement J2EE design patterns, if desired
- Relies on the Java object caching capabilities of the application server

OracleAS TopLink Plain Old Java Objects (POJO)

OracleAS TopLink is a technology for providing complex mappings between Java objects and a relational database. TopLink POJO uses ordinary JavaBeans as the Java objects. Unlike users of ADF BC, users of TopLink POJO do not extend a framework; instead, they create their own object model and allow the TopLink runtime to integrate it with the database.

TopLink technology with POJO:

- Automatically handles O/R mappings and persistence logic for arbitrary Java objects
- Allows you to make complex requests for data retrieval using SQL, EJBQL, or TopLink's own expression language
- Automatically manages transactions
- Requires you to implement your own business logic
- Requires you to implement J2EE design patterns, if desired
- Has a powerful caching and data passivation system for increasing the performance and scalability of applications

Like ADF Business Components, TopLink mapping technology is highly customizable.

Enterprise JavaBeans Technology with TopLink CMP

You can also use TopLink mappings to provide container-managed persistence for EJB entity beans. Doing so will override the CMP behavior of your application server.

Enterprise JavaBeans technology with TopLink CMP:

- Uses TopLink technology to handle complex object-relational mappings and persistence
- Allows you to make complex requests for data retrieval using SQL, EJBQL, or TopLink's own expression language
- Automatically manages transactions

- Requires you to implement your own business logic
- Requires you to implement J2EE design patterns, if desired
- Relies on the Java object caching capabilities of the application server

Web Services

Web services are a special case. Rather than a technology for *implementing* business services, web services use XML-based standards to enable application-to-application interaction across the Web regardless of platform, language, or data format. An MVC application can use web services as a wrapper for business services in any format deployed anywhere on the Web.

For example, a web service can wrap Enterprise JavaBeans, ADF Business Components, stored procedures in a database, or other business services written in a Java or any other language. MVC applications can access the service's API using XML messages sent over the Web.

Because of this, web service technology does not itself handle O/R mappings, persistence, data validation, or caching, instead leaving these to the underlying classes.

Java Objects with Hand-Coded Persistence

Oracle ADF also allows you to use any JavaBeans-compliant Java objects as business services for your application. If you choose to go this route, you will need to implement your own framework for data retrieval, persistence, and manipulation. Usually, this involves:

- Retrieving data from the database using JDBC
- Implementing your own O/R mapping framework
- Persisting data to the database using JDBC
- Manually coding transaction management using JDBC
- Adding your own business logic to the getters and setters for your classes
- Creating your own caching mechanism or sacrificing scalability

For the vast majority of users, this option is not recommended. It is intended primarily for users who have already created their own complex framework for business services.

Which Business Services Technology Should I Use?

There is no single answer to the question of which business services technology is the best. The right choice of a business services technology depends on your needs, your background, and your priorities.

Note: Web services are not included in the following discussion. They have a very specific purpose: providing very loose coupling between an MVC application and its business services. In cases where you need such loose coupling, web services are the only choice; in other cases, they are not an appropriate choice.

Do You Have Your Own Object Framework?

OracleAS TopLink POJO can provide O/R mappings and caching for arbitrary Java objects. For this reason, OracleAS TopLink is generally the best alternative for developers and organizations who have a Java object framework in place or who wish to create one. If you have your own systems or requirements for representing business objects, implementing business logic, and shaping and aggregating the data for clients, TopLink POJO is most likely the best option for your projects. EJB technology requires that your components match EJB specifications, and ADF BC component classes must extend ADF BC framework classes, but TopLink will work with any object model to provide O/R mapping, data retrieval and caching, and transaction functionality.

Do You Want to Use an Existing Object Framework?

If you are creating a completely new application, with no existing application infrastructure, Oracle ADF Business Components technology is the most productive option you can choose. ADF Business Components technology handles all aspects of application plumbing completely automatically: O/R mapping, data retrieval and caching, transaction management, and integration with the ADF data binding layer. In addition, ADF Business Components automatically implements key J2EE design patterns to improve performance and scalability; it provides a framework for creating validation rules and other business logic; and it includes base classes to represent your entities and views.

Can You Use Oracle Runtime Technology?

Oracle ADF Business Components and OracleAS TopLink are both 100% J2EE-compliant technologies that will run on any J2EE-compliant application server. Neither of these technologies requires you to use an Oracle database or application server, nor do they in any way restrict which technologies you can use for the view or controller layer of your application.

Both of these technologies, however, make use of some Oracle runtime classes on the application server. If you use ADF Business Components, your business services will extend the ADF Business Components base classes. If you use OracleAS TopLink, your business services will rely on the TopLink runtime to provide O/R mappings and caching.

If you have requirements that prevent you from using any Oracle classes at runtime, you will need to choose a different business service technology: either EJB with CMP provided by the application server, or entirely hand-coded JavaBeans-based business services.

Business Service Layers

All business services must have a way of handling three main tasks:

- Representing data
- Retrieving and shaping data for clients to use
- Presenting data and specific services to clients

Each of these tasks is accomplished by a separate layer of components: persistent business objects, data access objects, and service objects, respectively.

Note: “Components” and “objects” are here being used in a logical, rather than programmatic, sense. While some business service technologies use Java objects for their persistent business objects, data access objects, and service objects, others may use methods, queries, or other logical constructs.

Persistent Business Objects

In general, your business services should always have a layer of *persistent business objects*, which are based on the most logical representation of the data. If you already have a well-designed data source, the structure of your persistent business objects should reflect the structure of that data source. This practice allows you to write business rules in the most logical way, expressing the properties and requirements for the actual entities your application must represent.

If your application needs to work with customers, orders, and order items, for example, it will generally need to have persistent business objects representing customers, orders, and order items, respectively. You will write business rules on these objects, expressing the properties and requirements for customers, orders, and order items.

Data Access Components

Persistent business objects provide a logical representation of your data, but they do not always provide exactly the data your application needs. Data access components organize and shape data to match the needs of a particular application.

For example, your application might need to work with the specific data returned by the following SQL query:

```
SELECT *

FROM ORDER_ITEMS
WHERE PRODUCT_ID = 501;
```

or even the more complicated query

```
SELECT
   CUSTOMERS.CUSTOMER_ID,
   CUSTOMERS.FIRST_NAME,
   CUSTOMERS.LAST_NAME,
   ORDERS.ORDER_ID
FROM
   CUSTOMERS,
   ORDERS
WHERE
   CUSTOMERS.CUSTOMER_ID=ORDERS.ORDER_ID;
```

Retrieving this information would be handled by data access components. The power of data access components depends on the particular business services technology: some can handle queries of arbitrary complexity; some can only retrieve collections of persistent business objects (the equivalent of queries that start with `SELECT *` and have only one table in their `FROM` clause).

Service Objects

A *service object* is the single point of contact between your business services and the rest of the application. Service objects handle transactions, provide access to data model components, and expose high-level tasks to the application in the form of *service methods*.

Detailed Comparison of Business Service Architectures

Any business services technology must have a way of implementing the three business service layers.

Note: Web services technology and Java objects with hand-coded persistence are special cases. In the case of web service technology, the web service itself provides the service object, but the implementation of persistent business objects and data access objects is up to the developer who writes the web service. In the case of Java objects with hand-coded persistence, implementing persistent business objects, data access components, and service objects is also entirely up to the developer.

The following table shows how the other business services technologies implement persistent business objects, data access components, and service objects.

Technology	Persistent Business Objects	Data Access Components	Service Objects
Oracle ADF Business Components	ADF entity object definitions	ADF view object definitions	ADF application module definitions
Enterprise JavaBeans	EJB entity beans	Finder methods	EJB session beans
OracleAS TopLink with POJO	JavaBeans classes with TopLink mappings	TopLink queries	JavaBeans classes
Enterprise JavaBeans with TopLink CMP	EJB entity beans with TopLink CMP	TopLink finder methods	EJB session beans

How ADF Business Components Technology Provides Persistent Business Objects

ADF Business Components technology uses *ADF entity object definitions* as persistent business objects. As with persistent business objects in all technologies, a single entity object maps to a single entity in the data source (in the vast majority of cases, these data sources are tables or views in a database).

An entity object definition is the template for *entity object instances*, which are single Java objects representing individual rows in a database table. For example, an entity object definition called “Departments” could provide a template for entity object instances that represent individual rows of the DEPARTMENTS table.

An entity object definition can have up to four parts:

- An XML file, which represents the portion of the entity object definition that can be developed declaratively. Most of the information is that needed for O/R mapping, but it can also contain simple validation rules, called *validators*. For many entity object definitions, the XML file by itself is sufficient.
- An entity object class, which represents individual entity object instances. Entity object classes allow you to write complex business logic in Java, when using XML validators is not sufficient. Entity object classes extend the class `oracle.jbo.server.EntityImpl`. If you do not need custom Java business logic, you need not generate an entity object class—ADF can use `oracle.jbo.server.EntityImpl` directly to represent rows of the data source.
- An entity definition class, which represents the data source object in its entirety. Entity definition classes act as Java wrappers for the XML file, so if you need special handling of the metadata (for example, if you need to change it dynamically), you can add this code in the entity definition class. Entity definition classes extend the class `oracle.jbo.server.EntityDefImpl`. If you do not need custom handling of your metadata, you need not generate an entity definition class—ADF can use `oracle.jbo.server.EntityDefImpl` directly to wrap the metadata.
- An entity collection class, which represents the cache of rows (instances of the entity object class) held in memory for a single user. The vast majority of developers do not need to generate an entity collection class: you should do so only if you want to override ADF Business Components' default caching behavior.

When entity object definitions are based on database objects, columns in the database object map to a single *entity object attribute* in the entity object definition. The definitions of these attributes (reflected in the entity object definition's XML file) reflect the properties of these columns, such as the columns' data types, column constraints, and precision and scale specifications. When entity object definitions are based on objects from other data sources, entity object attributes map to "columns" from those objects, as defined by the developer. If you generate an entity object class, attributes will also be represented as fields in that class.

As mentioned previously, you can declaratively add validation logic to entity object definitions or attributes in the form of validators. JDeveloper comes with four simple validators:

- CompareValidator, which compares an attribute to a value (either a literal value or a value drawn from the data source).
- ListValidator, which checks to see whether an attribute is in a list of values (either a literal list or the results of a query).
- RangeValidator, which checks to see whether an attribute is between two literal values.
- MethodValidator, which can invoke any method which returns a boolean value. Validation is passed if the method returns `true`.

In addition, you can create your own custom validators. These require coding to create initially, but once created, they can be applied declaratively in many different projects.

If the default validators do not meet your needs, and you do not want to create your own validators, you can also put validation code in the entity object class: in the setter methods (for attribute-level validation) or in a method called `validateEntity()` (for multiattribute validation).

Entity object classes also provide hooks for other business logic, including the methods `create()`, which is called whenever a new row is created, and `remove()`, which is called whenever a row is deleted. By adding business logic to these methods, you can ensure that the logic is invoked whenever rows are created or deleted.

Finally, although ADF Business Components technology automatically handles DML operations for you (by issuing INSERT, UPDATE, and DELETE commands to the database), you can also override this behavior by overriding the `doDML()` method in the entity object class—if, for example, you want to use stored procedures in the database to handle DML operations.

Relationships between entity object definitions are handled by Oracle ADF associations, which define a relationship between two Oracle ADF entity object definitions based on sets of entity attributes from each. These can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships. For example, associations can represent:

- The one-to-many relationship between a customer and all orders placed by that customer
- The one-to-one relationship between a product and its extended description (if these are represented by separate entity object definitions)
- The many-to-many relationship between products and the warehouses that currently stock them

How ADF Business Components Technology Provides Data Access Components

ADF Business Components technology uses *ADF view object definitions* as data access components. ADF view object definitions collect data from the data source, shape that data for use by MVC applications, and allow those applications to change the data in the Oracle ADF Business Components cache.

A view object definition can have up to four parts:

- An XML file, which represents the portion of the entity object that can be developed declaratively. This information consists of the mechanism (usually a SQL query) that the view object uses to retrieve data from the data source, and the way in which the columns of the SQL query map to entity attributes (which handle actual O/R mapping). For many view object definitions, the XML file by itself is sufficient.
- A view object class, which represents an individual instance of the query result set, called a *view object instance*. Different users will always use different view object instances, but the same user may also use multiple view object instances. View object classes allow you to write custom methods that affect multiple rows in a query. View object classes extend the class `oracle.jbo.server.ViewObjectImpl`. If you do not need to write custom view object methods, you need not generate an entity object class—ADF can use `oracle.jbo.server.ViewObjectImpl` directly to represent instances of the query result set.

- A view row class, which represents individual rows of the query result. View row classes allow you to write custom methods that affect a single row of data, and they provide typesafe accessors to retrieve and change data. View row classes extend the class `oracle.jbo.server.ViewRowImpl`. If you do not need custom row-level methods or typesafe accessors, you need not generate a view row class—ADF can use `oracle.jbo.server.ViewRowImpl` directly to represent rows of the data source. (`ViewRowImpl` contains the `getAttribute()` and `setAttribute()` methods, which allow you to retrieve and change data, but these methods are not typesafe.)
- A view definition class, which represents the query itself. View definition classes act as Java wrappers for the XML file, so if you need special handling of the metadata (for example, if you need to change it dynamically), you can add this code in the view definition class. View definition classes extend the class `oracle.jbo.server.ViewDefImpl`. If you do not need custom handling of your metadata, you need not generate a view definition class—ADF can use `oracle.jbo.server.ViewDefImpl` directly to wrap the metadata.

Columns in the query map to individual *view object attributes* in the view object definition. The definitions of these attributes (reflected in the view object's XML file) reflect the properties of these columns, including data types and how, if at all, they should be mapped to entity object attributes. If you generate a view row class, attributes will also be represented as fields in that class.

A view object definition can be based on a query of arbitrary complexity: joins, calculated attributes, and even group functions can be represented within a view object definition.

As mentioned previously, view object definitions handle the mapping of query columns to entity object attributes. There is no requirement that attributes of a single view object all map to attributes of the same, or even any, entity object. The view object handles reading from the data source, and the entity object definitions (if any) handle DML. This operation of the ADF Business Components cache is one of its most powerful features.

Among Sun's J2EE BluePrints™ design patterns is the "fast-lane reader" pattern. Rather than search through persistent business objects for needed data, a fast-lane reader queries the data source directly. This is a much faster operation than searching the persistent business objects, but has the disadvantage of being read-only, since persistent business objects handle updates to the data source.

View object definitions are an improvement on the fast-lane reader pattern. Like the fast-lane reader, they query the database directly, but they optionally store the results in entity object instances and maintain pointers to those instances. Therefore, they can query the database with fast-lane reader speed but avoid the read-only disadvantages of standard fast-lane readers.

Relationships between view object definitions are handled by Oracle ADF view link definitions, which define a relationship between two Oracle ADF view object definitions based on sets of entity attributes from each. Like associations, these can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships.

Individual view object instances can also be related by individual view link instances, which create a master-detail relationship between the query result sets. For example, suppose that you have view object definitions representing a query for department information and a query for employee information, and a view link between the view object definitions representing the relationship between a department and its employees. If an instance of the former view object definition, `allDepartments`, is related to an instance of the latter, `employeesInDepartment`, by an instance of the view link, those instances will be synchronized: whenever a particular row of `allDepartments` is selected, `employeesInDepartment` will only display details of that row.

How ADF Business Components Technology Provides Service Objects

ADF Business Components technology uses *ADF application module definitions* as service objects. ADF application module definitions serve as a single point of contact between MVC applications and the business services layer; they manage transactions; and they provide a container for view object and view link instances.

An application module definition can have one or two parts:

- An XML file, which represents the portion of the application module definition that can be developed declaratively: the view object and view link instances that the application module definition contains and the way in which they are related. For many application module definitions, the XML file by itself is sufficient.
- An application module class, which lets you write custom code such as service methods that an MVC application can invoke for batch data handling. Application module classes extend the class `oracle.jbo.server.ApplicationModuleImpl`. If you do not need to write custom service methods, you need not generate an application module class—ADF can use `oracle.jbo.server.ApplicationModuleImpl` directly.

The most important feature of an application module definition is its *data model*—the view object and view link instances it contains. These specify what data the client will have access to, and what relationships hold within that data.

You can use application module definitions in two different ways:

- As a service object, in which case each instance of the MVC application has access to one instance of the application module. These *root-level* application module instances control ADF BC transaction objects, which in turn control the entity and view caches.
- As a resusable object for nesting, in which case you can create a data model and service methods on it and then nest one of its instances in other application module definitions. Those application module definitions can, in turn, access the nested module’s methods and data model. Nested application modules share the root-level application module’s transaction.

Application module instances are elements in an application module pool, a configurable resource manager that automatically decides whether an instance needs to be maintained, with its caches, in memory, whether it can be serialized to the database to save memory, or whether it can be removed altogether. Application module pooling can greatly increase the scalability of your application.

How Enterprise JavaBeans Technology Provides Persistent Business Objects

Enterprise JavaBeans technology uses *EJB entity beans* as persistent business objects. As with persistent business objects in all technologies, a single entity bean maps to a single entity in the data source (in the vast majority of cases, these data sources are tables or views in a database). For most purposes, applications should use container-managed persistence (CMP), which allows an external container (such as an application server or the TopLink runtime) to handle O/R mappings. The alternative is bean-managed persistence (BMP), which shares many disadvantages with Java classes that use hand-coded persistence: to use BMP entity beans, you must handle O/R mappings, persistence, and data retrieval yourself. BMP beans are mostly useful for applications that need to use a data source other than the database.

EJB CMP entity beans can have up to six parts:

- An element in the *EJB deployment descriptor*, an XML file that describes all aspects of a set of EJB beans. This entry describes the bean's metadata, such as how its fields are mapped to database columns.
- A bean class, which is an abstract class listing the fields of the bean.
- A local home interface, which contains methods that create instances of the bean class, for use by other beans in the same container.
- A home interface, which contains methods that create instances of the bean class, for use by classes outside the container.
- A local interface, which is how other EJB beans interact with the entity bean.
- A remote interface, which is how Java objects outside the EJB container interact with the bean.

Note that a CMP entity bean does not have any parts that can be directly instantiated: it consists of an XML element, an abstract class, and three interfaces. The CMP provider will automatically extend the abstract class to create an implementation class, which is directly instantiated to create *entity bean instances*, which represent single rows in the database. Similarly, the CMP provider automatically implements the home and/or local home interfaces. You will never need to work with the implementation classes directly: your application will always refer to entity bean instances through the local or remote interface, and to the home through the home or local home interface.

Columns in the database object map to single fields in the entity bean class and single subelements of the XML element. The attributes of the XML subelements reflect the properties of these columns, such as the columns' data types, column constraints, and precision and scale specifications.

Unlike ADF entity object definitions, EJB entity beans do not provide hooks for business logic. You cannot implement business logic in accessor methods because the implementation of those methods is left to the CMP provider (in the bean class, the methods are abstract). However, JDeveloper can automatically generate a *data transfer object* for an entity bean. The data transfer object is an implementation of a J2EE Blueprints design pattern and serves as an intermediary between MVC applications and the entity bean. Data transfer objects expose some or all of the entity bean's attributes, and have getters and setters that you can modify to implement business logic. They do not, however, provide a convenient way to implement multiattribute logic. If you use EJB technology, you must implement multiattribute logic in the controller layer or write your own framework code to create it.

Relationships between entity beans are handled by *container-managed relationships* (CMRs), which define a relationship between two entity beans based on sets of fields from each. These can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships. For example, associations can represent:

- The one-to-many relationship between a customer and all orders placed by that customer
- The one-to-one relationship between a product and its extended description (if these are represented by separate entity object definitions)
- The many-to-many relationship between products and the warehouses that currently stock them

How Enterprise JavaBeans Technology Provides Data Access Components

Unlike ADF Business Components technology, Enterprise JavaBeans technology does not use actual objects to provide data access components. Instead, EJB applications use *EJB finder methods* as data access components.

EJB finder methods are methods on the EJB's home interface. When you create a finder method, you can specify a query in a specialized query language called EJBQL. The query will get added to the deployment descriptor, and the CMP provider will use it to generate code to search among the entity bean instances for those matching the query conditions.

Unlike ADF view object definitions, EJB finder methods have limits to their complexity. Since they simply return collections of entity beans, they cannot implement joins or calculated attributes. Also, because EJB finder methods query the entity beans instead of the database, they do not provide the efficiency of the fast-lane reader pattern. You can, of course, implement the fast-lane reader pattern yourself, but since this pattern does not create a collection of entity beans (but rather simply a list of data), you cannot use it to perform DML operations. For optimal performance, you should create and use a fast-lane reader whenever read-only data is sufficient, and use EJB finder methods in other cases.

How Enterprise JavaBeans Technology Provides Service Objects

Enterprise JavaBeans technology uses *EJB session beans* as service objects. Session beans serve as a single point of contact between MVC applications and the business services layer, and provide access to entity bean home and local home interfaces.

As with entity beans, there are two types of session beans: for most purposes, applications should use *container-managed transactions* (CMT), which allows an external container (such as an application server or the TopLink runtime) to handle transactions. The alternative is *bean-managed transactions* (BMT), which requires you to write code to handle transactions yourself, and is useful only if you have specialized requirements for transaction handling.

An EJB CMT session bean definition can have up to six parts:

- An element in the EJB deployment descriptor. Unlike the elements for entity beans, this element contains only the bean's name.

- A bean class, which lets you write custom code such as service methods that an MVC application can invoke for batch data handling. Unlike bean classes for CMP entity beans, this class is not abstract.
- A local home interface, which contains methods that create instances of the bean class, for use by other beans in the same container.
- A home interface, which contains methods that create instances of the bean class, for use by classes outside the container.
- A local interface, which is how other EJB beans in the container interact with the session bean.
- A remote interface, which is how Java objects outside the EJB container interact with the bean.

Unlike ADF application module definitions, a session bean does not contain a complete data model. Instead, it has *EJB local references* to the local home interfaces of the “master” entity beans. Detail entity beans are accessed from the masters through container-managed relationships.

The ADF model layer can currently use only *stateless* session beans. These are beans that do not maintain any state in between client access—they simply serve as immediate intermediaries between the client and the database. The MVC application sends one request to the bean to retrieve data, caches the entity bean instances, and sends another request to the bean to post the data and commit. JDeveloper allows you to create stateful session beans as well, but you cannot create data controls from them.

How OracleAS TopLink Technology with POJO Provides Persistent Business Objects

TopLink technology with POJO uses ordinary JavaBeans classes as persistent business objects. You create the classes yourself, implementing everything except their interaction with the database. This interaction is handled by the TopLink runtime, which uses a *descriptor*—an element in the *TopLink deployment descriptor*, an XML file that describes all aspects of TopLink mappings for a set of JavaBeans classes. The descriptor describes the way in which the classes map to database columns, as well as providing information on keys and sequences.

One of the most important features of TopLink is its automation of complex O/R mappings. TopLink technology provides these O/R mappings to allow you to map database objects to arbitrary Java objects, even Java objects that have a substantially different structure than the objects in the database.

TopLink provides the following mappings between fields and database columns:

- *Direct-to-field* mappings, which are the simple mapping between one field and one database column.
- *Type conversion* mappings, which allow more complex conversions between the datatype of the field and that of the database column (for example, a String to a NUMBER or DATE).
- *Object type* mappings, which allow the data stored in the Java objects to be fundamentally different from the database data (for example, you could require a value of “M” in the database to map to a value of “male” in the field, and a value of “F” to map to a value of “female”).
- *Serialized object* mappings, which help you to efficiently map large data objects like BLOBs and multimedia files.
- *Transformation* mappings, which allow you to map several database columns onto a single field.
- *Array* mappings, which map VARRAYs and nested tables into Java arrays.

- *Structure* mappings, which map database structures onto Java classes.

Unlike ADF Business Components, TopLink technology is specifically intended to manage the interaction between an existing Java object framework and the database. For this reason, TopLink does not provide validation or other business rules functionality; such functionality is left up to your object model.

Relationships between JavaBeans classes are handled in the deployment descriptor by *Java object mappings*. As it does with O/R mappings, TopLink has a very flexible system of relationships, allowing you to relate Java objects based on a wide variety of relationships in the database:

- Simple one-to-many relationships, such as that between a customer and all orders placed by that customer
- Simple one-to-one relationships, such as that between a product and its extended description (if these are represented by separate entity object definitions)
- Many-to-many relationships, such as that between products and the warehouses that currently stock them
- One-to-many relationships between mapped objects and unmapped objects (such as Strings)
- *Aggregate object* relationships: one-to-one relationships between JavaBeans classes that require them to map to the same database row
- *Aggregate collection* relationships: one-to-many relationships that can be programmatically defined (as opposed to one-to-many relationships based on attribute matching or foreign keys)
- *Variable one-to-one* relationships, which can relate interfaces (with variable implementation classes) as opposed to JavaBeans
- Relationships based on REFs
- One-to-many or many-to-many relationships based on nested tables

How OracleAS TopLink Technology with POJO Provides Data Access Components

Like Enterprise JavaBeans technology, TopLink technology with POJO does not use actual objects to provide data access components. Instead, it uses *TopLink queries* as data access components.

TopLink queries are elements in the JavaBeans descriptors. When you create a TopLink query, you can specify it in SQL, EJBQL, or TopLink's own structured expression language. The TopLink runtime will use it to search in the TopLink cache for objects matching the query.

Like EJB finder methods, TopLink queries have limits to their complexity. Since they simply return collections of JavaBeans classes mapped to database tables, they cannot implement joins or calculated attributes. They also do not provide the efficiency of the fast-lane reader pattern, although TopLink does have a separate sort of query, called a *report query*, which does. The results of report queries, however, cannot be used to perform DML operations.

How OracleAS TopLink Technology with POJO Provides Service Objects

OracleAS TopLink technology does not provide true service objects. The TopLink deployment descriptor serves to aggregate all the JavaBeans classes, which the MVC portion of the application works with directly.

There is no fixed data model in TopLink. Applications can dynamically traverse Java object mappings to get detail or other related objects from their source objects.

TopLink does maintain a cache. Unlike ADF Business Components or standard EJB technology, the TopLink cache, and a single database transaction, is shared across users. The TopLink runtime automatically merges data to maintain consistency. The cache and transaction are accessed through a Java object called a *TopLink session*.

How Enterprise JavaBeans Technology with TopLink CMP Provides Persistent Business Objects

The TopLink runtime can also act as a CMP provider for EJB entity beans. Using TopLink as your CMP provider has the advantage of automating far more complex O/R mappings. Direct-to-field, type conversion, object type, serialized object, and transformation mappings are all available for EJB entity beans using TopLink CMP.

Like other entity beans, TopLink entity beans are related through CMRs.

How Enterprise JavaBeans Technology with TopLink CMP Provides Data Access Components

Like other CMP providers, the TopLink runtime uses EJB finder methods as data access components. However, the TopLink runtime allows you to specify queries in SQL, EJBQL, or TopLink’s own structured expression language.

How Enterprise JavaBeans Technology with TopLink CMP Provides Service Objects

As a CMP provider, the TopLink runtime does not interact with EJB session beans. Your EJB session beans will work the same way whether or not you use TopLink technology.

Chapter 3, ADF Business Components in Depth

ADF Business Components technology is a fully-featured, XML-based framework for creating business services. ADF Business Components evolved from the Business Components for Java (BC4J) technology distributed with Oracle9*i* JDeveloper and earlier releases. The ADF Business Components runtime library handles most business service functionality, which you can customize declaratively (by changing the XML files using JDeveloper's RAD tools) or programatically (by extending library classes). Oracle ADF Business Components technology:

- Automatically handles O/R mappings and persistence
- Allows you to make complex requests for data retrieval using SQL
- Automatically handles transaction management, including optimistic or pessimistic locking
- Provides a framework for implementing complex business logic
- Automatically implements many J2EE design patterns
- Has a powerful caching and data passivation system for increasing the performance and scalability of applications

All of the above functionality is fully customizable: if you do not like the way ADF Business Components handles O/R mappings, for example, you can override it.

This chapter introduces ADF Business Components technology. After you have read this chapter, you will be familiar with the ADF BC component types and with some basic issues in ADF Business Components design.

Summary

ADF Entity Object Definitions

ADF entity object definitions are business components that encapsulate the business model, including data, rules, and persistence behavior, for items that are used in your application. For example, entity objects can represent:

- Elements of the logical structure of the business, such as product lines, departments, sales, and regions
- Business documents, such as invoices, change orders, and service requests
- Physical items, such as warehouses, employees, and equipment

Entity object definitions map to single objects in the data source. In the vast majority of cases, these are tables, views, synonyms, or snapshots in a database. Advanced programmers can base entity objects on objects from other data sources, such as spreadsheets, XML files, or flat text files.

An entity object definition is the template for *entity object instances*, which are single Java objects representing individual rows in a database table. For example, the entity object definition called "Departments" provides a template for entity object instances that represent individual rows of the DEPARTMENTS table.

An entity object definition can have up to four parts:

- An XML file, which represents the portion of the entity object definition that can be developed declaratively. Most of the information is that needed for O/R mapping, but it can also contain simple validation rules, called *validators*. For many entity objects, the XML file by itself is sufficient.
- An entity object class, which represents individual entity object instances. Entity object classes allow you to write complex business logic in Java, when using XML validators is not sufficient. Entity object classes extend the class `oracle.jbo.server.EntityImpl`. If you do not need custom Java business logic, you need not generate an entity object class—ADF can use `oracle.jbo.server.EntityImpl` directly to represent rows of the data source.

- An entity definition class, which represents the data source object in its entirety. Entity definition classes act as Java wrappers for the XML file, so if you need special handling of the metadata (for example, if you need to change it dynamically), you can add this code in the entity definition class. Entity definition classes extend the class `oracle.jbo.server.EntityDefImpl`. If you do not need custom handling of your metadata, you need not generate an entity definition class—ADF can use `oracle.jbo.server.EntityDefImpl` directly to wrap the metadata.
- An entity collection class, which represents the cache of rows (instances of the entity object class) held in memory for a single user. The vast majority of developers do not need to generate an entity collection class. You should do so only if you want to override ADF Business Components' default caching behavior.

Attributes and Accessors

When entity objects are based on database objects, columns in the database object (such as a database table) map to single *entity object attributes* in the entity object, although the mapping is not necessarily one-to-one. The definitions of these attributes reflect the properties of these columns, such as the columns' data types, column constraints, and precision and scale specifications. When entity objects are based on objects from other data sources, entity object attributes map to "columns" from those objects, as defined by the programmer.

There are two sorts of entity object attributes:

- *Persistent attributes* are those attributes that do map to data source object columns.
- *Transient attributes* are all those attributes that do not map to data source object columns. Transient attributes may be derived from information in a database, and are often used for temporary storage and retrieval.

The values of entity attributes can be read and changed in one of two ways:

- The EntityImpl class provides methods, `getAttribute()` and `setAttribute()`, that accept the attribute's name as a String. For example, if the entity object has an attribute called "DepartmentName," you can access this value by calling `getAttribute("DepartmentName")` or `setAttribute("DepartmentName", "Marketing")`.
- If you generate an entity object class, it will contain typesafe getters and setters for each attribute. For example, if the entity object has an attribute, "DepartmentName," and you have generated an entity object class, you can access this value by calling `getDepartmentName()` or `setDepartmentName("Marketing")`.

Typesafe getters and setters are ideal places to put attribute-level business rules, such as validation logic. They contain calls to the methods `EntityImpl.getAttributeInternal()` and `EntityImpl.setAttributeInternal()` respectively, and by wrapping that call in additional code (such as an if-then block), you can place conditions on attribute change or access, or enforce logic before or after the attribute is accessed or changed.

Validators

In addition to adding validation logic to getters and setters, you can also declaratively attach *validators* to entity attributes, whether or not you have generated the entity object class. JDeveloper comes with four simple validators:

- CompareValidator, which compares an attribute to a value (either a literal value or a value drawn from the data source).
- ListValidator, which checks to see whether an attribute is in a list of values (either a literal list or the results of a query).
- RangeValidator, which checks to see whether an attribute is between two literal values.
- MethodValidator, which can invoke any method in the entity object class which returns a boolean value. Validation is passed if the method returns `true`.

In addition, you can create your own custom validators. These require coding to create initially, but once created, they can be applied declaratively in many different projects. The validator classes must implement the interface`oracle.jbo.server.rules.JbiValidator`, which has three requirements:

- A method, `validateValue()` , which accepts a `java.lang.Object` (the new attribute value) as a parameter and returns a boolean value, `true` or `false`. Generally, you will use this method to return `true` if the attribute value is acceptable and `false` if it is not.
- A method, `vetoableChange()` , which accepts a parameter of type `oracle.jbo.server.util.PropertyChangeEvent`. This is the method that `EntityImpl` calls when the attribute value is changed. Generally, you will use this method to extract the value from the `PropertyChangeEvent` instance, call `validateValue()` , and throw an exception if `validateValue()` returns `false`.
- A field (with accessor methods), `description`. ADF does not use this field, but it is required by the interface.

In addition to these requirements, you can add additional fields to the validation rule. When you apply the rule to an entity attribute, you can customize it declaratively by supplying values for the fields.

The validateEntity() Method

Row-level validation is useful when you need to validate two or more attributes at the same time. You implement this type of validation at the entity level by overriding `EntityImpl.validateEntity()`.

Whenever an instance of an entity object loses its currency (the client is done looking at a particular row), or whenever a client attempts to commit a transaction, `validateEntity()` is called. If `validateEntity()` throws an exception, the entity object is prevented from losing currency until the error is fixed.

You should always include a call to `super.validateEntity()` as part of your extended method. JDeveloper contains tools that will generate a skeleton `validateEntity()` method that does this. You can, however, add additional code before the call to throw exceptions if your validation logic is not satisfied.

Creation and Deletion Logic

Whenever an entity object instance is created or marked for deletion, it calls the method `EntityImpl.create()` or `EntityImpl.remove()`. You can override either or both of these methods to implement business rules that fire when rows are created or deleted.

You should always include a call to `super.create()` or `super.remove()` as part of your extended method. JDeveloper contains tools that will generate skeleton `create()` and `create()` methods that do this. By adding additional logic after these calls, you can perform additional initialization or cleanup tasks.

DML Customization

ADF BC technology will automatically handle DML operations for you, issuing INSERT, UPDATE, or DELETE commands to the database as necessary. However, you can override the method `EntityImpl.doDML()` to implement different persistence behavior (such as using stored procedures or writing to a data source other than a database).

`doDML()` takes as a parameter an `int`, `operation`, representing the DML operation requested. The value of the `int` will always be one of three numbers, each represented by a `final` variable:

- `DML_INSERT`, for row insertion requests
- `DML_DELETE`, for row deletion requests
- `DML_UPDATE`, for row update requests

By checking the value of operation against these three values, you can override the framework's default behavior.

Security

If you want, you can use Oracle ADF Business Components with the Java Authentication and Authorization Service (JAAS) to provide authentication of users of Oracle ADF applications. Oracle ADF Business Components works with both the OracleAS JAAS Provider and with any JAAS-compliant foreign implementation.

Oracle ADF Business Components uses OracleAS Single Sign-On (SSO) through either the Oracle Internet Directory (OID) or a lightweight JAZN-XML file (useful for testing and scenarios where there is a small number of authorized users) to manage identity.

If you elect to use JAAS authorization with your business components, you can attach permissions to entity object attributes to allow only particular users or members of particular groups access (a process called *authorization*). You can set access levels to no access, read-only access, or full access.

You can also use entity object attributes called *history column attributes* to maintain audit trails in the database. These trails contain information about which authenticated users inserted or modified rows and when the changes took place.

ADF Associations

Relationships between entity object definitions are handled by *Oracle ADF associations*, which define a relationship between two Oracle ADF entity object definitions based on sets of entity attributes from each.

Associations map to relationships between single objects in the data source. In the vast majority of cases, these are relationships among tables, views, synonyms, and snapshots in a database. Advanced programmers can use associations to represent relationships within other data sources, such as spreadsheets, XML files, or flat text files.

When the data source is a database, associations often map to foreign key relationships between tables in the database. Although you do not need to actually create a foreign key constraint between tables to create a one-to-one or one-to-many association between the corresponding entity objects, there should at least be an appropriate logical relationship between the tables.

Accessor Attributes

When you create an association between two entity object definitions, you can elect to add accessor attributes to the source entity object definition, the destination entity object definition, or both. These accessor attributes function much like other attributes:

- Their names can be passed as arguments to `EntityImpl.getAttribute()`.
- If you generate an entity object class, a getter method for the accessor attributes will be included in the class.

What is returned by the call to `getAttribute()` or the getter method depends on the cardinality of the association.

Cardinality

Associations can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships. For example, associations can represent:

- The one-to-many relationship between a customer and all orders placed by that customer
- The one-to-one relationship between a product and its extended description (if these are represented by separate entity objects)
- The many-to-many relationship between products and the warehouses that currently stock them

One-to-one and one-to-many associations work much like foreign key relationships: a set of attributes (such as those representing a primary key) of the source entity object are matched with a set of attributes (such as those representing a foreign key) of the destination entity object definition.

Many-to-many associations are effectively the same as two one-to-many relationships involving the source and destination entity object definitions and a third entity object definition, the *intersection*. For example, the many-to-many relationship between products and the warehouses that stock them can be thought of as two one-to-many relationships:

- The one-to-many relationship between products and the warehouse inventory entries that mention them
- The one-to-many relationship between warehouses and their inventory entries

These relationships require three entity objects: one representing products (the source), one representing warehouses (the destination), and one representing warehouse inventory entries (the intersection).

The cardinality of an association affects what is returned by its association accessors:

- Association accessors returning the "one" end of a one-to-many or a one-to-one association return individual entity object instances.
- Association accessors returning the "many" end of a one-to-many or a many-to-many association return row iterators.

Row Iterators

Row iterators are containers of entity object instances or view rows. Although row iterators occur in multiple places in the ADF BC architecture, the row iterators returned by association accessors contain entity object instances.

Row iterators contain a current row pointer that points to one particular entity object instance or view row. This pointer can be moved around and used to extract rows from the iterator.

Row iterators contain a number of methods to help you navigate and extract individual rows from them:

- `next()` advances the current row pointer in the row iterator and returns that row.
- `hasNext()` checks to make sure that the row iterator has more rows after the current row pointer. You can use `next()` and `hasNext()` together to create a loop to cycle through the rows in the iterator.
- `first()` moves the current row pointer to the first row in the iterator and returns that row.
- `last()` moves the current row pointer to the last row in the iterator and returns that row.
- `previous()` steps the current row pointer back one row and returns that row.
- `hasPrevious()` checks to make sure that the row iterator has more rows after the current row pointer. You can use `previous()` and `hasPrevious()` together to create a loop to cycle backwards through the rows in the iterator.

Compositions

A *composition* is an association in which the source object acts as a container for the destination objects. For inserts, updates, and deletes, instances of the destination entity object are considered parts of instances of the source entity object, rather than independent entities that are merely related to them. An example of this sort of relationship is that between a purchase order and the line items in that order. Unlike the relationship between a department and its employees (employees are independently existing entities that merely have membership in a department), line items are truly part of the purchase order, with no existence independent of it.

Making an association into a composition has the following effects:

- It prevents instances of the destination from existing independently of their source. You can have ADF Business Components automatically delete destination instances when the source instance is deleted; alternatively, you can have it throw an exception if a source instance is deleted while it still has destination instances.
- It marks source instances as needing revalidation whenever destination instances are changed.
- It validates destination instances as part of the validation of source instances.

ADF Domains

An *ADF domain* is a special datatype used for Oracle ADF Business Components attributes, such as entity attributes. Oracle ADF Business Components attributes must be objects: they can't be primitive Java types. Attributes can be of standard Java types, such as `java.lang.String`, or they can be special Oracle ADF BC components called *domains*. The Oracle typemap maps all SQL types except VARCHAR2 to domains by default (VARCHAR2 is mapped to `String`).

There are three types of domains:

- *Predefined domains* are wrappers for SQL types such as NUMBER or BLOB.
- *Validation domains* are used to implement business logic at the type level.
- *Oracle object type domains* are used exclusively for wrapping Oracle object types.

Predefined Domains

Predefined domains are Java classes in the ADF BC library that wrap JDBC classes (which, in turn, provide Java wrappers for SQL datatypes such as `NUMBER`). You can use these domains as attribute types when standard Java classes such as `String` are not appropriate.

The primary domains for use against an Oracle database are all in the package `oracle.jbo.domain`. They are listed in the following table.

Domain	JDBC class
`Array`	`oracle.sql.ARRAY`
`BFileDomain`	`oracle.sql.BFILE`
`BlobDomain`	`oracle.sql.BLOB`
`Char`	`oracle.sql.CHAR`
`ClobDomain`	`oracle.sql.CLOB`
`Date`	`oracle.sql.DATE`
`Number`	`oracle.sql.NUMBER`

`Raw`	`oracle.sql.RAW`
`Ref`	`oracle.sql.REF`
`RowID`	`oracle.sql.ROWID`
`Struct`	`oracle.sql.STRUCT`
`Timestamp`	`oracle.sql.TIMESTAMP`

Three of the above domains, `Char`, `Date`, and `Number`, are generic domains that can be used with any implementation of JDBC (and therefore against any JDBC-compliant database).

`oracle.jbo.domain` also contains the `DBSequence` domain. This domain functions like `Number`, but it maintains a temporary sequence value in memory until the data is posted, thus preventing the wasting of database sequence numbers.

In addition to `oracle.jbo.domain`, the ADF BC library also contains the package `oracle.ord.im`. This package contains domains that allow ADF BC to integrate with Oracle *inter*Media types for multimedia applications.

Oracle Object Type Domains

Oracle object types can be represented in either of two ways:

- If you use the Oracle object type as a type for object tables only, you can simply use an entity object definition to represent those tables. The entity attributes will match the columns in the object type.
- If you use the Oracle object type as a type for object columns, the object type will be represented as a custom domain.

Domains for Oracle object types have attributes representing each column in the object type. You can access column values using getter and setter methods, much as you do for entity object attributes.

Validation Domains

You can create custom domains to provide type-level validation. These domains wrap other types that could be used as attribute values (such as predefined domains or standard Java classes like `String`). After creating such domains, you can use them in place of the other datatype. For example, suppose the Employees entity object definition has an attribute, Email, of type `String`. You could create a domain, `EmailDomain`, that wraps `String`, and use it as the type of Email instead.

Validation domains contain a method, `validate()`, that is called whenever the domain is instantiated. You can write code in this method to perform tests and throw an exception if the tests are not passed. Doing so has the effect of adding validation logic to all attributes that use the domain as their type.

ADF View Object Definitions

ADF view object definitions are business components that collect data from the data source, shape that data for use by clients, and allow clients to change that data in the Oracle ADF Business Components cache. For example, a view object definition can gather all the information needed to:

- Populate a single table element in a form
- Create and process an insert or edit form
- Create an LOV for populating a dropdown list

View object definitions must have a mechanism for retrieving data from the data source. Usually, the data source is a database, and the mechanism is a SQL query. Oracle ADF Business Components can automatically use JDBC to pass this query to the database and receive the result.

When view object definitions use a SQL query, query columns map to *view attributes* in the view object definition. The definitions of these attributes reflect the properties of these columns, such as the columns' data types and precision and scale specifications. When view object definitions use other data sources, view object attributes map to "columns" of data from those data sources, as defined by the programmer.

A view object definition is a template for *view object instances*, which represent particular caches of rows of data. Different users will always use different view object instances, but the same user may also use multiple view object instances if they need separately maintained caches from the same query.

A view object definition can have up to four parts:

- An XML file, which represents the portion of the view object definition that can be developed declaratively. This information consists of the mechanism (usually a SQL query) that the view object uses to retrieve data from the data source, and the way in which the columns of the SQL query map to entity attributes (which handle actual O/R mapping). For many view object definitions, the XML file by itself is sufficient.
- A view object class, which represents an individual view object instance. View object classes allow you to write custom methods that affect multiple rows in a query. View object classes extend the class `oracle.jbo.server.ViewObjectImpl`. If you do not need to write custom view object methods, you need not generate an entity object class—ADF can use `oracle.jbo.server.ViewObjectImpl` directly to represent instances of the query result set.
- A view row class, which represents individual rows of the query result. View row classes allow you to write custom methods that affect a single row of data, and they provide typesafe accessors to retrieve and change data. View row classes extend the class `oracle.jbo.server.ViewRowImpl`. If you do not need custom row-level methods or typesafe accessors, you need not generate a view row class—ADF can use `ViewRowImpl` directly to represent view rows.
- A view definition class, which represents the query itself. View definition classes act as Java wrappers for the XML file, so if you need special handling of the metadata (for example, if you need to change it dynamically), you can add this code in the view definition class. View definition classes extend the class `oracle.jbo.server.ViewDefImpl`. If you do not need

custom handling of your metadata, you need not generate a view definition class—ADF can use `ViewDefImpl` directly to wrap the metadata.

Attribute Mappings

Like entity attributes, the values of view attributes can be read or changed using the methods `getAttribute()` and `setAttribute()` in the ViewRowImpl class or by using generated getters and setters in a custom view row class.

There are two different types of view attributes, however, for which these accesssor methods function quite differently:

- *SQL-only view attributes* are not mapped to entity attributes. For these attributes, the accessor methods read values from and make changes to data in the view object instance's cache of view rows.
- *Entity-derived view attributes* are mapped to attributes in an entity object definition. For these attributes, the accessor methods will call `getAttribute()` and `setAttribute()` on the relevant entity object instance. The data will be changed within the entity collection's cache of entity object instances, not within the view object instance's cache of view rows.

Because entity object definitions handle DML operations, attributes that will be used to make changes to the database must be entity-derived. However, if a view object definition will be used for data retrieval only, there is an advantage to making all its attributes SQL-only: such view object definitions, called *SQL-only view object definitions*, bypass the entity collection's cache entirely, avoiding the overhead and resources required to create entity object instances.

Navigating Through Result Sets

View object instances are row iterators. In particular, they are row iterators of view rows.

Like other row iterators, view object instances contain a current row pointer that points to one particular view row. This pointer can be moved around and used to extract rows from the iterator.

Row iterators contain a number of methods to help you navigate and extract individual rows from them:

- `next()` advances the current row pointer in the row iterator and returns that row.
- `hasNext()` checks to make sure that the row iterator has more rows after the current row pointer. You can use `next()` and `hasNext()` together to create a loop to cycle through the rows in the iterator.
- `First()` moves the current row pointer to the first row in the iterator and returns that row.
- `Last()` moves the current row pointer to the last row in the iterator and returns that row.
- `Previous()` steps the current row pointer back one row and returns that row.
- `hasPrevious()` checks to make sure that the row iterator has more rows after the current row pointer. You can use `previous()` and `hasPrevious()` together to create a loop to cycle backwards through the rows in the iterator.

Creating and Deleting Rows

`ViewObjectImpl` also contains methods to create rows:

- `createRow()` creates a view row appropriate to the view object definition.
- `insertRow()` inserts the row into the view cache.

You can mark a row for deletion by calling `Row.remove()` or `ViewObjectImpl.removeCurrentRow()`.

Keys

A *key* is a set of attributes that allow you to quickly retrieve one or more rows from a view object instance's query result. Persistent view object attributes based on primary keys are automatically part of the view object's key; you can make other attributes part of the view object's key as well.

You can use an array containing a partial or complete list of attribute values to set up an object of type `oracle.jbo.Key`. You can then pass this object into the method `ViewObjectImpl.findByKey()` to return an array of rows that match the key values.

View Criteria

View criteria are structured criteria that you can use to create searches.

View criteria are collections of *view criteria rows*. A view criteria row specifies query-by-example requirements for one or more view object attributes. A view row *matches* if it meets all of the requirements.

When you apply view criteria to a view object instance, the query is restricted to return those view rows that match at least one of the view criteria rows. Effectively, therefore, view criteria assemble a WHERE clause in conjunctive normal form: the WHERE clause is a disjunction of conjunctions of query-by-example requirements.

View criteria are implemented by the class `oracle.jbo.ViewCriteria`; view criteria rows, by the class `oracle.jbo.ViewCriteriaRow`.

ADF View Link Definitions

Relationships between view object definitions are handled by *Oracle ADF view link definitions*, which define a relationship between two Oracle ADF view object definitions based on sets of entity attributes from each. Like associations, these can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships.

Individual instances of view objects can also be related by individual instances of view links, which create a master-detail relationship between the query result sets. For example, suppose that you have view object definitions representing a query for department information and a query for employee information, and a view link between the view objects representing the relationship between a department and its employees. If an instance of the former view object definition, `allDepartments`, is related to an instance of the latter, `employeesInDepartment`, by an instance of the view link, those instances will be synchronized: whenever a particular row of `allDepartments` is selected, `employeesInDepartment` will only display details of that row.

Accessor Attributes

When you create a view link definition between two view object definitions, you can elect to add accessor attributes to the source view object definition, the destination view object definition, or both. These accessor attributes function much like the accessor attributes to associations:

- Their names can be passed as arguments to `ViewObjectImpl.getAttribute()`.
- If you generate a view row class, a getter method for the accessor attributes will be included in the class.
- The accessor method will return a view row or a row iterator, depending on the cardinality of the view link definition.

An accessor attribute returns a row or row iterator that is static, not one that maintains a synchronized master-detail relationship. For example, suppose DepartmentEmployees is an accessor attribute that returns rows of EmployeesView from rows of DepartmentView. Suppose you execute the following code on the current row of DepartmentView:

```
RowIterator details = current.getAttribute("DepartmentEmployees");
```

Then suppose the current row of DepartmentView changes. The row iterator in `details` will not change: it will still contain details of the original row.

To maintain a synchronized master-detail relationship, you should use view link instances in an application module instance.

Cardinality

Like associations, view link definitions can be one-to-one, one-to-many, or many-to-many. One-to-one and one-to-many view link definitions can either be based on associations or they can use attribute matching the way associations do. Many-to-many view link definitions must be based on many-to-many associations.

ADF Application Module Definitions

Oracle ADF application module definitions are business components that represent particular application tasks. The application module definition provides a data model for the task by aggregating the view object and view link instances required for the task. It also provides services that help the client accomplish the task. For example, an application module can represent and assist with tasks such as:

- Updating customer information
- Creating a new order
- Processing salary increases

The most important feature of an application module is its *data model*—the view object and view link instances it contains. These specify what data the client will have access to, and what relationships hold within that data.

You can use application module definitions in two different ways:

- As a service object, in which case each instance of the MVC application has access to one instance of the application module. These *root-level* application module instances control ADF BC transaction objects, which in turn control the entity and view caches.
- As a reusable object for nesting, in which case you can create a data model and service methods on it and then nest one of its instances in other application module definitions. Those application module definitions can, in turn, access the nested module's methods and data model. Nested application modules share the root-level application module's transaction.

An application module definition can have one or two parts:

- An XML file, which represents the portion of the application that can be developed declaratively: the view object and view link instances that the application module contains and the way in which they are related. For many application modules, the XML file by itself is sufficient.
- An application module class, which lets you write custom code such as service methods that an MVC application can invoke for batch data handling. Application module classes extend the class `oracle.jbo.server.ApplicationModuleImpl`. If you do not need to write custom service methods, you need not generate an application module class—ADF can use `oracle.jbo.server.ApplicationModuleImpl` directly.

View Object and View Link Instances

One of the primary functions of an application module definition is to provide applications with the data they need to complete a specific task. This data can be represented by a tree—the application module's data model—which, in turn, contains view object and view link instances.

A *view object instance* manages a single cache of retrieved data. View object instances use data retrieval mechanisms (usually SQL queries) provided in view object definitions. However, these mechanisms can be customized on an instance level. In other words, by dynamically adding or changing clauses in the query of one view object instance, you do not automatically make similar changes in other view object instances, even if the instances share a definition. Moreover, executing a query on one view object instance does not automatically execute the others' queries.

All view object instances have a name assigned to them when they are first added to a data model. This name is used by clients and service methods to access the instance and the data stored in the instance's cache. This name is not necessarily related to the view object definition name. For example, the same data model could contain two view object instances, called AllOrders and OrdersForCustomer, based on the same view object definition, called OrdersView.

A *view link instance* provides a master-detail relationship between view object instances. View link instances are based on view link definitions, which relate the relevant view object definitions.

Adding a view link instance to the data model puts two view object instances in a master-detail relationship; removing the view link makes the detail view object instance completely independent. This, rather than view link accessor attributes, is the way to dynamically maintain master-detail relationships: the detail view object instance’s cache will contain only those rows that are details of the master view object instance’s current row. When that row changes, the detail cache will automatically change as well.

Transactions

A *transaction object* is an Oracle ADF Business Components object that represents a database transaction. A transaction object maintains pointers to entity and view caches; it maintains a database connection; and it is responsible for post, commit, and rollback operations.

Unlike database transactions, transaction objects survive commit and rollback operations. Therefore, a single transaction object can correspond to several database transactions over its lifetime.

In general, there is exactly one transaction object per root-level (non-nested) application module instance. If you access a transaction object from a root-level application module instance, any of its nested application modules, or any of the entity object instances in its caches, you will retrieve the same object. If you access transaction objects from two root-level application module instances, you will retrieve different objects.

Service Methods

Service methods are methods on ADF application module definitions that perform complex operations on data. Applications can call these methods in a single network round-trip, saving processing on the client and reducing network chattiness.

Service methods are implemented in an application module’s class, and exposed on tier-independent interfaces that allow clients to access the application module consistently, whether it is deployed locally or as an EJB session bean. Inside the service method, you can do any of the following:

- Dynamically add view object and view link instances to the data model
- Remove view object and view link instances from the data model
- Find view object instances and perform operations on their row sets
- Retrieve and manipulate transaction objects

Application Module Pooling

An *application module pool* is a resource manager for top-level application module instances. Since storing the view object and entity object caches associated with a transaction and application module instance can be expensive, the application module pool maintains some instances in memory and reuses others. There is one application module pool for each application module definition: all instances of that application module are stored in the pool.

When an application is actively using an application module instance (for example, during a Struts data action), that application module instance is described as *checked out*. As soon as the application is done with the instance, the instance is *checked in* to the pool. When the application needs the instance again, it will attempt to check it out again.

When an application requests an application module instance for the first time, the application module pool checks to see how many instances it already contains. If this number is below a parameter called the *recycle threshold*, the pool creates a new instance for the application.

If the application module pool contains a number of instances equal to or higher than the recycle threshold, it recycles one of the instances. It does so using the following process:

1. The pool finds the application module instance that has been checked in for the longest time.
2. The pool writes a redo log of the instance’s transaction to the database table PS_TXN.
3. The instance clears its caches.
4. The pool passes the newly cleared caches to the application.

By default, the recycle threshold is 10. Many applications will perform better with a higher recycle threshold: setting the threshold is a balance between not having too much data in memory (which can degrade the performance of the application server) and not recycling too many times (because recycling is a time-consuming process). Trial and error using a load tester is often the best way to find this balance.

ADF Business Components Design Decisions

Two central design decisions face developers who are using ADF Business Components technology as their business services: where to put the code that implements their applications’ business rules and whether to base view object definitions on entity object definitions or to make them SQL-only.

Where to Implement Business Rules

Business rules can be provided at multiple levels of an application—in the database, the view, or the business services layer. There are times where each of these locations is appropriate.

Adding business rules to the database, in the form of triggers or stored procedures, provides the maximum level of robustness. These business rules are guaranteed to be available and respected by any application—even by SQL commands run directly from a SQL*Plus prompt. However, business rules coded in the database are not highly responsive. They do not fire until data is posted to the database, which either requires waiting for an explicit post command or requires posting data after every change, which will degrade performance by requiring excessive JDBC round-trips. In addition, adding business rules to the database requires the database to perform tasks other than handling data, which reduces its efficiency and your application’s modularity. Finally, adding business rules to the database requires you to integrate your Java or web application with business logic written in PL/SQL code.

Adding business rules to the view layer, in Java for Java client applications or JavaScript for web applications, provides the maximum level of responsiveness. Business rules that trigger as each character is typed into a field, for example, or as a mouse pointer wanders over a graphical image, must be implemented at the view level. However, business rules added to the view layer are not robust. If users access the data through any other user interface, business logic added to the view layer will not be available or enforced.

Adding business rules to ADF BC components is a compromise between these alternatives. Business rules in ADF BC components are more responsive than business rules coded in the database, because they are enforced as soon as changes are made to Java objects in memory, and they avoid the other disadvantages of adding business rules to the database. They are more robust than business rules coded in the view layer, because they will be enforced by any application that uses the components.

Your most critical business rules should be implemented in the database, or redundantly in the database and business services (for increased responsiveness and easier Java integration at the cost of some productivity). Business rules that require truly immediate responsiveness must be implemented in the view layer. The remainder of your business rules can be implemented in ADF Business Components. For this sort of business rule, the hooks provided by ADF BC provide a distinct advantage over other business services technologies.

If you decide to implement a business rule in ADF BC components, you should generally implement it in entity object definitions. Because entity object definitions perform all DML operations, any changes that will affect the database will trigger any appropriate business rules in entity object definitions. Business rules implemented in view object definitions are less robust—they will not be invoked when changes are made through other view object definitions, even those based upon the same entity object definition.

As discussed earlier in this chapter, there are still several choices for where to put business rules in entity object definitions:

- In the entity object class
- In validators
- In domains

If you have business rules other than validation rules, they must be placed in the entity object class. Validation rules can be placed in any of these locations, but there are reasons to choose one over another:

Validation code in the entity object class cannot easily be reused by other entity object definitions. However, it is the easiest way to initially create validation code—simply edit the entity object class and add a method. It’s also the only place you can put validation logic that traverses associations.

Validators are highly reusable: they can be reused with multiple entity objects, on attributes of differing type, and customized declaratively. However, they take more work to set up initially than the other forms of validation logic—you must create a validator class, implement the `JbiValidator` interface, create a property editor if you want to use one to customize the validator, and register the validator with JDeveloper.

Domains represent a compromise between these two options. Creating a validation domain involves creating the domain class and writing the code, but it is still significantly more straightforward than creating a validator. It is also reusable across many attributes in many entity object definitions, although all the attributes must have the same underlying type.

Some architects of large projects have reported that they have found it necessary to impose a single location for business rules. They have found that having some business rules in the entity object class, some in domains, and some in validators makes maintenance considerably more difficult.

Whether to Use Entity Object Definitions

If you want to be able to make changes to the database through a view object definition, you must base it on an entity object definition, because entity object definitions handle all DML operations. Basing view object definitions on entity object definitions has the following other advantages as well:

- If the view object definition's query is a join query (such as SELECT * FROM DEPARTMENTS, EMPLOYEES WHERE DEPARTMENTS.DEPARTMENT_ID=EMPLOYEES.DEPARTMENT_ID), each row from the master table will appear in many rows of the query result set. If you create an entity object definition for the master table and use it in the view object definition, the data from each row in the master table will be stored only once, in the entity cache. If you do not use an entity object definition, the data will be stored redundantly, in the view cache, for each row in the query result set.
- Each view object instance maintains its own view cache. If multiple view object instances query data from the same table, and they use an entity object definition to represent the table, the data will need to be stored only once. If they do not use an entity object definition to represent the table, the data must be stored in each view cache.
- Two entity-derived view attributes mapped to the same entity attribute will show synchronized values. If you call `setAttribute()` on one view row to change an attribute value for an entity-derived attribute, and `getAttribute()` on another to read the value of an attribute mapped to the same entity attribute, `getAttribute()` will return the changed value. If the attributes are SQL-only, they will not be synchronized in this way.

If you need to perform DML operations with a view object definition, you should definitely base it on entity object definitions, and if any of the above considerations apply to the view object definition, you should at least consider the option. Otherwise, you should generally not base view object definitions on entity object definitions—this saves time and resources by not creating entity object instances.

Chapter 4, Overview of the Oracle ADF Model Layer

The Oracle Application Development Framework (Oracle ADF) in JDeveloper supports data controls based on Java classes, EJB session beans, web services, and Oracle ADF application modules. The Oracle ADF components, including data controls and data bindings, are configured using XML metadata. This capability allows a small set of framework base classes to handle most of the application development needs without coding. In JDeveloper, you use the design time to expose desired business service data sources in the Oracle ADF model layer. The design time tools lets all developers visualize the exposed business services in a uniform way.

This chapter provides an overview of how you can work with JDeveloper to create the Oracle ADF data controls and data binding objects of the ADF model layer. Creating the Oracle ADF model layer objects is a preliminary step to working with the data binding objects in the application controller and view layers, as is described in Chapter 5, Overview of Oracle ADF Integration with Struts and Chapter 6, Overview of Oracle ADF Data Binding in View Technologies.

Summary

Role of the Model Layer

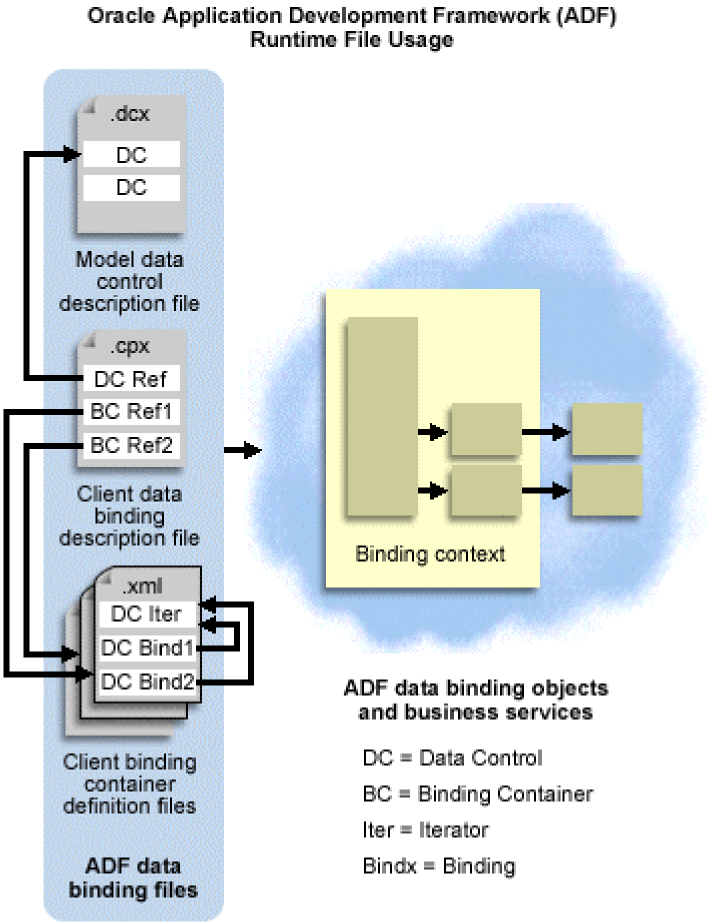
In a J2EE application that uses the Oracle ADF model, model data is surfaced to the view and controller layers through data control objects implemented for each type of data provider. A model-specific set of data controls:

- Manages the J2EE application's connection to the data provider
- Presents bindings and binding containers to the J2EE application client or controller

Once the data is surfaced by data control adapter classes, interaction with the business service by the J2EE application is possible by any method that acts on or retrieves data from data objects and collections (also described as rows and row sets).

About MetaData for the Oracle ADF Binding Context

Using the JDeveloper design time, you create a set of XML files that declaratively define the Oracle ADF data controls and data bindings. At runtime, the application creates the Oracle ADF binding context from the files in the application, as shown in the following diagram.


Oracle Application Development Framework (ADF)
Runtime File Usage
.dcx
DC
DC
Model data control description file
.cpx
DC Ref
BC Ref1
BC Ref2
Client data binding description file
.xml
DC Iter
DC Bind1
DC Bind2
Client binding container definition files
ADF data binding files
Binding context
ADF data binding objects and business services
DC = Data Control
BC = Binding Container
Iter = Iterator
Bindx = Binding

The client binding description file (`.cpx`) references the binding definitions in the client binding container definition files (`.xml`). The `.cpx` file also contains a reference to the data control description file (`.dcx`), which specifies the data control factory to use for a specific business service.

The design time supports creating these files through your interaction with the Data Control Palette, a visual editor, the Structure window, and the Property Inspector. After you use the Data Control Palette to insert a databound UI component into the displayed web page or JClient panel, the binding definition is created in the document's binding definition file. Additionally, source code is added to the document that references the binding objects, which the application makes available through the Oracle ADF binding context `bindings` namespace.

For more information about the generated project files and design time tools, see Chapter 6, Overview of Oracle ADF Data Binding in View Technologies.

Oracle ADF Model API Overview

Specifically, these objects in the ADF model layer provide runtime access to business services:

- Binding context object (`oracle.adf.model.BindingContext`)

  Defines a common namespace for use by the client application and allows all model objects to be exposed through a convenient root name. Each web page or Java panel registers with the binding context using the definition for the variable name `binding`.

- Data control interfaces (`oracle.adf.model.binding.DCDataControl`)

  Provides the client application with an interface into the model objects. One data control is needed to wrap the model objects of each business service configuration. Also, in the case of a JavaBeans model object, provides direct access to the native object, when programmatic access is desired.

- Binding container objects (`oracle.adf.model.binding.DCBindingContainer`)

  Defines a container for data binding objects, including iterator bindings and control bindings. One binding container is created for each web page or Java panel, but may be reused in multiple pages and panels when they share the same data. The binding container object also lets you specify whether the page or panel is to be used in data entry mode or query mode.

- Iterator binding objects (`oracle.adf.model.binding.DCIteratorBinding`)

  Handles the events generated from the associated business service row iterator and sends the current row to individual control bindings to display current data. Iterator bindings can specify the number of rows to display in a web page.

- Control binding objects (`oracle.jbo.uicli.binding.JUControlBinding`)

  Defines the way a specific UI component interacts with the corresponding business service. For example, depending upon the control binding selection, a text field may be bound to an Oracle ADF Business Components view object attribute and display as an editable field or a static label. Or, if the business service attribute defines an enumerated list of values, the bound UI component might display a static list, a single-selection list, or a multi-selection list. Other more complex UI components, such as table and graphs, are also supported by their own control bindings.

- Action binding objects (`oracle.jbo.uicli.binding.JUCtrlActionBinding`)

  At runtime, when the user initiates the action, using a button control, the action binding accesses the Oracle ADF binding context and initiates the specified action on the data objects of the selected collection. Action bindings can take parameters that will be passed to the controller to handle.

Benefiting from the Oracle ADF Model Layer

Role of the Oracle ADF Data Controls

Oracle ADF data controls provide an abstraction of the business service's data model. The Oracle ADF data controls provide a consistent mechanism for clients and web application controllers to access data and actions defined by these diverse data-provider technologies:

- Oracle ADF Business Components
- JavaBeans, including TopLink Plain Old Java Objects (TopLink POJO)
- EJB session beans
- Web services

For most of these, the data controls are implemented by a thin layer of adapter classes. The exception is Oracle ADF Business Components technology, which implements the data controls directly in the component classes.

Oracle ADF data controls are represented on the Data Control Palette, where they can be added to the view as UI controls or to the controller as operations.

The remainder of this section provides an overview of the base features of the provided data controls. The following terms have these definitions:

- *Business service*: Any JavaBean that publishes business objects and provides methods that manipulate business objects. Examples of business services include web services, EJB session beans, or any Java class being used as an interface to some functionality.
- *Business object*: A JavaBean that models a business entity. Business objects are typically persisted to a data source. Examples of business objects in an order entry application may include Order, Customer, and Product.
- *Data collection*: A collection of business objects. A collection may be an instance of `java.util.Collection`, `java.util.Iterator`, or `java array`, or a single business object that is treated as a collection of one.

Business Object Access Services

The business object access services provide built-in Oracle ADF support for binding to data collections and business objects using a business service or a business object. The business object access services are:

- Business collection iteration services

  The Oracle ADF data controls provide an Iterator pattern implementation that may be used with any business collection. You may use built-in operations on this iterator binding implementation to display or iterate business objects. Examples of operations include first, next, previous, and last.

- Find and refresh services

  The Oracle ADF model provides features for defining search parameters for a page model or an ADF model layer iterator. You may use built-in operations to toggle the Oracle ADF model find mode for defining search parameters and to refresh the Oracle ADF model layer iterator bindings.

- Business object property services

  The Oracle ADF model provides features for accessing JavaBeans-style properties of business objects. You may use built-in operations to bind to business object properties. A special class of business object properties includes properties whose accessor methods (like `Customer.getOrders()`) access business collections. These properties are referred to as *accessor attribute properties* or *accessor attributes*.

- Method invocation services

  The Oracle ADF model provides features for invoking business object and business service methods. You may use built-in operations to invoke methods and to bind iterators to the results of operations.

Transaction Services

The transaction services provide built-in Oracle ADF model layer support for mutating business objects. Please note that the Oracle ADF model transaction services do not provide a transaction system itself. Instead, the Oracle ADF model transaction services provide a notification system that individual Oracle ADF data controls integrate with. The presence of a notification system allows custom Oracle ADF data controls to be created that can integrate a model transaction service with the Oracle ADF model layer. ‘

The transaction services are:

- Business object services

  The Oracle ADF model provides features for adding business objects to a collection, for removing business objects from a collection, and for mutating the JavaBeans-style properties of business objects. You may use built-in operations to invoke these services, and the Oracle ADF model layer will automatically notify the Oracle ADF data control of a create, remove, or update event.

- Transaction demarcation services

  The Oracle ADF model provides features for committing and rolling back transactions. You may use built-in operations to notify the Oracle ADF data control to commit or roll back transaction changes.

State Management Services

The state management services provide built-in Oracle ADF model layer support for managing Oracle ADF data control user state. The state management services are:

- Data control lifecycle notifications

  The Oracle ADF model provides Oracle ADF data control lifecycle notifications to the Oracle ADF data control. The individual data controls implement these notifications to passivate or activate the business service user state.

- State distribution services

  The Oracle ADF model provides features to distribute state or to fail over state to other tiers.

For further details about the design time for working with the Oracle ADF model layer and the code generated by the design time, see Creating the Oracle ADF Model Layer in JDeveloper.

Role of the Oracle ADF Bindings

Oracle ADF provides several types of binding objects to support the attributes and operations exposed by the Oracle ADF data controls for a particular business object:

- Iterator binding, one per accessor attribute that your page or panel displays. Iterates over the business objects of the data collection and maintains the row currency and state.
- Value bindings, one for each databound UI component. Provides access to data.
- Action binding, one for each button component. Provides access to operations defined by the business object.

You create these bindings in JDeveloper through your interaction with the Data Control Palette, a visual editor, the Structure window, and the Property Inspector. You can view the binding in your application in any one of several ways:

- In the source code view of a web page, where binding references appear in expressions that get evaluated at runtime using the expression language (EL) features. In the code view, the expression looks like this:

  ```
  <c:forEach var="rv" items="${bindings.DataBindingObject.theCollectionProperty}"
  ```

  The data binding object accesses the Oracle ADF binding container through the `bindings` variable, which is a variable that specifies the namespace for the binding containers in the Oracle ADF binding context. The binding container is initialized for Model 2–style web applications in the Oracle ADF class `oracle.adf.controller.struts.actions.DataAction`.

Note: Although a data action class does not appear in your project, you can see that the action mapping for data action appears in the `struts-config.xml` file, which defines the `modelReference` property to initialize the binding container from the *`pageName`*`UIModel.xml` file.

- In the code view of a JClient panel or form, where the `setModel()` method call on the UI component initializes the data binding object and accesses the binding container through the ADF binding context (specified by the `setBindingContext()` method call on the data panel).

  The binding container is initialized from the *`PanelName`*`UIModel.xml` file in the JClient project when the panel binding is created (by the `JUPanelBinding panelBinding` constructor).

- In the Structure window for either client type (web page or JClient panel), where the data binding objects defined for each UI component in your view document appear in the (**UI Model**) tab. To edit the declarative properties of the binding definitions, right-click and choose **Edit**. To understand what values you can set for runtime-only properties, select the binding node and press F1.

Generic Runtime Properties for All Oracle ADF Bindings

When you create a databound component using the Data Control Palette, JDeveloper adds metadata to the `UIModel.xml` file defined for your view-controller project. The metadata determines the default display characteristics of each databound UI component and sets properties of the binding object at runtime. You can work with the data binding metadata through:

- Design time editors that you display on specific bindings in the **UI Model** tab of the Structure window

Additionally, each binding's implementing class defines methods that give you access to runtime information about the binding. You can work with runtime-specific properties through:

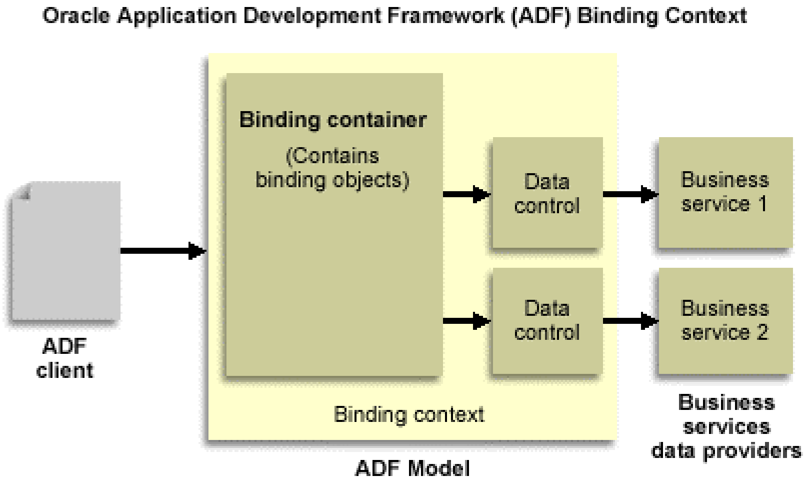
- EL expressions that you insert into the HTML of your web page when you want to access the runtime properties of the value binding object

The `oracle.jbo.uicli.binding.DCControlBinding` class provides accessor methods for these properties that are accessible by all value bindings:

- **error** returns any cached exception which was raised during the invocation of the method or action to which the binding is bound.
- **fullName** returns the fully qualified name of the binding object in the Oracle ADF binding context.
- **iteratorBinding** returns the iterator binding that provides access to the data collection.
- **name** returns the name of the binding object in the context of the binding container to which it is registered.
- **rowKeyStr** returns the key of the data collection. The key specifies the location of the data object and is returned in its String format.

Oracle ADF Data Control Runtime Integration with Business Services

At runtime, the interaction with the business services initiated from the client or controller is managed by the application through a single object known as the *Oracle ADF model binding context*. The Oracle ADF binding context is a container object that defines a hierarchy of data controls and data binding objects derived from the Oracle ADF model layer, as shown in the following figure.


Oracle Application Development Framework (ADF) Binding Context
Binding container
(Contains binding objects)
Data control
Data control
Business service 1
Business service 2
ADF client
Binding context
ADF Model
Business services data providers

The binding context defines these hierarchies, whose objects reference one another in order to allow the Oracle ADF model to service the controller and view layers.

The hierarchy for the data controls looks like this:

- binding context -> data controls

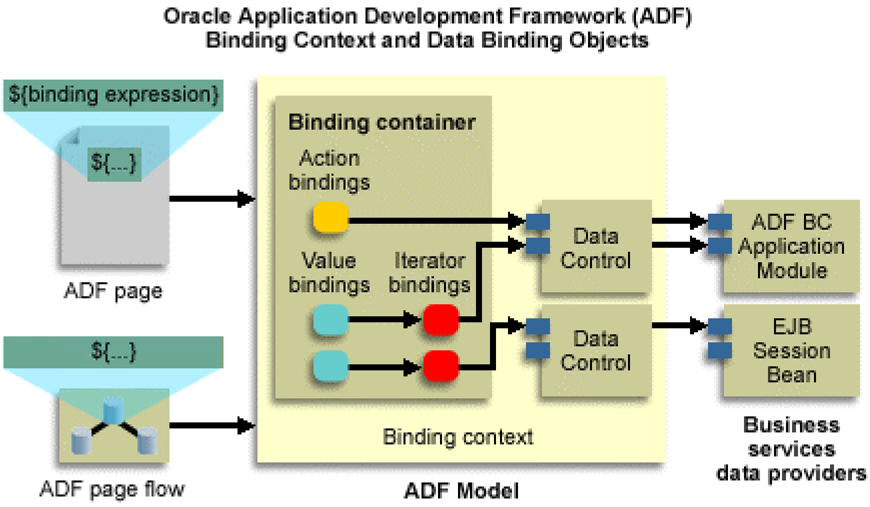
The hierarchy for the data binding objects, which also starts from the same parent container, looks like this:

- binding context -> binding container -> iterator bindings

and, additionally:

- binding context -> binding container -> data control binding -> value bindings and action bindings

The following diagram depicts binding container containment hierarchy in detail. Note that the definition of the binding context and the data binding objects available to the application are specified by XML-based files at the level of the client project. For details about the client project, see Chapter 6, Overview of Oracle ADF Data Binding in View Technologies.



Through the iterator binding's object hierarchy path, the iterator binding accesses the data collection and iterates over its data objects. In contrast, value binding objects allow UI components to display or update the current data object of the collection being iterated over.

For example, in the source of the web page or Java panel, a UI component's value binding definition references the page or panel's iterator binding object. The iterator binding object, in turn, references the data controls, which completes the data-access path and allows the Oracle ADF model to traverse the object hierarchy:

1. Starting from an event (such as rendering the data) that is received by the control binding
2. All the way back to the data controls, specified by the binding context, that interact finally with business services

Note: The above description has been simplified to include both web applications and Java client applications. In an actual web application, the controller layer is responsible for populating the Oracle ADF model binding context prior to dispatching to the view layer. Then, following the pull model for interactive web applications, the view layer pulls the data to render values in the page.

Creating the Oracle ADF Model Layer in JDeveloper

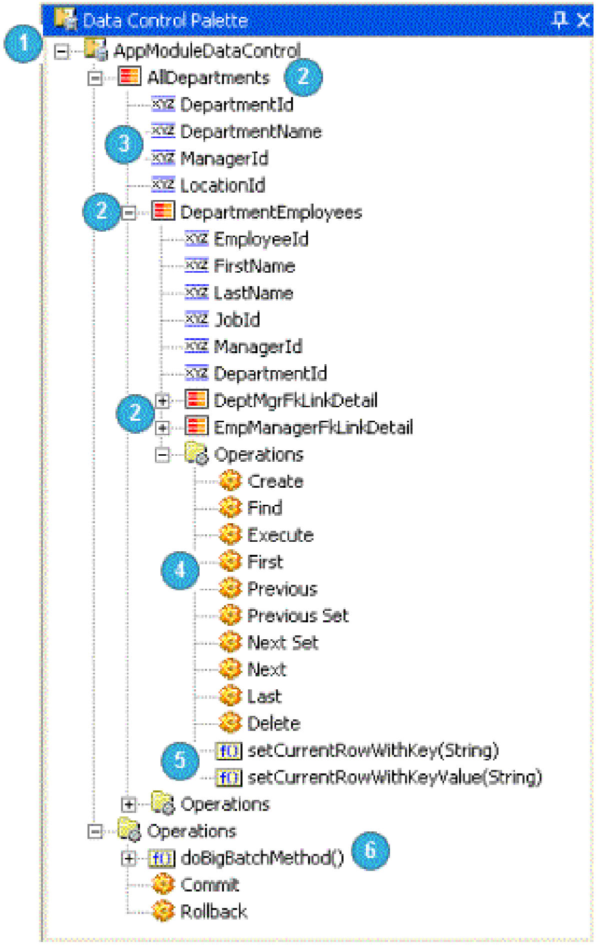
Oracle ADF data controls permit the application client to access business services defined by your application's model object layer. Business services can be any collection, value, or action that your model project defines. At runtime, the Oracle ADF binding context is initialized with the data control definition to permit databound UI components to access the business services.

Before you create data controls, ensure that you have created (or otherwise have access to) business services.

Oracle ADF Business Components as Data Controls

Note: You do not need to create data controls for Oracle ADF Business Components—Oracle ADF BC data model components already implement the data control interfaces.

If you use Oracle ADF Business Components as your business service technology, your data model components will be exposed in the model layer as Oracle ADF data controls, as shown in the following figure.



1 The application module

This node represents the top-level application module itself (nested application modules are represented by the same icon). Directly under this node, you can find nested application modules, top-level view object instances, and application module–level methods.

2 Row sets

These nodes represent row sets—usually view object instances in the data model, but also the row sets returned by view link accessors. In this case, **AllDepartments** represents a top-level view object instance, **DepartmentEmployees** represents a detail instance of **AllDepartments**, and both **DeptMgrFkLinkDetail** and **EmpMgrFkLinkDetail** represent the static row sets returned by view link accessors of the same name. For more information on view links, see the JDeveloper help system.

3 Attributes

These nodes represent individual view object attributes.

4 Standard operations

These nodes represent standard row set–level operations. The application module has similar nodes that represent transaction-level operations. The operations available for a row set include previous set and next set, which scroll through the row set based on range size. For more information, see Summary of Oracle ADF Data Control Operations.

5 Special navigation operations

These nodes allow you to set the row set's current row to a particular displayed row.

6 Custom method

Exported custom service methods will appear on the Data Control Palette.

Oracle ADF Data Controls for EJB Components

If you use EJB technology as your business service technology, model information will be exposed to the view and controller layers through Oracle ADF data control interfaces implemented by thin, Oracle-provided adapter classes.

To create adapter classes and data controls:

1. In the navigator, select the **Model** project which contains the model objects you want to expose to the application client.
2. In the model project, right-click a stateless session bean and choose **Create Data Control**.

   OR

   Drag the session bean from the Application Navigator onto the Data Control Palette to create the data controls.

   Note: You can create data controls only from stateless session beans.

3. If your bean defines attributes or methods that return a `Collection` type, you must specify the return type for the data control.

JDeveloper adds the data control definition file (`DataControls.dcx`) to the model project. The `.dcx` file identifies the Oracle ADF model layer adapter classes that facilitate the interaction between the client and the available business services.

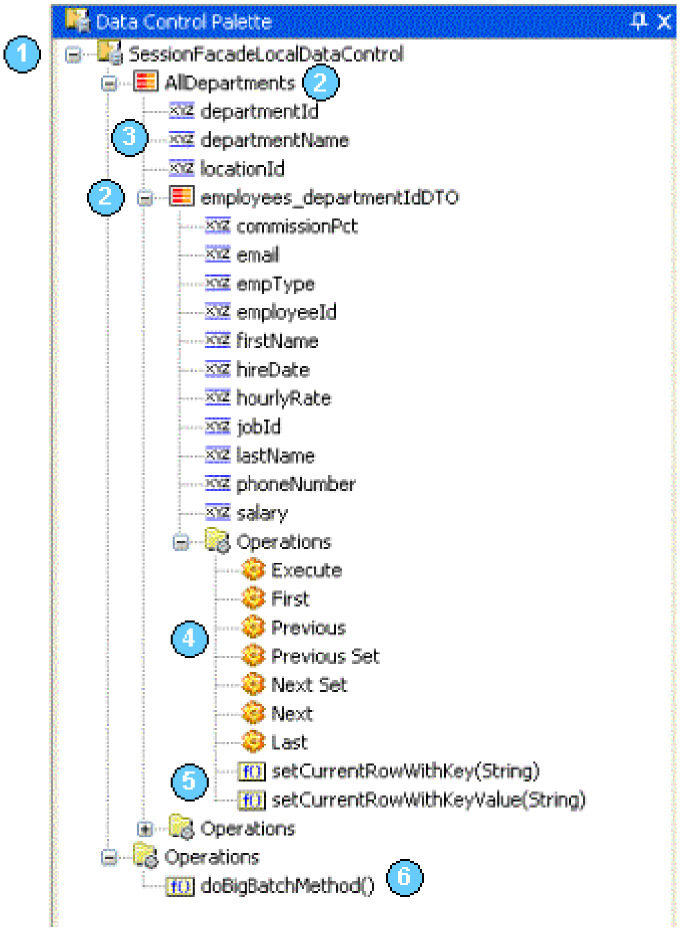
To view the business services you have registered for use with your client application:

- If the palette is not yet displayed, choose **View | Data Control Palette**.

  OR

- If the palette is already displayed, right-click in the palette and choose **Refresh Palette**.

If you use EJB technology as your business service technology, model information will be exposed to the view and controller layers through Oracle ADF data control interfaces implemented by thin, Oracle-provided adapter classes, as shown in the following figure.



1 The data control

This node represents the data control, which in this case is a session facade. Drag a stateless session bean from the navigator and drop it onto the Data Control Palette to create the Oracle ADF data control or to refresh its contents.

2 Row sets

These nodes represent collections:

- **AllDepartments** is a collection of the Departments DTO.
- **employees_departmentIdDTO** is a collection within **AllDepartments**, equivalent to a master-detail relationship.

3 Attributes

These nodes represent individual attributes.

4 Standard operations

These nodes represent standard collection operations. The session facade has similar nodes that represent transaction-level operations. The operations available for a collection include previous set and next set, which scroll through the data based on range size.

5 Special navigation operations

These nodes allow you to set the row set's current row to a particular displayed row.

6 Custom method

Custom service methods will appear on the Data Control Palette.

There are four files that are generated when you create a data control from a session bean, two Java files and two XML files. The JDeveloper design time creates the XML files automatically to facilitate such things as resolving accessor types in the Property Inspector. The four generated files are:

- `SessionEJB.xml` is a generated XML document that represents all JavaBeans surfaced through the ADF data controls, other than built-in classes like `java.lang.String`. The `create()` method exposed through the session EJB data control returns a session EJB instance. This is the XML file to represent that Java type.
- `SessionEJBDataControl.java` is a generated Java class that provides a JavaBeans wrapper to expose the stateless session bean's remote (or local, if no remote interface is exposed) methods, delegating the method requests to a lazily-created session bean instance.
- `SessionEJBDataControl.xml` is an XML document that may be edited through the Property Inspector, typically to assign member type information for collection or list methods exposed through the session bean's component interface.
- `SessionEJBDataControlBeanInfo.java` is the BeanInfo class for `SessionEJBDataControl.java`. The BeanInfo class serves to filter out noncomponent interface methods.

Oracle ADF Data Controls for Web Services

If you use web services as your business service technology, model information will be exposed to the view and controller layers through Oracle ADF data control interfaces implemented by thin, Oracle-provided adapter classes.

JDeveloper creates Oracle ADF data controls for a web service by generating a stub or proxy to the service, and creating the data controls from the stub. Any web service is available to be exposed as a data control in JDeveloper as long as the design time can create a stub for that web service.

You can create data controls for web services that you have created in JDeveloper as part of your application with just one mouse-click. The process of creating data controls for external web services, that is web services somewhere on the Web, is different in that you have to make the WSDL document available in the navigator first. Creating data controls is described for both these cases below.

Finally, you can view the business services you have registered for use with your client application.

Creating Data Controls for Web Services Created in JDeveloper

You can incorporate the functionality of a web service that you have created in JDeveloper in your application. This can be either a SOAP web service or a J2EE web service, and in either case the service should be deployed in the usual manner before creating data controls. When you have created a web service in JDeveloper, the web service container is listed in the navigator.

To create data controls for a web service created in JDeveloper:

- With the web service deployed, right-click the web service container in the navigator and choose **Create Data Control**. Alternatively, drag the web service container node to the Data Control Palette.

  OR

- With the web service container selected in the navigator, right-click the WSDL in the Structure window and choose **Create Data Control**. Alternatively, drag the WSDL node to the Data Control Palette.

Creating Data Controls for External Web Services

When you know the URL of the WSDL document, you need to make the WSDL available in the Application Navigator before you can create data controls for the service. There are two ways to do this:

- By creating a new WSDL document and importing the external WSDL document into it
- By creating a stub to the web service, and allowing the wizard to add the WSDL document to the navigator

Another way of using an external web service in an Oracle ADF application is to locate the service in a UDDI registry, and create the data controls from the Connection Navigator.

These three cases are described below.

To create data controls for an external web service by creating a new WSDL document:

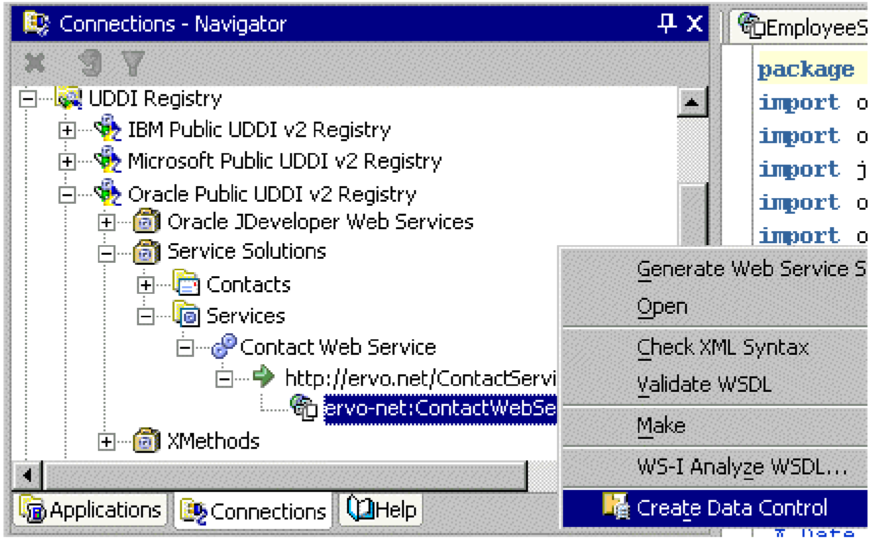
1. Create a new WSDL document, accepting the defaults.
2. In a browser, open the WSDL document of the external web service you want to use as the service provider. View the source of the document, and copy the XML source of the WSDL.
3. Replace the contents of the WSDL document you have just created with the source from the WSDL document of the web service you want to create data controls for.
4. Right-click the WSDL document and choose **Create Data Control**. Alternatively, drag the WSDL node from the navigator to the Data Control Palette.

To create data controls for an external web service using a wizard:

1. Create a stub to the web service, and paste the WSDL URL into the **WSDL Document URL** field on the Select Web Service Description page. On the same page, select **Import WSDL URL Into Project**, and complete the wizard.
2. Right-click the WSDL document and choose **Create Data Control**. Alternatively, drag the WSDL node from the navigator to the Data Control Palette.

To create data controls for a web service located in a UDDI registry:

1. Locate the web service in the UDDI registry. The service is listed in the Connection Navigator under the UDDI registry node.
2. Right-click the web service and choose **Create Data Control**, as shown in the following figure.



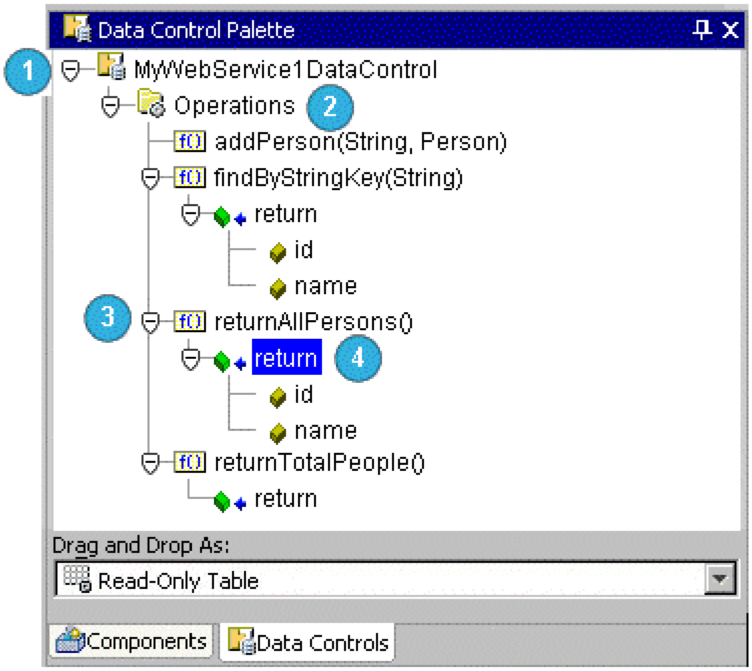
To view the business services you have registered for use with your client application:

- If the palette is not yet displayed, choose **View | Data Control Palette**.

  OR

- If the palette is already displayed, right-click in the palette and choose **Refresh Palette**.

If you use web services as your business service technology, model information will be exposed to the view and controller layers through ADF data control interfaces implemented by thin, Oracle-provided adapter classes, as shown in the following figure.



1 Web Service Data Control

The **MyWebService1DataControl** is the data control node. It is created by right-clicking on the web service container **MyWebService1** in the navigator, and choosing **Create Data Control**.

2 Operations

Under the **Operations** node you can find the available web methods for the web service.

3 Data Control Associated with Web Service Method

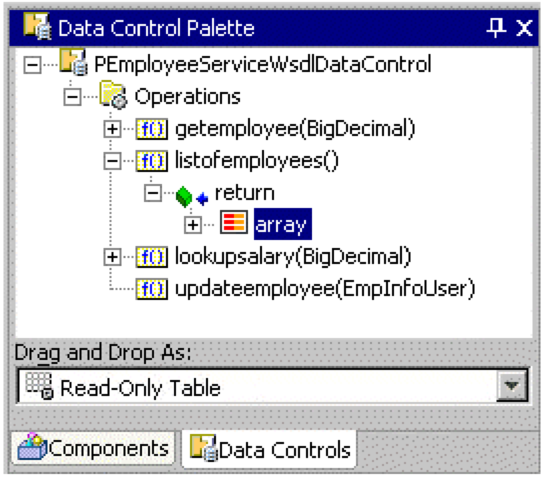
The **returnAllPersons()** node is the data control associated with the web service.

4 return Node

This node is the return from the method on the web service.

Web Services That Return Arrays

When a web service returns an array, it is important that the array node be the one used to provide the return values in the application, as shown in the following figure.



The following files are created:

- `DataControls.dcx` identifies the Oracle ADF model layer adapter classes that facilitate the interaction between the client and the available business services.
- `<`*`WebService`*`>.xml`, which contains metadata used by the data control.
- `<`*`WebService`*`>.java`, which contains connection information to the service.
- `<`*`WebService`*`>BeanInfo.java`, which provides the methods available in the web service.
- `<`*`Bean`*`>.java` is created when you register an external web service from the WSDL node in the Application Navigator, and it acts as a proxy to the service.

Oracle ADF Data Controls for JavaBeans and TopLink-Based Beans Components

If you use JavaBeans technology as your business service technology, model information will be exposed to the view and controller layers through Oracle ADF data control interfaces implemented by thin, Oracle-provided adapter classes.

Note: Your JavaBean may have been created based on TopLink mappings that you specified.

To create adapter classes and data controls:

1. In the navigator, select the **Model** project which contains the model objects you want to expose to the application client.
2. In the model project, right-click the bean and choose **Create Data Control**.

   OR

   Drag the bean from the Application Navigator onto the Data Control Palette to create the data controls.

3. If your bean defines attributes or methods that return a `Collection` type, you must specify the return type for the data control.

JDeveloper adds the data control definition file (`DataControls.dcx`) to the model project. The `.dcx` file identifies the Oracle ADF model layer adapter classes that facilitate the interaction between the client and the available business services.

To view the business services you have registered for use with your client application:

- If the palette is not yet displayed, choose **View | Data Control Palette**.

  OR

- If the palette is already displayed, right-click in the palette and choose **Refresh Palette**.

If you use JavaBeans technology as your business service technology, model information will be exposed to the view and controller layers through ADF data control interfaces implemented by thin, Oracle-provided adapter classes.

If you use Oracle TopLink POJO to create JavaBeans as your business service technology, your application can also use Oracle ADF data controls to access them.

For more information, see the TopLink documentation provided by the JDeveloper help system.

Summary of Oracle ADF Data Control Operations

When you create and register business services for an application, the Data Control Palette displays two types of actions:

- Actions that typically operate on all data collections in the current web page’s binding context (such as commit and rollback) in the **Operations** folder at the root level of the hierarchy.
- Operations on a specific data collection (for example, EmployeesView). Data collection–specific operations (such as create and delete) appear in the **Operations** folder as child nodes of the collection in the Data Control Palette.

Typical data control–level, global actions defined by business services include:

- **Commit** commits a transaction that updates the values of data objects from the bound data collection to the database.
- **Rollback** rolls back a transaction meant to update the values of data objects in the bound data collection. No data is sent to the database.

Typical data collection–specific operations defined by business services include:

- **Create** creates a new data object in the bound data collection.
- **Delete** deletes the current data object from the bound data collection.

- **Execute** executes the bound action defined by the data collection. In the case of a JavaBean, the execute operation will refresh the data control.
- **Find** retrieves a data object from the data collection.
- **First** navigates to the first data object in the data collection range.
- **Last** navigates to the last data object in the data collection range.
- **Next** navigates to the next data object in the data collection range. If the current range position is already on the last data object, then no action is performed.
- **Next Set** moves the viewable range to the first data object after the current range position defined by the bound data collection. For example, when the data collection is a row set, next set navigates to the first row after the current range position. If the current range position is already on the last set, then no action is performed. **Note:** This operation is available only with Oracle ADF Business Components.
- **Previous** navigates to the previous data object in the data collection range. If the current position is already on the first data object, then no action is performed.
- **Previous Set** moves the viewable range to the data objects located just before the current range defined by the bound data collection. For example, when the data collection is a row set, previous set navigates to the last row before the current range position. If the current range position is already on the first set, then no action is performed. **Note:** This operation is available only with Oracle ADF Business Components.
- **setCurrentRowWithKey(String)** passes the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. When passing the key, the URL for the form will not display the row key value. You may use this operation when the data collection defines a multipart attribute key.
- **setCurrentRowWIthKeyValue(String)** passes the row key as a String specified by the input field. The row key is used to set the currency of the data object in the bound data collection. Use this operation only when the data collection uses a single key attribute and does not define a multipart attribute key. When passing the key as a value, the URL for the form will display the row key value.

Summary of Oracle ADF Bindings

Oracle ADF provides these types of bindings:

- Iterator binding
- Value binding
- Action binding

About the Iterator Binding

The `oracle.jbo.uicli.binding.JUIteratorBinding` class implements the iterator binding.

The iterator binding is a runtime object that your application creates to access the Oracle ADF binding context. The iterator binding holds references to the bound data collection, accesses it, and iterates over its data objects. The iterator binding notifies value bindings of row currency and row state. Then, value bindings that you define allow UI components to display or update the current data object of the collection being iterated over. In this way, the iterator binding provides uniform access to various collection types from different business services.

In the case of an Oracle ADF Business Components view object, the Oracle ADF bindings for the UI components may be able to display a row currency indicator. In a table, for example, the current row is identified at runtime by an asterisk (*) symbol displayed in the first column of that row. The currency indicator is available through the iterator binding only when the bound view object has a key attribute defined. When no key attribute is defined for the bound view object, each row of the table will display the asterisk. In this case, you can edit the UI component in the visual editor to prevent the asterisks from displaying at runtime.

The `oracle.jbo.uicli.binding.JUIteratorBinding` class abstracts out the most commonly used methods for collection and currency management:

- **error** returns any exception that was cached while validating the changes made to data through the iterator or through any Oracle ADF data control associated with the iterator.
- **estimatedRowCount** returns the maximum row count of the rows in the collection with which this iterator binding is associated.
- **name** returns the name of this iterator binding.
- **rangeSize** returns the range size of an Oracle ADF Business Components row set iterator. This property is limited to data controls registered with Oracle ADF Business Components.
- **rowAtRangeIndex** returns the data object at the specified index of the collection. In the case of data controls registered with Oracle ADF Business Components, it returns the row of the index in the current range.

About the Value Bindings

Attribute Value Binding

The `oracle.jbo.uicli.binding.JUCtrlValueBinding` class implements the attribute value binding.

The attribute value binding permits the databound UI component to obtain the attribute value of the specified collection’s data object. Depending on the type of UI component, users may view and, in some cases, edit the value of the attribute.

The `oracle.jbo.uicli.binding.JUCtrlAttrsBinding` class defines no properties of its own.

However, you can work with these properties defined by the class hierarchy of the attribute binding:

- **attributeValue** returns the value of the first attribute to which the binding is associated.
- **attributeValues** returns the value of all the attributes to which the binding is associated in an ordered array.
- **attributeDef** returns the attribute definition for the first attribute to which the binding is associated.

- **attributeDefs** returns the attribute definitions for all the attributes to which the binding is associated.
- **displayHint** returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not.
- **inputValue** returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.
- **label** returns a map of labels keyed by attribute name for all attributes to which the binding is associated.
- **labels** returns the label for the first attribute of the binding.
- **labelSet** returns an ordered set of labels for all the attributes to which the binding is associated.
- **mandatory** returns whether the first attribute to which the binding is associated is required.
- **tooltip** returns the tooltip hint for the first attribute to which the binding is associated.
- **updateable** returns whether the first attribute to which the binding is associated is updateable.

And you can work with the generic properties defined by the root class `DCControlBinding`.

Boolean Value Binding

The `oracle.jbo.uicli.binding.JUCtrlBoolBinding` class implements the boolean binding.

The boolean binding obtains the attribute value of the specified collection’s data object based on the control’s selection state.

The `oracle.jbo.uicli.binding.JUCtrlBoolBinding` class has no properties of its own.

However, you can work with these properties defined by the class hierarchy of the boolean binding:

- **attributeValue** returns the value of the first attribute to which the binding is associated.
- **attributeValues** returns the value of all the attributes to which the binding is associated in an ordered array.
- **attributeDef** returns the attribute definition for the first attribute to which the binding is associated.
- **attributeDefs** returns the attribute definitions for all the attributes to which the binding is associated.
- **displayData** returns a list of map elements. Each map entry contains the following elements:
    - `selected` - A boolean `TRUE` if current entry should be selected.
    - `index` - The index value of the current entry.
    - `prompt` - A concatenated string of all display attribute values for the current entry.
    - `displayValues` - The iterator of display attribute values.
    - `selectedIndex` - The index of the selected entry to which the binding is associated.
- **displayHint** returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not.

- **displayHints** returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements:
    - `label` - The label to display for the current attribute.
    - `tooltip` - The tooltip to display for the current attribute.
    - `displayHint` - The display hint for the current attribute.
    - `displayHeight` - The height in lines for the current attribute.
    - `displayWidth` - The width in characters for the current attribute.
    - `controlType` - The control type hint for the current attribute.
    - `format` - The format to be used for the current attribute.
- **inputValue** returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.
- **label** returns a map of labels keyed by attribute name for all attributes to which the binding is associated.
- **labels** returns the label for the first attribute of the binding.
- **labelSet** returns an ordered set of labels for all the attributes to which the binding is associated.
- **mandatory** returns whether the first attribute to which the binding is associated is required.
- **tooltip** returns the tooltip hint for the first attribute to which the binding is associated.
- **updateable** returns whether the first attribute to which the binding is associated is updateable.

And you can work with the generic properties defined by the root class `DCControlBinding`.

List Value Binding

The `oracle.jbo.uicli.binding.JUCtrlListBinding` class implements the list binding.

Depending on the type of UI component, the list binding may:

- Update an attribute of a data object in the bound collection
- Traverse the data objects of the bound collection
- Update a target data object in one collection using the value from another collection's data object

The `oracle.jbo.uicli.binding.JUCtrlListBinding` class provides accessor methods for these properties that you can work with:

- **displayData** returns a list of map elements. Each map entry contains the following elements:
    - `selected` - A boolean `TRUE` if current entry should be selected.
    - `Index` - The index value of the current entry.
    - `Prompt` - A concatenated string of all display attribute values for the current entry.
    - `displayValues` - The iterator of display attribute values.
    - `selectedIndex` - The index of the selected entry to which the binding is associated.
- **displayHints** returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements:

- o `label` - The label to display for the current attribute.
- o `tooltip` - The tooltip to display for the current attribute.
- o `displayHint` - The display hint for the current attribute.
- o `displayHeight` - The height in lines for the current attribute.
- o `displayWidth` - The width in characters for the current attribute.
- o `controlType` - The control type hint for the current attribute.
- o `format` - The format to be used for the current attribute.

Additionally, you can work with these properties defined by the class hierarchy of the list binding:

- **attributeValue** returns the value of the first attribute to which the binding is associated.
- **attributeValues** returns the value of all the attributes to which the binding is associated in an ordered array.
- **attributeDef** returns the attribute definition for the first attribute to which the binding is associated.
- **attributeDefs** returns the attribute definitions for all the attributes to which the binding is associated.
- **displayHint** returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not.
- **inputValue** returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.
- **label** returns a map of labels keyed by attribute name for all attributes to which the binding is associated.
- **labels** returns the label for the first attribute of the binding.
- **labelSet** returns an ordered set of labels for all the attributes to which the binding is associated.
- **mandatory** returns whether the first attribute to which the binding is associated is required.
- **tooltip** returns the tooltip hint for the first attribute to which the binding is associated.
- **updateable** returns whether the first attribute to which the binding is associated is updateable.

And you can work with the generic properties defined by the root class `DCControlBinding`.

Range Value Binding

The `oracle.jbo.uicli.binding.JUCtrlRangeBinding` class implements the range binding.

The range binding permits the databound UI component to obtain a range of attribute values from the specified collection’s data objects and to display the position of the current data object relative to that range.

The `oracle.jbo.uicli.binding.JUCtrlRangeBinding` class provides accessor methods for these properties that you can work with:

- **estimatedRowCount** returns the maximum row count of the rows in the collection with which this iterator binding is associated.
- **rangeSet** returns a list of map elements over the range of rows from the associated iterator binding. The elements in this list are wrapper objects over the indexed row in the range that restricts access to those attributes to which the binding is bound. The properties returned on the reference object are:
    - `index` - The range index of the row this reference is pointing to.
    - `key` - The key of the row this reference is pointing to.
    - `keyStr` - The String format of the key of the row this reference is pointing to.
    - `currencyString` - The current indexed row as a String. Returns "*" if the current entry belongs to the current row; otherwise, returns " ". This property is useful in JSP applications to display the current row.
    - `attributeValues` - The array of applicable attribute values from the row.

    And you may also access an attribute value by name on a range set like `rangeSet.Dname` if `Dname` is a bound attribute in the range binding.

Additionally, you can work with these properties defined by the class hierarchy of the range binding:

- **attributeValue** returns the value of the first attribute to which the binding is associated.
- **attributeValues** returns the value of all the attributes to which the binding is associated in an ordered array.
- **attributeDef** returns the attribute definition for the first attribute to which the binding is associated.
- **attributeDefs** returns the attribute definitions for all the attributes to which the binding is associated.
- **displayHint** returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not.
- **inputValue** returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.
- **label** returns a map of labels keyed by attribute name for all attributes to which the binding is associated.
- **labels** returns the label for the first attribute of the binding.
- **labelSet** returns an ordered set of labels for all the attributes to which the binding is associated.
- **mandatory** returns whether the first attribute to which the binding is associated is required.
- **tooltip** returns the tooltip hint for the first attribute to which the binding is associated.
- **updateable** returns whether the first attribute to which the binding is associated is updateable.

And you can work with the generic properties defined by the root class `DCControlBinding`.

Scroll Value Binding

The `oracle.jbo.uicli.binding.JUCtrlScrollBinding` class implements the scroll binding.

The scroll binding permits the databound UI component to display the current position of the data object in the selected collection. When the user scrolls the UI component, the scroll binding maintains the current position in the collection.

The `oracle.jbo.uicli.binding.JUCtrlScrollBinding` class defines no runtime properties of its own.

However, you can work with the generic properties defined by the root class `DCControlBinding`.

About the Action Binding

The `oracle.jbo.uicli.binding.JUCtrlActionBinding` class implements the action binding.

The action binding is a type of binding object defined by Oracle ADF that performs actions on the bound data collection’s row set iterator. At runtime, when the user initiates the action, using a button control, the action binding accesses the Oracle ADF binding context and initiates the specified action on the data objects of the selected collection. The action binding supports many predefined actions.

The action binding defines its own set of runtime properties.

Chapter 5, Overview of Oracle ADF Integration with Struts

The Oracle Application Development Framework (Oracle ADF) extends the Struts framework to integrate with the Oracle ADF model layer. Oracle ADF provides a pure Struts integration, completely consistent with the Struts specification, that simplifies and speeds up web application development. JDeveloper fully supports the use of Struts, providing both visual and XML editors for the Struts configuration file, as well as other features that simplify building applications that use either plain Struts or Struts integrated with the Oracle ADF.

When you start a project using Struts technology, JDeveloper creates all of the basic Struts components for you, including the required definitions in the `web.xml` file, the Struts configuration file itself, and a resource file for use by Struts internationalization features.

This chapter provides an overview of the integration of Oracle ADF and Struts.

Summary

Highlights of the Struts Framework

The Apache Software Foundation's Struts framework helps web application developers create applications that implement the Model-View-Controller (MVC) design pattern. This design pattern allows web application developers to cleanly separate the display code (for example, HTML and tag libraries) from flow control logic (the Struts controller and action classes) from the data model to be displayed and updated by the application.

The *model* is the repository for the application data and business logic. Part of the model's function is to retrieve data from, and persist data to, the enterprise information system, but it is also responsible both for exposing the data in such a way that the view can access it and for implementing a business logic layer to validate and consume the data entered through the view. At the application level, the model acts as a validation and abstraction layer between the user interface and the business data that is displayed. For more information about working with the model layer using Oracle ADF, see Chapter 4, Overview of the Oracle ADF Model Layer.

The *view* is responsible for rendering the model data. The view code itself does not hard-code application or navigation logic, although it may contain some logic to carry out tasks like conditional data display based on a user's role. When an end user carries out an action within the HTML page that is eventually rendered from the view, an event is submitted to the controller, and it is up to the controller to work out what to do next. Struts applications typically use JSP pages for the view layer, but you can also use other technologies. For information about working with the view layer in Oracle ADF, see Chapter 6, Overview of Oracle ADF Data Binding in View Technologies.

Every user action carried out in the view is submitted through the *controller*, which, based on the contents of the request from the browser and the controller's own programming or metadata, decides what to do next. In Struts, the base controller functionality is implemented in the action servlet. The integration of Oracle ADF and Struts includes extensions to several other controller components that provide major benefits to developers. Here is a brief description of their base functionality:

Action class

> The `org.apache.struts.action.Action` class is an extension of the `ActionServlet` class that performs one or more operations in response to a client request. The action class processes a request using its `execute()` method and returns an action forward object that identifies where control should be forwarded (a web page or another action, for example) to supply the appropriate response. By default, the `execute()` method returns `null`, so you must always subclass the base `Action` class in a plain Struts application.

Action mapping class

> An action mapping provides the information the Struts controller servlet needs to know about what action class to call when the controller receives a request. Struts developers define this information as `<action>` elements in the Struts configuration file, an XML file. The Struts framework parses this file and creates the appropriate objects initialized to the correct default values. The `org.apache.struts.action.ActionMapping` class represents the information specified in the `<action>` elements.

Action forward class

> The org.apache.struts.action.ActionForward class is the destination to which the controller sends control when an action class is executed. Like action mappings, action forwards are defined in the Struts configuration file, typically as <forward> elements within an <action> element.

For detailed information about the Struts architecture and base classes, see http://struts.apache.org/index.html.

Oracle ADF Extensions to Struts

The integration of Oracle ADF and Struts comprises the following key elements:

- DataAction and DataForwardAction Classes
- Oracle ADF Lifecycle
- Named Events in Oracle ADF
- The DataActionMapping Class
- The Oracle ADF Data Form Bean
- Oracle ADF Binding Filter

Oracle ADF Data Action and Data Forward Action Classes

The DataAction and DataForwardAction classes are core components of the integration of Oracle ADF and Struts. They are subclasses of the base Struts Action class that prepare the Oracle ADF model binding context for databound web pages. Your databound web application works with these classes through entries in the Struts configuration file.

Unlike the base Struts Action class, these classes offer a set of functionality that often makes it possible to use them without further subclassing. When you do need to subclass, using these classes simplifies the coding process. They implement a pluggable request-handling lifecycle that is Oracle ADF binding container–aware and fully customizable.

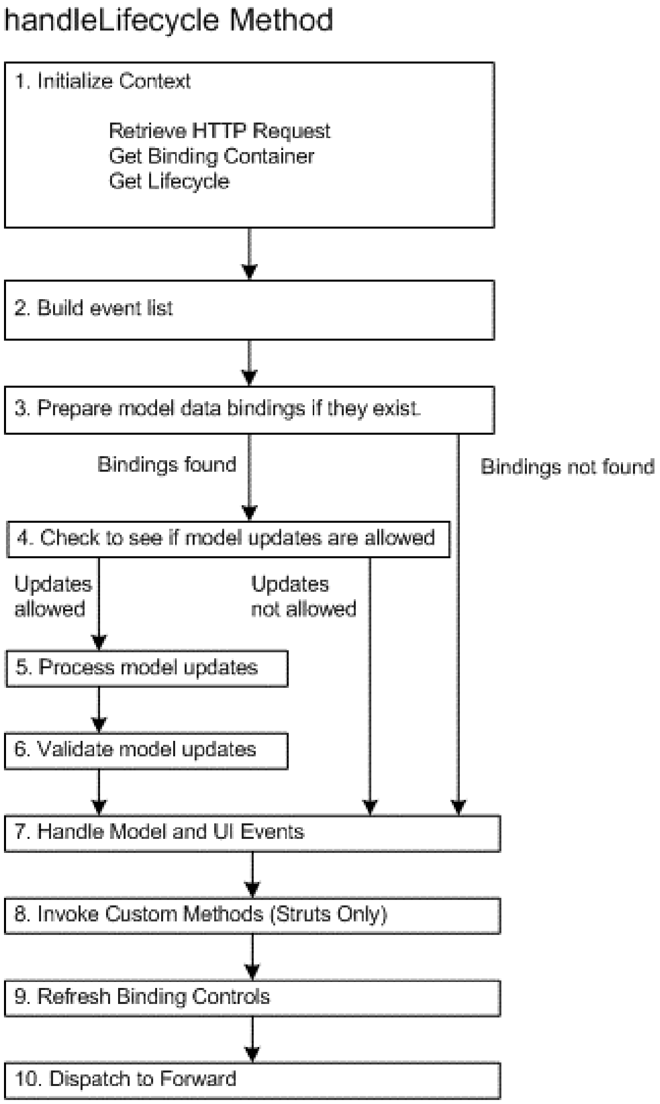
The fully qualified classnames are oracle.adf.controller.struts.actions.DataAction and oracle.adf.controller.struts.actions.DataForwardAction (represented in the Page Flow Diagram as a data page) to prepare the binding context for databound web pages. The DataForwardAction and DataAction classes extend org.apache.struts.action.Action.

Note: DataForwardAction is a subclass of DataAction and is different only in that it is used with the data *page* component, rather than with the data *action* component, in the Page Flow Diagram. For more information about these components, see Data Pages and Data Actions in the Databound Struts Page Flow.

Oracle ADF Lifecycle

In the integration of Oracle ADF with Struts, the `DataAction` and `DataForwardAction` classes implement the Oracle ADF lifecycle interface. This interface provides the code necessary to connect the action of a web page to the ADF model data bindings. At runtime, the lifecycle object calls the correct binding container needed for a web page and tells it to prepare the data to be rendered. The binding container pools the data and stores it locally before rendering the page. By avoiding additional round trips to the database before a web page is rendered, the lifecycle object improves application performance during the rendering process.

The `handleLifecycle()` method does most of the work in the lifecycle by calling a series of operations in a set order. The following diagram shows the steps performed by the method in the `oracle.adf.controller.lifecycle.PageLifecycle` class. Following the diagram is a table that describes what happens in each step.



The following table describes the handleLifecycle() method diagram in detail.

Step	Description
1. Initialize context.	The first step that the method handleLifecycle() performs is to initialize the lifecycle context. The context object holds the value of the associated request, binding container, and lifecycle objects. In a Struts application, the context is an instance of oracle.adf.controller.struts.actions.DataActionContext, a subclass of LifecycleContext. The method handleLifecycle() calls the lifecycle context initialize() method.
2. Build event list.	Next, handleLifecycle() builds the list of events to be performed by retrieving them from the request object with the lifecycle method buildEventList().
3. Prepare model data bindings if they exist.	At this point, the method handleLifecycle() checks for model data bindings by calling getAttribute("bindings") on the binding container retrieved in step 1. If model data bindings do exist, this phase of the lifecycle prepares the data model to receive possible updates from the request. This phase also validates the state token. Note that a lifecycle instance may not have associated model bindings. Instead, the lifecycle can call events that operate only in the user interface. You may want to call custom page navigation events that do not use data bindings. For example, you might want to create an event associated with a help button that would take the user to an HTML page for the web application. If there are no associated model bindings, handleLifecycle() skips to step 7.
4. Check to see if model updates are allowed.	Some events should not be allowed to update the model. If the event is performing a rollback on data changes, for example, you do not allow model updates. The method handleLifecycle() calls shouldAllowModelUpdate() on the lifecycle instance to see whether model updates are allowed. If the user is allowed to update model data from the user interface, the method handleLifecycle() goes to the next step. If model updates are not allowed, handleLifecycle() skips to step 7.
5. Process model updates.	At this point, handleLifecycle() collects the new data values from the request and updates the model with them.
6. Validate model updates.	At this point, handleLifecycle() validates updates to the model by calling validateInputValues() on the binding container associated with the current lifecycle instance.

7. Handle model and UI events.	Next, `handleLifecycle()` calls `processComponentEvents()` on the lifecycle instance. This method handles both named events, which you create and name yourself, and events tied to the data bindings you drop on a page (action bindings). For more information about creating your own named events, see Named Events in Oracle ADF.
8. Invoke custom methods	You can drag and drop methods onto a data action or data page without having to subclass the parent class, as described in Adding Business Service Methods to a Data Action. The method `handleLifecycle()`calls `invokeCustomMethod()` on the lifecycle instance to execute these custom methods at this point.
9. Refresh binding controls.	This step notifies the binding container associated with the current lifecycle instance that all model updates for the action or page are complete. At this point, the method `handleLifecycle()` calls `refreshControl()` on the binding container instance.
10. Dispatch to forward	At this point, the method `handleLifecycle()` calls `findForward()` on the lifecycle instance. For Struts applications, `findForward()` looks for the value of the `<forward>` element in the `struts-config.xml` file, which it passes to the `execute()` method of the data action class or subclass.

You can override lifecycle methods to customize the behavior of your Oracle ADF application. For a Struts-based application, you can override the main lifecycle methods by subclassing the `DataAction` or `DataForwardAction` class. When you want to modify the lifecycle to perform certain behaviors across the application, or if you want to change the behavior of a method that may be called repeatedly by a step in the lifecycle, you must subclass the lifecycle itself. For more information, see When to Use an Oracle ADF Lifecycle Plugin.

Note: You cannot change the order of the lifecycle phases listed in the preceding table.

Named Events in Oracle ADF

An *event* is a specific operation that is executed for a specific command. An event is typically executed by a button or link in a web page. For example, when a user clicks the **Next** button, that click triggers the Next event. *Next* is the event name: naming an event simplifies the process of calling events and creating event handlers. When a command on a web page triggers an event, an event handler performs the work. Oracle ADF provides an easy way to build commands and event handlers for working with events in your web applications. For example, you can:

- Use events to execute action binding model operations. When you create and register business services for an application, the Data Control Palette displays the operations available from the business service. If you create or customize an operation in the business service, you do not have to do any work in the controller layer. You simply have to drag the operation to the web page or data action.

- Write a custom method that overrides an existing method in the action bindings. For example, if you want to override the next operation to behave differently, then you add code to the data action. The data action looks for a custom method first, then looks to see whether the method exists in the action bindings. This way, if a customized method exists, the data action finds it first.
- Develop actions that are independent of the model layer, that is, data actions that do not need to call an action binding to perform their operations. For example, if you want to include a help button in your web pages that takes the user to a help topic for your web application, you can do this with a named event.
- Use events to forward to a new page without subclassing the `DataAction` or `DataForwardAction` class. You can use a named event to define the forward for a data action or data forward action in a web page by making the the value of the forward `name` attribute for the action identical to the name of the associated event in the web page. When Oracle ADF action subclasses encounter an event that is defined neither in the action subclass nor in the action bindings, they assume that the event is the name of a forward.

Note: Both the Oracle ADF data action classes and Oracle ADF UIX use events. In general, when you combine Oracle ADF UIX and Struts in a single application, the data action class takes precedence over the UIX servlet in managing events. If the data action class does not recognize the event as one it knows how to handle, the UIX servlet handles the event.

Oracle ADF Data Action Mapping Class

The Struts action mapping class represents the information configured in the `<action>` element in the Struts configuration file. The `DataActionMapping` class extends the basic Struts data action mapping class to support a number of custom action properties related to ADF data binding. The `DataActionMapping` class determines which lifecycle class should be used based on the type of page (JSP or UIX, for example) to be rendered by the action. It also defines and handles the following additional properties for the `DataAction` and `DataForwardAction` classes:

- `modelReference` is the name of the binding container the data action should use.
- `methodName` is the name of an action binding with a custom method that is to be executed in the data action during the `invokeCustomMethod()` lifecycle phase.
- `numParams` is the number of parameters for a custom method.
- `paramNames` is the EL expressions that retrieve the value for each method parameter.
- `resultLocation` is the EL expression representing the location where the method result is to be stored.

If you want to add additional `<set-property>` elements to the `<action>` element metadata, you need to subclass the `DataActionMapping` class to handle the additional elements.

The `DataActionMapping` class is in the `oracle.adf.controller.struts.actions` package. This class extends the `org.apache.struts.action.ActionMapping` class.

Oracle ADF Data Form Bean

By default, Struts forms in an Oracle ADF web application use a data form bean. The data form bean dynamically makes the attributes for any binding container available to the form and saves you the work of creating the `ActionForm` beans required by your applications.

When you drag and drop a data binding from the Data Control Palette to a JSP page, at runtime Oracle ADF automatically refers to the data form bean for the application. For each value binding in the associated binding container, JDeveloper dynamically creates the `get` and `set` methods for each binding.

Note: JDeveloper does not populate the `<form-property>` element in the `struts-config.xml` file when it uses the data form bean. Your application retrieves the necessary values from the associated binding container.

At runtime, the associated action class, either a `DataAction` or `DataForwardAction` instance or subclass, uses the data form bean to populate the form and submit updates, if any.

Here is a code snippet showing the `<html:form>` tag to illustrate the data form bean behavior:

```
<html:form action="MyDataAction.do">
     <html:text property="dname">
</html:form>
```

At runtime, the `MyDataAction` class needs to resolve the `dname` property. In Oracle ADF, the HTML form is associated with a data action, which is tied to the data form bean. The form goes to the data form bean to resolve the property. The data form bean in turn asks the binding container if it has a binding with that name. The binding container returns the binding if it exists, and the data form bean populates the HTML form with that binding’s value.

Warning: Do not modify, rename, or remove the `DataForm` class or the `DataForm` entries in the `struts-config.xml` file. To work correctly throughout the application, this bean name must not be changed in any way. `DataForm` is a reserved form bean name in Oracle ADF.

The data form bean is an instance of `oracle.adf.controller.struts.forms.BindingContainerActionForm` (a subclass of `org.apache.struts.action.ActionForm` that implements the `apache.commons.beanutils.DynaBean` interface).

Oracle ADF Binding Filter

Oracle ADF web applications use the Oracle ADF binding filter to preprocess any HTTP requests that may require access to the binding context. The binding filter is a servlet filter that does the following:

- Overrides the character encoding at filter initialization time with the name specified as a filter parameter in the `web.xml` file. The parameter name of the filter `<init-param>` is `encoding`.

- Initializes the Oracle ADF model binding context for a user’s HTTP session (for more information about the binding context, see Struts Runtime Integration with the Oracle ADF Model Layer).
- Serializes incoming HTTP requests from the same browser (from framesets, for example) to prevent multithreading problems.
- Notifies data control instances that they are about to receive a request, allowing them to do any necessary pre-request setup.
- Notifies data control instances after the response has been sent to the client, allowing them to do any necessary post-request cleanup.

JDeveloper creates the ADF binding filter and automatically configures it in the application’s `web.xml` file the first time you add a control binding to a web page or drag a business service method to a data action in the Page Flow Diagram.

Here is an example of the elements added when you create a data page in a Struts-based Oracle ADF application for JSP pages and drag a control binding to its associated web page.

Note: This configuration file is included for information only. In most cases you do not need to modify this file.

```
 .
 .
<!--
|Servlet context parameter, which determines which CPX file the filter reads
|at runtime to define the application binding context.
+-->

<context-param>
   <param-name>CpxFileName</param-name>
   <param-value>DataBindings</param-value>
</context-param>
<!-- ADF Binding Filter Class Setup -->
<filter>
   <filter-name>ADFBindingFilter</filter-name>
   <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>

<!-- Default language encoding, which can be set in Tools>Preferences dialog -->
 <init-param>
    <param-name>encoding</param-name>
    <param-value>windows-1252</param-value>
 </init-param>
</filter>

<!--
|A filter mapping links the filter to a static resource or servlet in the
|web application. When a mapped resource is requested, the filter is invoked.
+-->
<filter-mapping>
   <filter-name>ADFBindingFilter</filter-name>
   <url-pattern>*.jsp</url-pattern>
</filter-mapping>
  .
  .
```

```
<filter-mapping>
   <filter-name>ADFBindingFilter</filter-name>
   <servlet-name>action</servlet-name>
</filter-mapping>
<filter-mapping>
   <filter-name>ADFBindingFilter</filter-name>
   <servlet-name>jsp</servlet-name>
</filter-mapping>
```

Note: If you have multiple filters for your application, make sure they are listed in `web.xml` in the order in which you want to run them. At runtime, the filters are called in the sequence listed in that file.

The Oracle ADF binding filter implements the `javax.servlet.Filter` interface and is an example of an intercepting filter.

Where to find additional information:

- For more information about servlet filters, see `http://java.sun.com/products/servlet/docs.html`.
- For more information about intercepting filters, see `http://java.sun.com/blueprints/patterns/InterceptingFilter.html`.

Struts Design Time Integration with Oracle ADF

Oracle ADF provides a rich set of features that help you build Struts-based applications quickly and easily.

Struts Page Flow Diagram

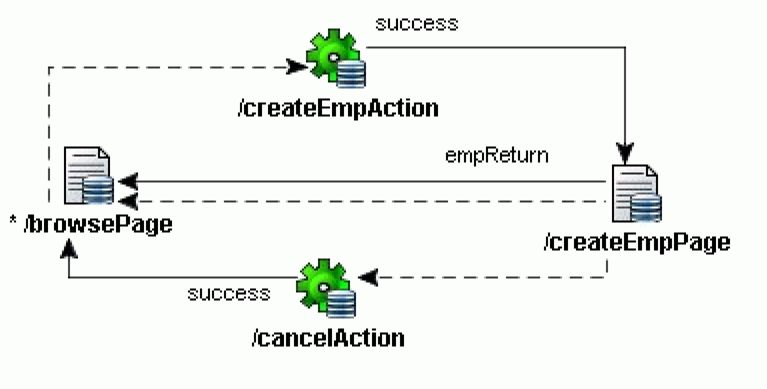
The main working environment you use when building Struts-based applications in JDeveloper is the Struts Page Flow Diagram. This is a visual representation of the Struts configuration file. Changes you make in the Page Flow Diagram are synchronized with the file; changes you make by editing the Struts configuration file manually are also reflected in the page flow. The Page Flow Diagram is your workbench for:

- Creating the application’s page flow
- Selecting web pages and Struts actions to edit
- Running and debugging the application

The standard objects that you require for a Struts page flow are available in the Component Palette, along with the specialized data action and data page used to handle databound pages that use the Oracle ADF model. The data action component represents an Oracle ADF subclass of the Struts action class. The data page represents a data action subclass combined with an action forward and a destination web page. These components are described in more detail in Oracle ADF Data Action and Data Forward Action Classes.

The Component Palette also includes the specialized page forward element to represent a Struts forward action that always performs a simple forward to a specified destination web page. Using the page forward allows you a level of indirection and flexibility in working with web pages: you can change the name of the underlying web page in the Struts configuration file action mapping without having to change all of the components that forward to the page.

By dragging components from the Component Palette onto the Page Flow Diagram, you can create the Struts action mapping elements and action forwards required by the application without needing to edit the file directly. You can also annotate diagrams for documentation purposes. The following diagram shows a simple page flow with two data pages and two data actions with page links (dashed arrows, here representing explicit links between pages and actions) and action forwards (solid arrows).



The Page Flow Diagram also provides:

- Aids for creating and navigating around large page flows
- Tools to organize the layout of your flow
- Customization of diagram fonts and colors

Source View Tab

The **Source** view tab gives you access to the underlying XML in the Struts configuration file. This XML text editor is useful for drilling down to the XML information in the file and making more detailed additions and edits than is possible with the Page Flow Diagram. The XML Editor also allows developers familiar with the structure of the `struts-config.xml` file to update the file rapidly

The XML Editor is fully validated and synchronized with the Page Flow Diagram. You can edit the Struts configuration file in either mode.

Property Inspector Integration with the Struts Configuration File

The Property Inspector is synchronized with the underlying Struts configuration file. You can edit the XML metadata directly in the Property Inspector. You can also use the Property Inspector to display the possible values for Struts tags that take a cue from the contents of the Struts configuration file. For example, when you add an `<html:form>` tag to a web page, you can display a dropdown list of all actions currently defined in the Struts configuration file.

Design Time Rendering of Struts Tag Libraries

You can create web pages with HTML, JSTL, Struts, and other custom tag libraries to implement the view of the data. You can enhance your JSP pages using a large set of custom JSP tag libraries that work with the Struts framework. All of the Struts tag libraries are accessible from the JDeveloper Component Palette when you open a JSP page in the editor.

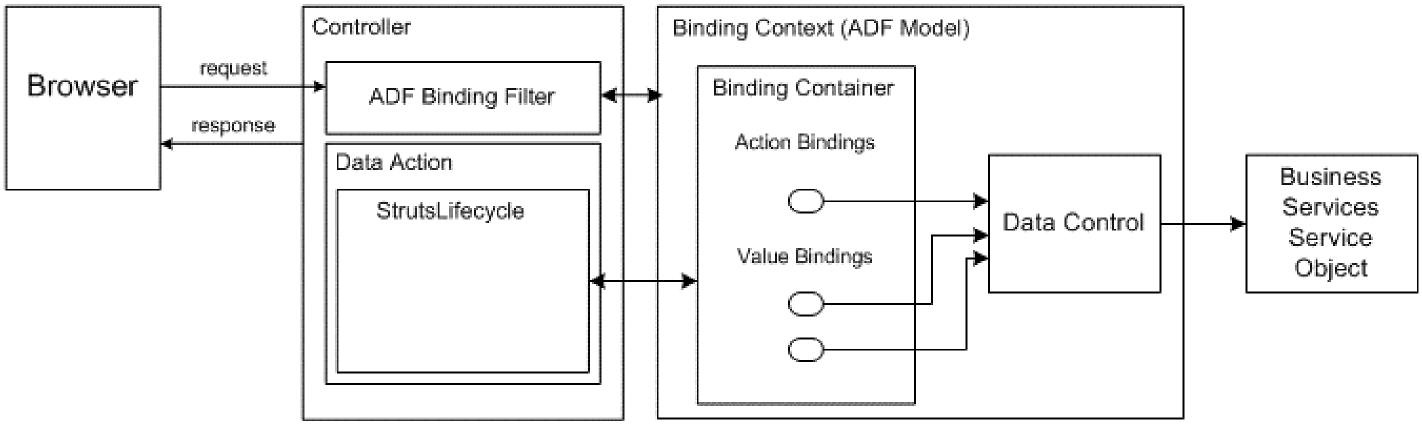
Interactive Code Insight for JSP Code Editing

If you prefer to hand-code your JSP pages instead of using the visual editor and the Property Inspector, the source view of the JSP editor is still Struts-aware and provides Code Insight to assist you. For example, if you are creating a `<bean:message>` tag to display a value from the Struts resource bundle and enter `key="`, JDeveloper displays a list of all valid keys in the resource bundle.

Struts Runtime Integration with the Oracle ADF Model Layer

In a Struts-based application integrated with the Oracle ADF model layer, data control objects implemented for each type of business service expose model data to the controller layer, as shown in the following diagram.

Integration of Controller and Model Layers in Oracle ADF



1. At runtime, the HTTP request goes through a servlet filter, the Oracle ADF Binding Filter, for preprocessing. The binding filter initializes the Oracle ADF model binding context and notifies data control instances that they are about to receive a request, allowing them to do any necessary pre-request setup.

   The binding context contains a series of binding containers and data controls. The binding container is a group of related control and iterator bindings used together for a single page in a web application. A data control abstracts the implementation of a business service, allowing the binding layer to access the data from all services in a consistent way.

2. Next, control passes to the data action class, most of whose functionality is encapsulated in an implementation of the Oracle ADF lifecycle. In a Struts-based application, the lifecycle used is a subclass of `StrutsLifecycle`.
3. The Oracle ADF lifecycle:
    - Calls the correct binding container needed for a web page and tells it to prepare the data to be rendered
    - Processes and validates any model updates submitted by the user
    - Notifies the binding container when all model updates are complete

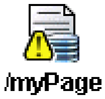
For more information about working with the model layer using Oracle ADF, see Chapter 4, Overview of the Oracle ADF Model Layer.

Data Pages and Data Actions in the Databound Struts Page Flow

If you use ADF data controls in a page or page forward, the web page must either be associated with a data action or it must be part of a data page. If you drag an ADF data control from the Component Palette to a web page before associating it with an action class, a dialog prompts you to choose a data page or data action in the current context. You can also use the dialog to set a default choice for converting any page that needs to be databound.

Working with Data Pages

When you use the Struts Page Flow Diagram to create a data page, JDeveloper updates the `struts-config.xml` file. For example, if you drag the data page icon to an empty Page Flow Diagram and rename it `/myPage`, you get a data page icon with a warning icon overlaid to show that the associated web page has not been created:



At the same time JDeveloper creates the following entry in the `struts-config.xml` file:

```
<action-mappings>
   <action path="/myPage"
      className="oracle.adf.controller.struts.actions.DataActionMapping"
      type="oracle.adf.controller.struts.actions.DataForwardAction"
      name="DataForm"
      parameter=unknown">
   </action>
</action-mappings>
```

The following table describes the attributes and subelements of the action mapping.

Attribute or Element	Description
`action`	The `<action>` element describes a mapping from a request path to the corresponding action class that is used to process the request.
`path`	The name of the data page.
`className`	The fully qualified Java classname of the action mapping subclass to use for this action mapping object. For an Oracle ADF Struts application, the default is `oracle.adf.controller.struts.actions.DataActionMapping`. This class determines which lifecycle class should be used, based on the type of page to which the action forwards. For more information, see Oracle ADF Data Action Mapping Class.
`type`	Fully qualified Java classname of the action subclass that processes requests for this action mapping. When you use a data page, the default class name is `oracle.adf.controller.struts.actions.DataForwardAction.`
`name`	Unique name of the form bean, if any, that is associated with this action mapping. For a data page, the default form bean is `DataForm`. By default, all data actions and data pages share a single form bean. For more information, see Oracle ADF Data Form Bean.
`parameter`	General-purpose configuration parameter used to pass extra information to the action object selected by this action mapping. When you specify the associated web page, the value of this attribute changes from `unknown` to the page name.

When you create the associated web page (`myPage.jsp`, in this example) in a Struts application, the warning overlay disappears and the data page icon appears normal:

At the same time, JDeveloper updates the action mapping in the `struts-config.xml` file as shown in bold:

```
<action-mappings>
    <action path="/myPage"
        className="oracle.adf.controller.struts.actions.DataActionMapping"
        type="oracle.adf.controller.struts.actions.DataForwardAction"
        name="DataForm"
        parameter="/myPage.jsp">
       <set-property property="modelReference" value="myPageUIModel"/>
    </action>
</action-mappings>
```

JDeveloper updates the value of the `parameter` attribute in the action mapping to the name of the associated web page file. In addition, JDeveloper adds a `<set-property>` definition in the action with the property set to `modelReference` and the value set to the name of the binding definition (*`pageName`*`UIModel`). When you add the first data control to the associated web page, JDeveloper also creates these project files:

- A client binding definition file (`myPageUIModel.xml`), which is specific to the web page. JDeveloper creates a client binding definition file for each web page in the project.
- A client project definition file (`DataBindings.cpx`), which creates the Oracle data controls registered with your application's business services. JDeveloper creates only one client project definition file per project.

Working with Data Actions

When you use the Struts Page Flow Diagram to create a data action, JDeveloper updates the `struts-config.xml` file. For example, if you drag the data action icon to an empty Page Flow Diagram and rename it as shown here:

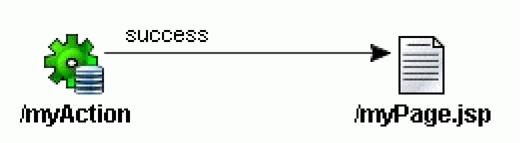
JDeveloper creates the following entries in the `struts-config.xml` file:

```
<form-beans>
    <form-bean name="DataForm"
         type="oracle.adf.controller.struts.forms.BindingContainerActionForm"/>
</form-beans>
<action-mappings>
    <action path="/myAction"
         className="oracle.adf.controller.struts.actions.DataActionMapping"
         type="oracle.adf.controller.struts.actions.DataAction"
         name="DataForm"/>
</action-mappings>
```

There are two main differences between this entry and the entry for the previously described data page:

- The action type is `oracle.adf.controller.struts.actions.DataAction`.
- The action mapping for a data action does not use the `parameter` attribute.

The data action does not have an associated web page, so you need to create a forward that will redirect to the next page to be displayed. You can define the `<forward>` element by dragging the **Forward** icon (→) to the Page Flow Diagram. You can forward to a data page, a page forward, a web page, or another data action. In this example, the forward is to a web page, `myPage.jsp`:



When you add the forward to the web page, JDeveloper updates the action mapping for the data action as shown in bold:

```
<action path="/myAction"
      className="oracle.adf.controller.struts.actions.DataActionMapping"
      type="oracle.adf.controller.struts.actions.DataAction" name="DataForm">
      <set-property property="modelReference" value="myPageUIModel"/>
      <forward name="success" path="/myPage.jsp"/>
</action>
```

The following table describes the attributes and subelements of the complete action mapping.

Attribute or Element	Description
`action`	The `<action>` element describes a mapping from a request path to the corresponding action class that is used to process the request.
`Path`	The name of the data action.
`className`	The fully qualified Java clasname of the action mapping subclass to use for this action mapping object. For an Oracle ADF Struts application, the default is `oracle.adf.controller.struts.actions.DataActionMapping`. This class determines which lifecycle class should be used, based on the type of page to which the action forwards. For more information, see Oracle ADF Data Action Mapping Class.
`Type`	Fully qualified Java classname of the action subclass that processes requests for this action mapping. When you use a data action, the default classname is `oracle.adf.controller.struts.actions.DataAction.`
`Name`	Unique name of the form bean, if any, that is associated with this action mapping. For a data action, the default form bean is `DataForm`. By default, all data actions and data pages share a single form bean. For more information, see Oracle ADF Data Form Bean.
`set-property`	Specifies the method name and initial value of an additional JavaBean configuration property. When the object representing the surrounding element is instantiated, the accessor for the indicated property is called and passed the indicated value. In this case the property is set to `modelReference` and the value set to the name of the binding definition (*pageName*`UIModel`).
`forward`	The page or other resource to which the action forwards. In this case, the resource is a JSP page, `myPage.jsp`.

For information about when to use a data page or data action, see When to Use a Data Page or a Data Action.

Best Practices

The integration of Oracle ADF with Struts provides you with several options for simplifying application development, from working with individual events to modifying behavior across a web application.

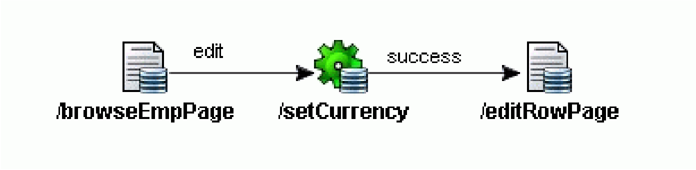
When to Use a Data Page or Data Action

The data page is the recommended component for adding an action that forwards to a databound web page. Use a data page when:

- You have a data action that is forwarding directly to a web page. In the Page Flow Diagram, a data page represents the combination of a data forward action class instance, a forward transition, and a web page.
- You want to simplify a complex page flow diagram. Using the data page reduces the number of elements in your page flow and makes complex application diagrams easier to read.

Use a data action when:

- You need to perform multiple operations before rendering a web page (this process is called *chaining* data actions). Use the data action to perform any operations that do not forward directly to a web page. For example, the data action in the following diagram sets the current row before forwarding to a data page that prepares the data for the edit form on the next web page. In this example, setting the current row requires a separate data action:



Depending on the design of your application, you may also want to use a data action that forwards to a separate page forward or page instead of using a data page. You may consider this approach to be appropriate, for example, when you are forwarding to a single page from multiple data actions. The page forward performs a simple forward to a destination web page.

Adding Business Service Methods to a Data Action

You can use the Data Control Palette to drag methods onto data actions in your databound Struts page flow. This option provides another way to add functionality to your web application without subclassing the data action class. The design time updates the action mapping for the data action like this:

```
<set-property property="methodName" value="MyPageUIModel.ActionName"/>
<set-property property="resultLocation" value="${requestScope.methodResult}"/>
<set-property property="numParams" value="1"/>
<set-property property="paramNames[0]" value="${param.paramName0}"/>
```

At runtime, the lifecycle implemented in the data action executes the method defined by the business service through the Oracle ADF data controls. For more information, see step 8 in the lifecycle-handling table earlier in this chapter.

Note: There is a paramNames[] property for each method parameter. If the method has no parameters, the value of numParams is set to 0 and there are no paramNames properties set. If you are using a custom method that requires parameters, you must add the paramNames properties using the Struts Configuration Editor.

When to Subclass the DataAction or DataForwardAction Class

The integration of Oracle ADF with Struts is designed to minimize the need to subclass the DataAction and DataForwardAction classes. The execute() method is final in both classes. However, you may need to subclass when:

- You want to develop named events that are independent of the model layer, as described in Named Events in Oracle ADF
- You want to override the behavior of an existing action binding
- You want to change the behavior of one of the steps in the lifecycle (as described in Oracle ADF Lifecycle) for a single action

The data action follows a general pattern of exposing the lifecycle methods and delegates most of its functionality to the lifecycle class.

When to Use an Oracle ADF Lifecycle Plugin

For a Struts-based application, you can override the main lifecycle methods by subclassing the DataAction or DataForwardAction class. You must subclass the lifecycle itself and register a lifecycle plugin when:

- You want to modify the lifecycle to perform certain behaviors across the application
- You want to change the behavior of a method in the lifecycle subclass that cannot be overwritten in the DataAction or DataForwardAction class

The data action follows a general pattern of exposing the lifecycle methods and delegating most of its functionality to the lifecycle class (also known as the Decorator Design Pattern). For example, handleError() follows this pattern, as do the lifecycle methods described in Oracle ADF Lifecycle. However, some methods, such as handleEvent(), do not follow this pattern. When a method is not a step of the lifecycle that is executed only once, you must subclass the lifecycle to change it. There is a difference between the phases of the lifecycle and the individual operations that each phase executes. handleEvent() is in the second category because the lifecycle calls it for each event, which means it may be called more than once. You may want to customize the basic handleEvent() logic for all events in your application, to do something before, after, or instead of the default. You can override the handleEvent() method on the lifecycle class but not on the data action class. You must create and register your own lifecycle implementation.

Modifying code once for all data actions in your application instead of overriding each data action may also be an effective development strategy. JDeveloper includes a Struts plugin implementation that allows you to specify a new LifecycleFactory class so that you can create and use a lifecycle subclass. A LifecycleFactory class implements StrutsPageLifecycleFactory, which is an abstract class.

To override the Struts lifecycle you need to specify the new `Lifecycle` class during the data action configuration, because some of the lifecycle methods are used during configuration. To specify the new class, you add a `<plug-in>` element to the `struts-config.xml` file. The plugin is a Struts mechanism that allows you to load components dynamically at application startup.

The entry in the configuration file has the following syntax:

```
<plug-in
    className="oracle.adf.controller.struts.actions.PageLifecycleFactoryPlugin">
    <set-property property="lifecycleFactory"
    value="mypackage1.myStrutsLifecycleFactory"/>
</plug-in>
```

The property is used to pass the name of the new `LifecycleFactory` class to the application. A `LifecycleFactory` class implements the abstract class `StrutsPageLifecycleFactory` class. You must define the following method in the class:

```
public StrutsPageLifecycle getPageLifecycle(String path)
```

See the javadoc for `DefaultStrutsPageLifecycleFactory` for an example.

Summary of Best Practices in Working with Oracle ADF/Struts Integration

If you want to....	Then...
Create a custom method that accesses the ADF model layer.	Create a custom method in the business services layer.
Create an action that prepares model data for display, forwards to a specific databound page, and manages any submissions from that page.	Use a data page.
Create an action that executes a custom method exposed by the business service.	Use a data action.
Override an existing method in the action bindings.	Subclass the data action.
Create a custom event that does not access the ADF model layer	Subclass the data action to create a named event.
Use events to forward to a new page without subclassing the data action.	Use an event to define an action forward.
Create an operation that is called every time a data action or data page is called.	Subclass the data action or data forward action.
Modify lifecycle behavior across an entire application (for example, to modify how individual events are handled globally).	Create a lifecycle plugin and register it.
Add extra properties to an action mapping.	Subclass the data action mapping class.

Chapter 6, Overview of Oracle ADF Data Binding in View Technologies

Data binding in view technologies is the ability to create UI components that are bound to data in back-end business services. The Oracle Application Development Framework (Oracle ADF) enables data binding through the objects of the Oracle ADF model layer. These Oracle ADF binding objects are accessible to the web application and Java client application at runtime, where they are instantiated by the data of the business service and the metadata that defines how they will be rendered as UI components.

When the developer creates the view layer, the JDeveloper design time tools help to simplify the task of creating a databound client. Without needing to write code, client developers can assemble databound web pages and Java clients. The client-design task is aided by the JDeveloper design time and its cooperation with the Oracle ADF model layer.

This chapter provides an overview of the integration of Oracle ADF and various view technologies.

Summary

Role of the View Layer

The view layer is that part of the J2EE application that end users of your application interact with:

- For Web applications, the UI is ultimately displayed as HTML rendered by a browser, where the displayed data is pulled from business services in the model layer.
- For Java client applications, which run standalone, the UI consists of Swing components, into which data from business services in the model layer is pushed.

Overall, the view layer for these applications is responsible for:

- Referencing the Oracle ADF bindings in the model layer through the UI components
- Rendering the data in the appropriate format for each UI component to display
- Communicating with the controller layer to handle user interactions, including editing data and navigating the page flow of the application
- Optionally, handling validation of data entered into the UI components (known as *client-side validation*)

When you create J2EE applications in JDeveloper, the design time tools help to simplify the task of creating a databound client. Without needing to write code, client developers can assemble databound web pages and Java clients. The client-design task is aided by the JDeveloper design time and its cooperation with the Oracle ADF model layer. Without needing to understand the inner workings of the Oracle ADF model layer, client developers can insert UI components that access actions and data in selected business services. The process for creating databound clients is the same in JDeveloper for any of these supported client technologies:

- Generic JSP pages that use HTML elements and, optionally, JSP tags from JSP custom tag libraries
- Web pages based on Oracle ADF UIX (the Oracle XML web presentation framework available in JDeveloper)
- Java clients created with Swing components and extended by ADF JClient (the Oracle Java client framework available in JDeveloper)

Differences Between JSP Pages and UIX XML Documents

UIX XML is an XML language for defining the user interface of a web application using a rich set of Oracle ADF UIX components. A UIX XML document has the file extension `.uix`, and the file contains a declarative description of the UIX components that define the layout, navigation, and content of a single web application page. At runtime, the UIX servlet interprets the UIX XML documents and renders the appropriate output for the browser or device that requested the page.

In JDeveloper you can use the UIX Visual Editor to visually create your UIX web pages by adding and arranging UIX user interface components, and then test and run your application. JDeveloper also provides wizards to help you build individual UIX pages, and it provides UIX XML template (UIT) files for quicker development. Additionally, because UIX is part of Oracle ADF, it supports data binding of diverse business sources.

JavaServer Pages (JSP) technology is based on Java servlets and, like Java servlets, JSP is a server-side technology. A key difference between JSP pages and servlets is that JSP pages keep static page presentation and dynamic content generation separate. JSP web page designers use:

- HTML tags to design and format the dynamically generated web page
- JSP standard tags or Java-based scriptlets to call other components that generate the dynamic content on the page
- JSP tags from custom tag libraries that generate the dynamic content on the page

UIX JSP provides a tag library that invokes UIX components via a set of tags from a JSP 1.2–compliant tag library. The JSP tags generate the HTML to render tabs, buttons, tables, headers, and other layout and navigational components.

A JSP page has the extension `.jsp`. This extension notifies the web server that the page should be processed by a JSP container. The JSP container interprets the JSP tags and scriptlets, generates the content required, and sends the results back to the client as an HTML or XML page.

JDeveloper provides data binding, tag insight, and other editing features for both technologies.

Some of the key differences are:

- UIX XML exposes a larger set of functionality, as the UIX JSPs are a JSP interface to a subset of the UIX UI components. The JSP tags implement only a subset of the UIX elements and only a subset of the attributes.
- UIX XML provides more powerful templating mechanisms. You can create your own templates in JDeveloper and then build your application pages based on these templates.

JDeveloper Design Time Integration with the Oracle ADF Model Layer

Client developers use the Data Control Palette to create databound HTML elements (for JSP pages), databound Oracle ADF UIX elements (for UIX XML pages), and databound Swing UI components (for JClient panels). The Data Control Palette comprises two selection lists:

- Hierarchical display of available business objects, methods, and data control operations
- Dropdown list of appropriate visual elements that you can select for a given business object and drop into your open client document

Additionally, web application developers use the Data Control Palette to select methods provided by the business services that can be dropped onto the data pages and data actions of the Struts page flow.

Overview of Data Control Palette Usage

In the case of client documents, the hierarchical structure of the business services displayed is determined by:

- Which business services you have registered with the data controls in your model project. The palette displays a separate root node for each business service that you register:
    - Oracle Business Components application modules
    - EJB session beans
    - TopLink mappings–based beans
    - Standard JavaBeans classes
    - Web services
- A bean design time description that is generated in an `.xml` definition file when you create the data control for the bean. The bean's XML definition classifies the bean's property accessors and methods into various categories described below.

  Note: In the case of Oracle ADF BC, the Oracle ADF BC application modules in your model project are automatically published as a data control. Their XML metadata file already contains the information needed by the Oracle ADF BC data control and no additional XML definition files are required when you create the model project. In the case of web services, no bean is involved and the XML definition describes only methods exposed by the web service.

At design time, the Data Control Palette provides the first step to lay out the client user interface and prepare the Oracle ADF bindings. The task of selecting a business object, choosing a visual element for the service, and dropping it into the page generates these items:

- A visual element, which is defined by source code to access the bindings in the client document (HTML and tags for JSP pages, UIX components for UIX XML pages, or Java method calls for JClient-generated panels and forms).
- A binding container when one does not yet exist for the page. The binding container is an XML file that appears in the directory corresponding to the package currently set to the project's default package. (To modify the default package, select the project node in the Application Navigator and display the Property Inspector, where you can set the `defaultPackage` property.)
- An appropriate binding definition to support the visual element. The binding definition is added to the binding container.

Note that, as an alternative, you can also create bindings in the JDeveloper Structure window, which you later reference in your source code, without dragging and dropping from the Data Control Palette.

The code that the Data Control Palette generates in your client document and the bindings that it creates depend on:

- The type of document displayed in the visual editor (must be a JSP page, UIX page, or JClient-generated panel or form)
- The combination of business service and visual element you select and drop into the open document

After you have completed laying out the client document with the Data Control Palette, you can view and customize the individual binding definitions.

Overview of the Data Control Business Objects

The root node of the Data Control Palette represents the data control registered for the business service. While the data control itself is not an item you can select, you may select among the operations it supports. All data control–specific operations appear in the **Operations** folder of the root node. You can work with this type of operation when you want to perform an operation that applies across the Oracle ADF binding context, such as the commit and rollback operations provided by the data control for Oracle ADF BC.

Proceeding down the hierarchy from the root data control node, the palette represents bean-based business services as either:

- Attributes, such as bean properties, which can define simple scalar value objects, structured objects (beans), or collections

  OR

- Operations, such as bean methods, which may or may not return a value or take method parameters

An exception to this hierarchy is the web services, for which the Data Control Palette displays only operations.

In the Data Control Palette, attributes and operations are represented by the following specific icons. Note that icons which appear more than once represent various accessor return types and may be supported by a different set of visual elements, as shown in the following table.

Icon	Description	Visual Element Choices
XYZ	An attribute that represents a scalar value (such as simple Integer or String objects).	The full list of attribute-bound visual elements, such as text input, checkbox, choice list, and radio buttons. These choices vary depending on the document type you create.
▤	An accessor attribute that represents a collection of scalar values (such as ones that provide an array of Integers or a list of Strings).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.
▤	An accessor attribute that represents a collection of structured objects that contain attributes and operations (such as a collection of employee objects).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.

▤	An accessor attribute that represents a collection of structured objects that contain only operations and no attributes (such as a collection of service beans).	Read-only dynamic table and navigation buttons for JSP, more choices for UIX, and the full list of collection-bound controls for JClient.
▬	An accessor attribute that returns a structured object that contains attributes and operations (such as an address object).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.
▬	An accessor attribute that returns a structured object that contains only operations and no attributes (such as a service bean).	Read-only dynamic table and navigation buttons for JSP, more choices for UIX, and the full list of collection-bound controls for JClient.
f()	An operation which may or may not take parameter values (such as a Java method on a bean).	Button or a button with form (JSP), submit button (UIX), button (JClient).
⚙	Built-in operations (such as navigate to row and execute).	Button or a button with form (JSP), submit button (UIX), button (JClient).
f()	Built-in operations to pass the primary key value of a row set (such as set current row with key)	Button or a button with form (JSP), submit button (UIX), button (JClient).

In the Data Control Palette, operations that specify return values are represented by the following specific icons. The specific object returned determines which visual elements are available as shown in the following table.

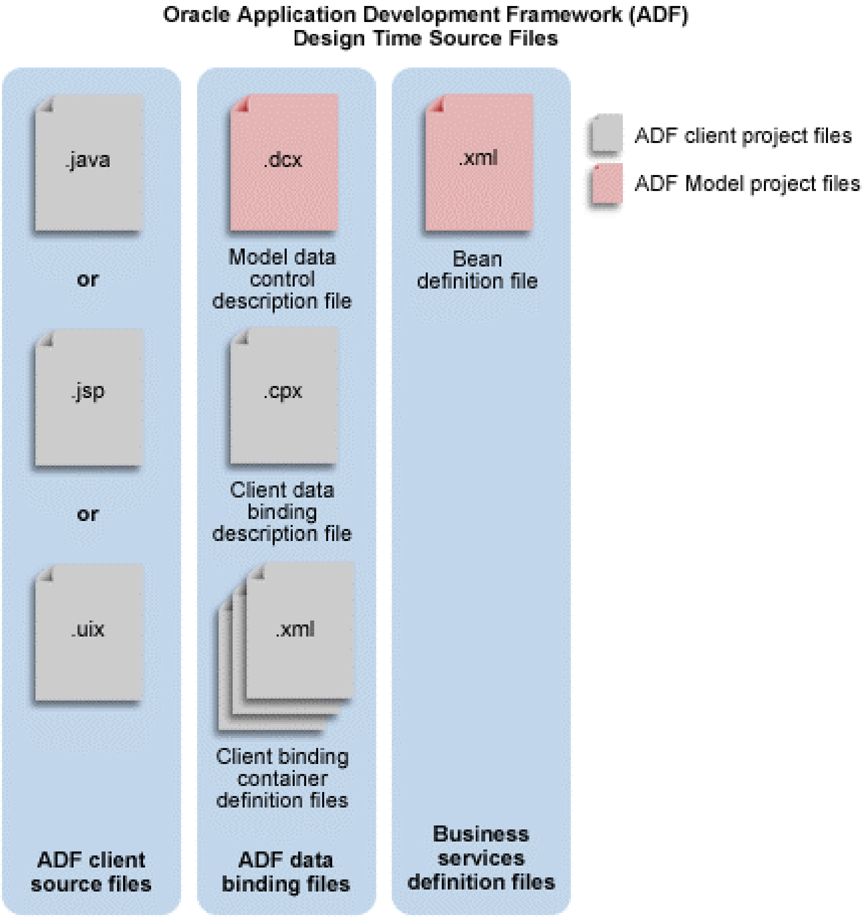
Icon	Description	Visual Element Choices
xyz	An operation that returns a scalar value (such as simple Integer or String objects).	Show result read-only visual element for JSP and UIX. The full list of attribute-bound controls for JClient.
◆◆	An operation that returns a collection of scalar values (such as ones that provide an array of Integers or a list of Strings).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.
◆◆	An operation that returns a collection of structured objects that contain attributes and operations (such as a collection of employee objects).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.
◆◆	An operation that returns a collection of structured objects that contain only operations and no attributes (such as a collection of service beans).	Read-only dynamic table and navigation buttons for JSP, more choices for UIX, and the full list of collection-bound controls for JClient.

◆◆	An operation that returns a structured object that contains only operations and no attributes (such as a service bean).	Read-only dynamic table and navigation buttons for JSP, more choices for UIX, and the full list of collection-bound controls for JClient.
◆◆	An operation that returns a structured object that contains attributes and operations (such as an address object).	The full list of collection-bound visual elements, such as table, graph, navigation buttons, and forms. These choices vary depending on the document type you create.

Note that, depending on the business service definition, method returns which appear in the Data Control Palette may be redundant with the data control attributes defined for the business service. When a choice is available, it is preferable to select the attribute and accessor nodes rather than method returns. Only data control attributes and accessors provide full support for the Oracle ADF bindings. Iterator bindings, for example, are not available for method returns.

Overview of Oracle ADF Project Files

When you build an Oracle ADF–based application using the JDeveloper design time tools, JDeveloper generates project files specific to Oracle ADF, as shown in the following figure.


Oracle Application Development Framework (ADF)
Design Time Source Files
.java
or
.jsp
or
.uix
ADF client source files
.dcx
Model data control description file
.cpx
Client data binding description file
.xml
Client binding container definition files
ADF data binding files
.xml
Bean definition file
Business services definition files
ADF client project files
ADF Model project files

Files in the Oracle ADF Model Project

The DataControls.dcx file is created when you register data controls on the business services. Note that this file is not generated for the Oracle ADF Business Components and Oracle ADF TopLink Mappings data controls. In those cases, the data control obtains the metadata directly from the generated services.

The DataControls.dcx file specifies the factory classes for a bean registered as an Oracle ADF data control. In the case of EJB, web services, and bean-based data controls, you can edit this file in the Property Inspector to add or remove parameters and to alter data control settings.

Various .xml files are created when you register a bean (for example, an EJB session bean) as an Oracle ADF data control. The definition file specifies the bean's available attribute, accessors, and collections available for use by the client application. You will modify this file only when an accessor method returns a collection. In this case, it is necessary to specify the return type. Note that in the case of Oracle ADF Business Components, all accessor return types are known and you do not need to manually perform this step.

About the DataControls.dcx File Syntax

In the case of bean-based and web service–based business services, the DataControls.dcx file appears in the /src/package directory of the model project folder. The Application Navigator displays the file in the model package of the **Application Sources** folder. When you double-click the file node, the data control description appears in the XML code editor. To edit the data control parameters, use the Property Inspector and select the desired parameter in the Structure window.

The following code describes the syntax for a combination of Oracle ADF Business Services, JavaBeans, and web service data controls:

```
<DataControlConfigs
   id="DataControls"
   xmlns="http://xmlns.oracle.com/adfm" >
   <Contents >
      <DataControl
            id="ClassNameDataControl | AppModuleDataControl |

                        PXWebServiceNameDataControl"

            <!-- Indicates for the ADF design time the class to use to represent
                  the data control on the Data Control Palette. -->
                  SubType="DCBC4J | DCJavaBean | DCWebService"

            <!-- Indicates whether the data control for the business service
                  supports query-by-example. This enables the find operation choice
                  in the Data Control Palette for this data control and the

                  associated services. Not used by ADF. -->
                  SupportsFindMode="true | false"

            <!-- Indicates whether the data control for the business service
                  supports transaction semantics. This enables commit and rollback
```

```
                 operations on the data control. -->
                 SupportsTransactions="true | false"
            <!-- Oracle ADF Business Components definitions, including the
                 package, the bc4j.xcfg configuration, and the factory class. -->
                 Package="model"
                 Configuration="AppModuleLocal"
                 FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"

            <!-- Standard Bean definitions, including the factory class,
                 the XML definition file, and the bean class file. -->
                 FactoryClass="oracle.adf.model.generic.DataControlFactoryImpl"
                 Definition="model.Class1"
                 BeanClass="model.Class1" >
      </DataControl>
   </Contents>
</DataControlConfigs>
```

Files in the ViewController Project

Various .jsp, .uix, or .java files are the presentation documents of the client application. You lay out the UI in an Oracle ADF application using a visual editor and the Data Control Palette. When you insert a databound UI component into your document, the page will contain binding expressions that access the Oracle ADF binding objects at runtime. You can edit the binding expressions directly in the source code in order to specify runtime behavior using the available properties of the UI component's binding object.

The DataBindings.cpx file is created the first time you open a web page from the Struts Page Flow Diagram in the visual editor. The .cpx file defines the Oracle ADF binding context for the entire application. The .cpx file provides the metadata from which the Oracle ADF binding objects are created at runtime. The binding context provides access to the bindings across the entire application. You can edit this file in the Property Inspector to add or remove parameters and to alter the binding container settings.

The *pageName*UIModel.xml files are created each time you design a new web page or JClient panel using the Data Control Palette and a visual editor. These XML files define the Oracle ADF binding container for each presentation document in the client application. The binding container provides access to the bindings within the page. Therefore, you will have one XML file for each databound web page or JClient panel. You may need to edit the binding definitions in this file when you remove binding expressions from your presentation documents.

Note: You cannot rename the *pageName*UIModel.xml file in JDeveloper, but you can rename the file outside of JDeveloper in your MyWork/ViewController/src/view folder. If you do rename the *pageName*UIModel.xml file, you must also update the DataBindings.cpx file references in the <Containee> id and FullName attributes.

About the UIModel.xml File Syntax

The `UIModel.xml` file appears in the `/src/view` directory of the view-controller project folder. The Application Navigator displays the file in the view package of the **Application Sources** folder. When you double-click the file node, the binding container description appears in the XML Code Editor. To edit the binding container parameters, use the Property Inspector and select the desired parameter in the Structure window.

The following syntax was generated for a web page that accesses business service objects `MyAttribute1`, `MyAttribute2`, and `MyDataCollectionIterator`, through their corresponding binding objects:

```
<DCContainer
   id="PageNameUIModel"
   xmlns="http://xmlns.oracle.com/adfm"
   Package="view"

   <!-- Indicates whether find mode should be enabled for the page. -->
        FindMode="false | true"

   <!-- Indicates whether to check the currency of the bound collection.
        This ensures that row updates will be applied to the correct row. -->
        EnableTokenValidation="true | false" >

   <Contents >
      <DCIterator
         id="MyDataCollection1Iterator"
         Binds="BusinessServiceDataControl.MyDataCollection"
         <!-- Indicates the number of rows to display from bound collection. -->
         RangeSize="10"
      </DCIterator>
      <DCControl
         id="MyBusinessServiceName"
         Subtype="DCBindingType"
         IterBinding="MyDataCollection1Iterator" >
         <AttrNames>
            <Item Value="MyAttribute1" />
            <Item Value="MyAttribute2" />
         </AttrNames>
      </DCControl>
 </DCContainer>
```

About the DataBindings.cpx File Syntax

The `DataBindings.cpx` file appears in the `/src` directory of the view-controller project folder. The Application Navigator displays the file in the **Application Sources** folder. When you double-click the file node, the binding context description appears in the XML Code Editor. To edit the binding context parameters, use the Property Inspector and select the desired parameter in the Structure window.

The following describes the syntax for a combination of Oracle ADF Business Services, JavaBeans, and web service data controls:

```
<JboProject
   id="DataBindings"
   xmlns="http://xmlns.oracle.com/adfm"

   <!-- Indicates that the components of this project may appear
         in separate XML files. Not used by ADF. -->
         SeparateXMLFiles="false"

   <!-- Used by JClient applications to locate a bc4j.xcfg file. For backwards

        compatibility. -->
        Package=""

   <!-- Indicates whether the ADF bindings use generic classes or JClient-specific

        classes. -->
        ClientType="Generic | JClient" >

   <Contents >
      <DataControl
         id="ClassNameDataControl | AppModuleDataControl |
                           PXWebServiceNameDataControl"

             <!-- Indicates for the ADF design time the class to use to represent
                  the data control on the Data Control Palette. -->
                  Subtype="DCBC4J | DCJavaBean | DCWebService"

             <!-- Indicates whether the data control for the business service
                  supports query-by-example. This enables the find operation choice

                  in the Data Control Palette for this data control and the

                  associated services. Not used by ADF. -->
                  SupportsFindMode="true | false"

             <!-- Indicates whether the data control for the business service
                  supports transaction semantics. This enables commit and rollback
                  operations on the data control. -->
                  SupportsTransactions="true | false"
```

```
            <!-- Oracle ADF Business Components definitions, including the

                 package, the bc4j.xcfg configuration, and the factory class. -->
                 Package="model"
                 Configuration="AppModuleLocal" >
                 FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"

            <!-- Standard Bean definitions, including the factory class,
                 the XML definition file, and the bean class file. -->
                 FactoryClass="oracle.adf.model.generic.DataControlFactoryImpl"
                 Definition="model.Class1"
                 BeanClass="model.Class1" >

            <!-- Indicates whether the Application Module synchronization will
                 use Batch mode or Immediate mode. -->
                 <Parameters >
                   <Parameter
                      name="Sync"
                      value="Batch" >
                   </Parameter>
                 </Parameters>
      </DataControl>

      <!-- References the binding containers to create from the UI Model
           definition files. This allows the .cpx file to create the binding
           context for the application at runtime. -->
           <Containee
               id="MyPage1UIModel"
               ObjectType="BindingContainerReference"
               FullName="view.MyPage1UIModel" >
           </Containee>
           <Containee
               id="MyPage2UIModel"
               ObjectType="BindingContainerReference"
               FullName="view.MyPage2UIModel" >
           </Containee>
              ...
   </Contents>
</JboProject>
```

Web Application Runtime Integration with the Oracle ADF Model Layer

A web application that relies on Oracle ADF model objects to perform data binding to a back-end business service involves the interaction of several Oracle ADF–specific components with the page's Request object. Initially, when the application is run, an ADF-specific servlet filter specified in the application's `web.xml` file is executed. The Oracle ADF filter, an instance of `oracle.adf.model.servlet.ADFBindingFilter`, reads the metadata of the `DataBindings.cpx` file and creates the Oracle ADF data binding objects. Next, the page lifecycle object, which is an implementation of the `oracle.adf.controller.lifecycle.Lifecycle` interface, intervenes to place the Oracle ADF model objects on the Request object of the page. The Oracle ADF model objects define the binding context for the web application and become accessible through a `bindings` namespace using expression language (EL) syntax in the web page like this:

```
   ${bindings.SomeBindingContainer.someBinding.inputValue}
```

This expression refers to the current value of the binding named `someBinding` in the `SomeBindingContainer` binding container in the Oracle ADF model layer binding context. The `bindings` namespace is defined in the web page when you drop any data control–bound UI component from the Data Control Palette into the page. This namespace makes the binding context accessible through EL expressions, where it is identified as `bindings` in the expression.

In order to render the data, a JSP or UIX page relies on tags in the Core JSTL tag library which support the use of a standard expression language for referencing beans and collections. For example, the rows of the Employee range binding model object are rendered with this fragment of tags and expressions:

```
 <c:forEach var="x" items="${bindings.Employees.rangeSet}">
    <tr>
       <td><c:out value="${x.Empno}"/></td>
       <td><c:out value="${x.Ename}"/></td>
       <td><c:out value="${x.Sal}"/></td>
    </tr>
</c:forEach>
```

In this example, the `rangeSet` property of the Employees range binding exposes the rows in the current range of the Oracle ADF model object as a collection. The `var="x"` attribute of the `<c:forEach>` tag assigns a looping variable named `x` and then the tags inside the loop refer to the values of the attributes in each row bean through the EL dot notation. The JSTL specification provides this object with the property `index` that tells us which row of the iteration we are on.

Note: For additional details about how the Oracle ADF model objects manage the user's interaction with the data, see Chapter 5, Overview of Oracle ADF Integration with Struts. Understanding the Oracle ADF lifecycle object, as explained in that chapter, provides the rest of the story about how the objects of the controller layer can validate model changes and process custom events, before pushing the data to the page for display.

JClient Application Runtime Integration with the Oracle ADF Model Layer

In an Oracle ADF JClient application, data binding between the Swing controls and the business services' data sources relies on the creation of a set of JClient objects that closely resemble the UI containers used to assemble the JClient forms. You can see these containers and their JClient-specific code when you use the JClient Form wizard to generate a complete application. For example, assuming a master-detail type form, based on a Dept and Emp view object, the wizard would generate the following classes:

- `FrameDeptViewEmpView1` extends `JClientFrame` (a dummy implementation of the `JClientPanel` interface)
- `MDPanelDeptViewEmpView1` extends `JPanel` and implements `JClientPanel`
- `PanelDeptView` extends `JPanel` and implements `JClientPanel`
- `PanelEmpView1` extends `JPanel` and implements `JClientPanel`

where `JPanel` is a Swing class, and `JClientFrame` and `JClientPanel` are part of JClient and constitute your application's data browsing panels.

About Data Binding in JClient

Data binding in JClient is the ability to create Swing containers and components that are bound to data in back-end business services. To enable data binding, JClient provides a small API that works with the Oracle ADF model layer. The API is exposed in the application source code through a combination of JClient bootstrap code:

- Call `loadCpx()` to load the application metadata (specified in the `DataBindings.cpx` file), which specifies a connection to the business service implementation instance (for example, a Business Components application module instance) through the Oracle ADF data control and the Oracle ADF binding context.
- Call `setBindingContext()` to make the Oracle ADF binding context available to the frame or panel.
- Call `createPanelBinding()` to create an object that will access the business service's contained data collections through Swing component models.
- Call `bindUIControl()` on the panel binding to set the Oracle ADF model for the individual components of the JClient form or panel.

To work with the Oracle ADF data bindings in your Swing application, each container (a frame or panel) must either create a panel binding object or get one from the source from which it originated. The frame that creates the first panel binding also contains the JClient bootstrap code, where the connection to the business services is created. Subsequent containers that your application creates either chain from the original panel binding or they create their panel binding in order to display unrelated data. How you want to partition the data views of your application determines whether a container sets a new panel binding or whether it gets an existing one:

- If you want to create independent branches of the business services views, then your application should open a frame that sets a new panel binding.
- If you want to maintain the same view along a continuous branch of your application (say a master and detail branch, for example), then secondary containers all “share” the panel binding object created by the initial frame.

Generated JClient Containers

The easiest way to create databound containers is to use the JClient wizards (see the **Swing/JClient for Oracle ADF** folder in the JDeveloper New Gallery). Specifically, if you use these two JClient wizards, then the source code will contain the bootstrap code and constructors needed to create the panel binding:

- Use the Create JClient Empty Form wizard to generate an empty frame that creates a JClient panel binding with a connection to the business service used by your application, for example ADF Business Components.
- Use the Create JClient Empty Panel wizard to generate an empty panel with constructors to create a new panel binding or to share one from its parent frame.

An additional benefit to using these two wizards is their support for easy drag-and-drop UI design within JDeveloper. Because they are generated with the bootstrap code for a specific data control object (which contains the business service’s collections, structured objects, attributes, and methods), all of the Swing components that you insert from the Data Control Palette in JDeveloper will have access to any business service that the data control object contains.

Standard Java Containers

If you were to start with a standard frame or panel (one generated without using the JClient wizards) that you want to enable a JClient data binding for, you can can add the appropriate JClient bootstrap code to the main frame and then handle the panel binding in your secondary windows this way:

- If you want to share the panel binding with the parent frame:

```
BusinessCompViewName(getPanelBinding());
frame.setVisible(true);
```

- If you want the new frame to define its own panel binding:

```
BusinessCompViewName(new
              JUPanelBinding(getPanelBinding().getApplicationName(),null));
frame.setVisible(true);
```

The first call will create the frame object and set the panel binding. The second call makes the frame visible.

How JClient Preserves the Data Context Between Data Panels

The JClientPanel interface implemented by JClientFrame or JPanel permits your JClient application to:

- Maintain a consistent data context between the databound panels (also known as *chaining* between data panels)
- Access data through databound Swing controls

During design time, each data browsing panel you add to the JClient application gets its context for marshaling interactions between the UI controls and the business service's row set iterator from the panel binding object created in the frame or containing panel (such as the master-detail layout panel). The capability in JClient to chain data browsing panels is provided without the need to write additional code. For example, the data browsing panels generated by the wizard, `PanelDeptView` and `PanelEmpView1`, share the same data context through an instance of a panel binding (`JUPanelBinding`) when each JPanel implements the `setPanelBinding()` and `getPanelBinding()` methods of the `JClientPanel` interface.

Once you have a frame or panel that creates this panel binding, JClient permits you to assemble the application by adding new data browsing panels that either share the existing panel binding object or create a new one.

Then you can use the Data Control Palette in JDeveloper to add databound controls one by one to the data panel. At the level of the Swing component, this sets the data binding by specifying a JClient control model on the control's document or model property. At runtime, each control in the data panel becomes databound through the panel binding object as an argument to the control's `setModel()` or `setDocument()` method.

Process for Creating and Using the Panel Binding

To understand how the panel binding is created and used by the databound panels, consider what happens when you run the application, starting with the JClient frame, and the following JClient code is executed:

1. The `main()` method bootstraps the application. It starts a binding context and loads the Oracle ADF data control, based on entries in the `DataBindings.cpx` file. Then it passes the binding context with initialized Oracle ADF model objects to the panel binding to create the Oracle ADF data bindings.
2. The frame is initialized (`FrameDeptViewEmpView1`, in the example above) through a constructor that takes an application object. Initialization of the frame results in a panel binding object (`JUPanelBinding`), based on an Oracle ADF model definition that may have components that are bound to data from more than one data control. The creation of the panel binding is an important part of the JClient functionality, which enables data binding for Swing components and chaining of data panels.
3. The frame or applet class initializes a layout panel (`MDPanelDeptViewEmpView1`, in the example above) and sets the panel binding on the new layout panel, using the `setBindingContext()` method.

4. In the layout panel's `jbInit()` method, the data browsing (children) panels are created. For this, JClient uses the shared binding context for binding the child data panels (`PanelDeptView` and `PanelEmpView1`, in the example above).
5. A control-to-attribute data binding occurs using the control's specified JClient model. (This binding information is stored in the binding container XML metadata.)
6. The control binding handles events to populate and update data for the UI control.

About the Frame Class in JClient

Application Bootstrap

When you select the frame class in the navigator and choose **Run**, the `main()` method "bootstraps" the application. It starts a binding context and loads data controls, based on entries in the `DataBindings.cpx` file. Then it passes the binding context with initialized data controls to the panel binding to create the Oracle ADF data bindings.

The following code shows the bootstrap code created by the Create Form wizard, using selected columns from the Employees and Departments tables from the HR schema:

```
// bootstrap application
JUMetaObjectManager.setBaseErrorHandler(new JUErrorHandlerDlg());

// Lookup the *.cpx file and create all data controls listed in this file.
JUMetaObjectManager mgr = JUMetaObjectManager.getJUMom();

// Use the definition classes provided by JClient. Change only if you do not want
// to use custom DefClasses.
mgr.setJClientDefFactory(null);

// Create a new binding context that extends java.util.Hashtable.
BindingContext ctx = new BindingContext();

// Get user connection information if available. If not, display logon dialog.
ctx.put(DataControlFactory.APP_PARAM_ENV_INFO, new JUEnvInfoProvider());

// Set locale to the default locale of the JVM.
ctx.setLocaleContext(new DefLocaleContext(null));

// Load data binding container data binding file.
HashMap map = new HashMap(4);
map.put(DataControlFactory.APP_PARAMS_BINDING_CONTEXT, ctx);
mgr.loadCpx("DataBindings.cpx", map);

// Get handle to the Business Components application module.
DCDataControl app = (DCDataControl)ctx.get("model_AppModuleDataControl");
app.setClientApp(DCDataControl.JCLIENT);
```

```
// Despite the following line of code, attribute sets and fetches are normally
// performed in one batch operation. This requires only one network round
// trip. Attributes that aren't needed are not loaded to the client. The code
// line below is added only when using the JClient Form wizard. Declaratively
// creating the frame, starting with an empty form wizard does not add the
// following lines.
app.getApplicationModule().fetchAttributeProperties(new String[]
  {"DepartmentsView1", "EmployeesView3"}, new String[][] {{"DepartmentId",
   "DepartmentName" }, {"EmployeeId", "FirstName", "LastName" "DepartmentId" }},
   null);

// Initialize application root class.
FormDepartmentsView1EmployeesView3 frame = new
FormDepartmentsView1EmployeesView3();

// Set binding context to the frame.
frame.setBindingContext(ctx);
frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
Dimension frameSize = frame.getSize();
```

Frame Initialization

The frame is intialized by its constructor, which does not expect any arguments by default. The binding context of the application is passed to the `setBindingContext()` method of the frame.

Initialization of the frame results in a panel binding object (`JUPanelBinding`) based on an Oracle ADF model definition that may have components that are bound to data from more than one data control. The creation of the panel binding is an important part of the JClient functionality, which enables data binding for Swing components and chaining of data panels.

After you lay out the data panel or form, you may improve the performance of your JClient application by defining the `fetchAttributeProperties()` method in your form. This will ensure that your form performs in batch mode to fetch attribute values.

About the Layout Panel in JClient

When you use the Create JClient Form wizard to generate a JClient application with master and detail panels based on an Oracle ADF Business Components data model, the wizard generates a container panel within a JClient frame. This panel is known as the layout panel because it groups several data panels together. In addition to functioning as a UI container for one or more data browsing panels, the layout panel is able to maintain the data context for the contained data panels through its shared binding context.

Note: While the layout panel is generated by the Create JClient Form wizard, it is not an essential part of the JClient application. It is described in this topic primarily to demonstrate how the JClient application maintains a data context between data browsing panels through a shared binding context.

The binding context from the application frame can be passed to its contained JClient panels by a call to the panel's `setBindingContext()` method:

```
// get the binding context from the frame
BindingContext _bctx = panelBinding.getBindingContext();

// pass the context to the first child panel
dataPanel.setBindingContext(_bctx);

//alternatively you can use
dataPanel.setBindingContext(panelBinding.getBindingContext());
```

About Data Panels in JClient

A data browsing panel contains controls through which the user can view and edit data. Thus, it has a set of controls declared and instantiated as fields. The data browsing panel receives its panel binding from the parent frame or panel (through a `setBindingContext()` call):

```
panel.setBindingContext(panelBinding.getBindingContext());
```

After the parent container creates the data browsing panel and its panel binding, `jbInit()` is called. In the `jbInit()` method, the control is bound to attributes. Examine the following code:

```
textFieldDeptName.setDocument((Document)panelBinding.bindUIControl
     ("DepartmentName, "mDepartmentName"));
```

In the above code line, `mDepartmentName` is a `JTextField` component that is bound to the `DepartmentName` attribute of the underlying business service, where the identifier `DepartmentName` is a reference to a definition in the `UIModel.xml` file (the file defines the binding container). The binding container keeps a list of iterator bindings. Each iterator binding specifies the view object instance and (optionally) the row set iterator.

Thus, at runtime when `setDocument()` is called, JClient looks for a control binding by the specified name (`DepartmentName`). If one is found in the binding context for the form, JClient uses that control binding's associated iterator binding to access the value.

About Control Binding in JClient

Populating Controls with Data

After data browsing panels are initialized, the layout panel calls `executeIfNeeded()` on the panel binding to execute the query on the Business Components data source.

This `executeIfNeeded()` method determines whether the query had been executed on the view object, and if not, the method calls `executeQuery()` on it. This executed query brings data from the database into the cache and causes the Oracle ADF Business Components row set listener events to fire. The first among these would be the `RowSetListener.rangeRefreshed` event. This event is captured by the iterator binding (because it implements `RowSetListener` and has registered itself as a listener). It retrieves the rows of the range and calls `updateValuesFromRows()` on the control binding. The control binding takes the data out from the rows and assigns them to the controls using the Swing API. As a result, the Swing API updates the panel UI with the data.

Updating Data Through Controls

The user’s interaction with a JClient-bound control may cause the Oracle ADF Business Components to update the data. For example, in the case of the text field (`textFieldDname`), if the user edits the text field’s content and leaves the control (generating `focusLost` event), JClient is notified of the event. As a result, JClient will retrieve the updated data from the control and call `setAttribute()` on the row.

Best Practices

Customizing the Oracle ADF Iterator Binding for UI Access

You can set the number of data objects in a range to fetch from the bound data collection when you do not want to work with an entire set or when you want to display a certain number of data objects on the page. The range defines a window you can use to access a subset of the data objects in the collection. By default, the range size is set to a range that fetches just ten data objects.

Tip: In general, it is recommended that all iterator bindings referred to by multiple binding containers in one application should utilize the same range size. Utilizing the same range size prevents the binding container from generating unnecessary fetch operations against the same data collection. When your application requires different range sizes and you are using Oracle ADF Business Components, you can create a secondary row set iterator declaratively by creating an iterator binding against a given collection and providing a unique name (within the view object’s row set iterators).

To set the range size for an iterator binding:

1. With the document open in the visual editor, choose **View | Structure** to open the Structure window.
2. Click (**UI Model**) in the Structure window toolbar and expand the node to display the list of bindings.
3. Select the iterator binding for which you wish to set a range size and choose **View | Property Inspector** to open the Property Inspector.
4. In the **Range Size** field of the Property Inspector, edit the value and press Enter. The default value is `10`.

   Note that the values `-1` and `0` have specific meaning: the value `-1` returns all available objects from the collection, while the value `0` will return the same number of objects as the collection retrieves from its data source.

When you use the Data Control Palette to drop next set or previous set operations (displayed as a button component when you work with Oracle ADF Business Components) onto your page, the range size for the iterator will be set by default to fetch ten data objects at a time. This behavior will override any previous setting you may have made for the iterator. To maintain a unique range size with Oracle ADF

Business Components, you can specify a row set iterator name for the iterator binding in the Property Inspector.

To specify a secondary row set iterator for a collection (supported by Oracle ADF Business Components only):

1. With the (**UI Model**) displayed in the Structure window, select the iterator binding for which you wish to supply a unique row set iterator name.

   Supplying a unique name for the row set iterator for which the binding operates ensures that another page's iterator binding will not reset the range size on the binding container.

2. Locate the **Rowset Iterator** field of the Property Inspector, which initially has no value.
3. Type a unique identifying name for the property of the selected iterator binding and press Enter.

   At runtime, the binding container will create a unique row set iterator corresponding to the customized iterator binding.

Creating a Search Criteria Form Using Oracle ADF Find Mode

When you create Oracle ADF–enabled web pages, you can support parameterized queries against Oracle ADF Business Components by using an input form and setting the find mode for the page's binding container to enabled. The Oracle ADF binding container supports find operations by executing a parameterized query using the search criteria specified in the form against the view object specified by an Oracle ADF iterator binding.

Once the find operation is executed, the binding container is taken out of find mode and the web page functions as an input entry form. In this way, the binding container toggles the find mode between enabled and disabled for a specific web page.

About Parameterized Queries

A parameterized query is a query that contains a placeholder that must be supplied at runtime. For example, in the following PL/SQL statement, `min_salary` is a placeholder for a parameter value that will be supplied at runtime:

```
SELECT ename, job, mgr FROM emp WHERE sal < :min_salary
```

The input form in find mode uses the Oracle ADF bindings to display fields for each attribute in the bound Oracle ADF Business Components view object whose `Queriable` property is set to `true`. The view object defines the initial query executed by the business components.

Process for Displaying Results

In a Struts-based web application, the user interacts with an input form with find mode as follows:

1. The web page with input form displayed by the user runs with find mode enabled.

   For instance, a user may click a link to open a page with the find mode enabled.

2. The user enters search criteria to restrict the results of the data.

   The user can enter comparison symbols (>, <, =) as part of the search criteria. All values in the same view criteria participate in the search.

3. The user clicks an **Execute** button on the form, which initiates a find operation on a Struts action to perform an anchored, wild card search.

   The operation uses the first character of the search column as an anchor, where all the strings that begin with the entered string are matched.

4. The Struts action forwards to another page, where a read-only table displays the results of the parameterized query.

Summary of UI Components in Oracle ADF Web Pages

The client developer uses the Data Control Palette to insert already databound UI components into their web page:

- In the case of Model 1 JSP pages (which do not use the Struts page flow), visual elements that you select will appear in the JSP page as code snippets that use a combination of Oracle ADF tags (a custom tag library), HTML tags, and EL (expression language) syntax.
- In the case of Struts-based JSP pages, visual elements that you select will appear in the JSP page as code snippets that use a combination of Struts tags (for Struts-based web application), JSTL tags, and EL (expression language) syntax.
- In the case of UIX pages, visual elements that you select are UIX elements, runtime components represented in the UIX page by XML syntax.

  Note: The Data Control Palette detects the type of web application your project defines and displays the appropriate components for a Struts-based JSP project, a Model 1 JSP project, or a UIX project.

The remainder of this section describes the visual elements that you can select from the Data Control Palette.

Value Bindings for the Entire Collection or Row Set

The Data Control Palette provides UI components for web pages that you can use to bind an entire data collection (which consists of a data object that comprise a row set), as shown in the following table.

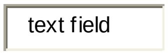
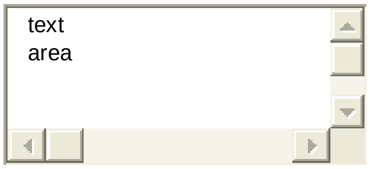
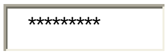
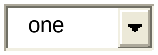
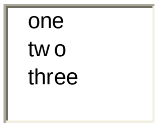
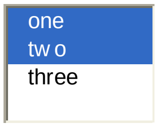
Drag and Drop into a JSP Page As	Drag and Drop into a UIX Page As	ADF Binding Type
Read-Only Table / Dynamic Table (used when the bean has no scalar attributes)	Read-Only Table	Table binding Note that in the case of the dynamic table (JSP page only), all attributes of the selected collection will be displayed by the table

Read-Only Form	Read-Only Form	Table binding (JSP page only) Attribute bindings for the text fields (both JSP/UIX pages) Action bindings for the buttons (UIX page only)
Navigation Buttons	*not available*	Action bindings
Graph	*not available*	Graph binding
Input Form	Input Form	Attribute bindings
not available	Input Form (with Navigation)	Attribute bindings for the text fields Action bindings for the buttons
not available	Search Form	Table binding for the table Attribute bindings for the text fields Action bindings for the buttons
not available	Master Detail (Self)	Table bindings for the table Attribute bindings for the text fields
Selected Row Link	*not available*	*not applicable*
Navigation List	encodedparameter	List binding in navigation mode
not available	textinput (secret)	textinput (secret)

Value Bindings for Individual Data Object Attribute Values

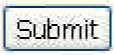
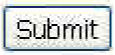
The Data Control Palette provides UI components that you can use to bind a single data object attribute, as shown in the following table.

UI Component	Drag and Drop into a JSP Page As	Drag and Drop into a UIX Page As
Hello	Value	*not available*
For Oracle ADF Business Components, displays label control hint, all other services display the attribute value as a label	Label Note that the label can be defined by Control Hints in the case of Oracle ADF Business Components	*not available*

not visible	Input Render Note that custom renders can be defined in the case of Oracle ADF Business Components	*not available*
Displays the attribute value, using a custom renderer	Render Value Note that custom renders can be defined in the case of Oracle ADF Business Components	*not available*
Label:  text field	Input Field	TextInput MessageTextInput
Label:  text area	Text Area	MessageStyledText
Renders a hidden field bound to a model object	Hidden Field	encodedparameter
not visible	File Input Field	*not available*
Label:  *********	Password Field	textinput (secret)
not visible	Render Value	*not available*
Label:  one Label:  one tw o three	Single Select List Static Single Select Field	List MessageList (Select One), MessageList (Select Many)
Label:  one tw o three	List of Values	MessageLovInput
Label: ○ smoking Label: ○ non-smoking	Radio Button Group	RadioSet MessageRadioSet
Label: □ apples Label: □ oranges	*not available*	CheckBox MessageCheckBox

Action Bindings for Business Object Methods and Data Control Operations

The Data Control Palette provides UI components that you can use to bind a method or operation, as shown in the following table.

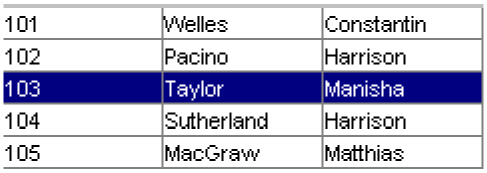
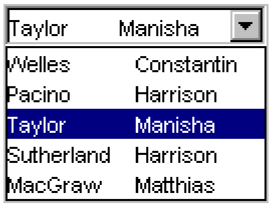
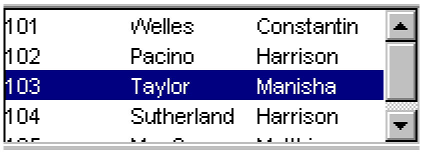
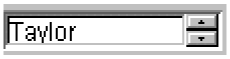
UI Component	Drag and Drop into a JSP Page As	Drag and Drop into a UIX Page As
	Button	SubmitButton
	Button with Form	*not available*
ActionRequestURI	Link (Strut-based applications only)	*not available*

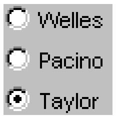
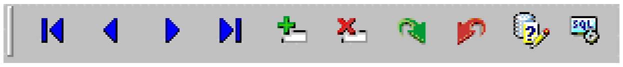
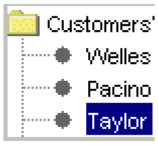
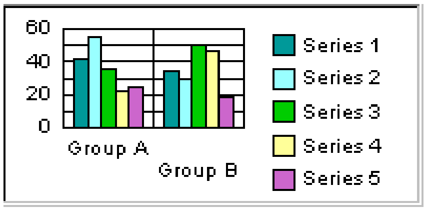
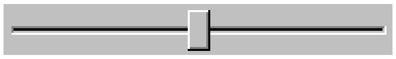
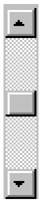
Summary of UI Components in Oracle ADF Java Clients

The client developer uses the Data Control Palette to insert already databound UI components into a JClient-prepared form or panel.

Value Bindings for the Entire Collection or Data Object

The Data Control Palette provides UI components that you can use to bind an entire data collection (which consists of data objects that comprise a row set), as shown in the following table.

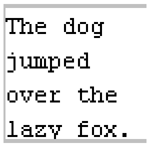
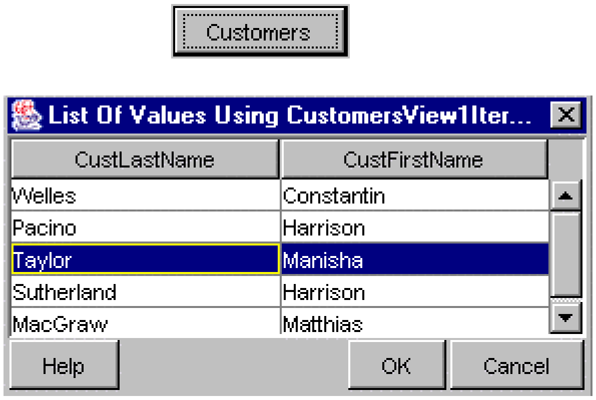
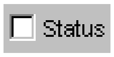
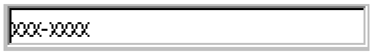
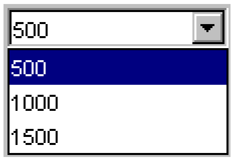
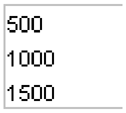
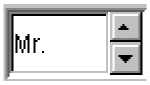
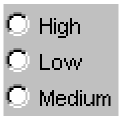
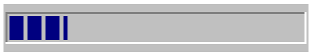
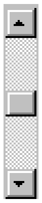
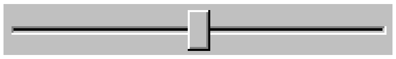
UI Component	Drag and Drop As	ADF Binding Type
101 Welles Constantin 102 Pacino Harrison 103 Taylor Manisha 104 Sutherland Harrison 105 MacGraw Matthias	Table	Table binding
Taylor Manisha Welles Constantin Pacino Harrison Taylor Manisha Sutherland Harrison MacGraw Matthias	Combo Box	List binding in navigation mode
101 Welles Constantin 102 Pacino Harrison 103 Taylor Manisha 104 Sutherland Harrison	List (inside a ScrollPane)	List binding in navigation mode
Taylor	Spinner	List binding in navigation mode

 ○ Welles ○ Pacino ● Taylor	Radio Button Group	List binding in navigation mode
	NavigationBar	Iterator binding
 Customers' Welles Pacino Taylor	Tree	Tree binding
 60 40 20 0 Group A Group B Series 1 Series 2 Series 3 Series 4 Series 5	Graph	Graph binding
	Slider	Scroll binding
	ScrollBar	Scroll binding

Value Bindings for Individual Data Object Attribute Values

The Data Control Palette provides UI components that you can use to bind a single data object attribute, as shown in the following table.

UI Component	Drag and Drop As	ADF Binding Type
 Taylor	TextField	Attribute binding
 asdfasdf	Edit Pane	Attribute binding
 Last Name	JULabel	Attribute binding
 Last Name	Label For Used with Oracle ADF Business Components to display attribute's label control hint	Attribute binding
 **********	Password Field	Attribute binding
 The dog jumped over the lazy fox.	Text Area	Attribute binding

 The dog jumped over the lazy fox.	Text Pane	Attribute binding
 List Of Values Using CustomersView1Iter... CustLastName · CustFirstName Welles · Constantin Pacino · Harrison Taylor · Manisha Sutherland · Harrison MacGraw · Matthias 	Button LOV	LOV binding
 	Check Box	Boolean binding
 xxx-xxxx	Formatted Edit Field	Formatted text binding
 	Combo Box	List binding in enumeration mode
 	List	List binding in enumeration mode
 	Spinner	List binding in enumeration mode
 	Radio Button Group	List binding in enumeration mode
	Progress Bar	Bounded range binding
	Scroll Bar	Bounded range binding
	Slider	Bounded range binding

Appendix A, JDeveloper Runtime Problems and Solutions

This appendix describes common problems and solutions. It contains the following topics:

A.1.1 JSP Page Fails with HTTP 404-Page Not Found Error

You have successfully deployed the ADF web application with JSP files in the WAR file, but the browser displays the error `HTTP 404-Page Not Found` when you attempt to run the application.

Problem

The URL of the web page does not match the context root configured for the application server. The context root of the application may differ because JDeveloper has picked up a default that you will have to override. If you deployed the WAR file using Enterprise Manager, then the context root specified by Enterprise Manager is the one picked up by Oracle Application Server.

Solution

Check that the URL for the web page follows this format:

```
http://<host>:<http port for iAS>/context-root/<sub-directory structure
within public_html>/<the page>.jsp
```

The context root for the application is available in the `http-web-site.xml` or `default-web-site.xml` file located in `<mypath>/j2ee/home/config` on the Oracle Application Server installation.

For example, with a context root like `/war1`, the `mod_oc4j.conf` file in the Oracle Application Server installation has the following entry:

```
Oc4jMount /war1 home
Oc4jMount /war1/* home
```

In other words, the URL you see should have only one context root, followed by any subdirectories.

To modify the context root for your WAR file in JDeveloper:

1. In the navigator, double-click **webapp*x*.deploy** and select **General** in the WAR Deployment Profile Properties dialog.
2. Select **Specify J2EE Web Context Root** and enter the value in the field.
3. Redeploy the WAR file to the Oracle Application Server installation.

A.1.2 Browser Locates JSP File But Fails to Render with Content

You have successfully deployed the ADF web application with JSP files and the `Struts-Config.xml` file, yet when you attempt to run the application, the page appears empty. Additionally, the URL of the web page is correct and the browser displays no HTTP errors.

Problem

Your web application relies on the Struts controller to forward to a page from a Struts action. Your application implements the Struts action class in order to prepare the data for the page before rendering. You have attempted to run the web page without first executing the appropriate Struts action class.

Solution 1

Do not run the Struts-based web application by supplying the URL of the JSP page. You must run the application by invoking the Struts action. Typically this is accomplished by displaying a web page that contains a link with the name `<`*`action name`*`>.do`. The extension `.do` redirects the link to the Struts controller, which executes the corresponding action from the `Struts-Config.xml` file. The browser displays the web page mapped to the action in the `Struts-Config.xml` file.

Solution 2

When you want to run your web application within JDeveloper, do not run the JSP directly. While JDeveloper does allow you to choose **Run** on any JSP file in your Strut-based web application, you must invoke the web page from the `Struts-Config.xml` file. Right-click **Struts-Config.xml** in the navigator and choose **Run** corresponding to the desired action. The action you choose will allow the Struts controller to execute a corresponding, mapped JSP file.

A.1.3 JDeveloper Unable to Establish Connection to Embedded OC4J Server

You have created a web application, but the application fails to run when JDeveloper attempts to establish a connection to the embedded OC4J server. The error message you get originates from your proxy server.

Problem

By default, JDeveloper uses the proxy settings from the default browser on the same machine. However, if `localhost` and `127.0.0.1` do not appear in the list of proxy exceptions, and you have not specified these exceptions within JDeveloper, when you attempt to connect to an application server residing on the same machine as JDeveloper, the connection intended for your local machine may actually be opened on the proxy server instead.

For example, if `localhost` is not excluded from the proxy list, a request in your browser like `http://localhost/MyApp/index.html` will be sent to the proxy server, and the server will resolve `localhost` to itself, rather than to your machine. In this case, the error message you see is actually returned by the proxy server rather than by any server running on your local machine.

Solution

If you are connecting to an IP address behind a proxy server, and your machine is also behind the same proxy server, then make sure that JDeveloper's web proxy preferences exclude the IP address you are trying to connect to.

To verify and modify the proxy preferences in JDeveloper:

1. From the **Tools** menu, choose **Preferences** and select **Proxy Server** from the dialog.
2. Be sure that **Use HTTP Proxy Server** is selected and specify the proxy port and proxy host.

To verify that your proxy settings are being picked up, start JDeveloper and observe the command that the console window displays to start the embedded OC4J server. The command will contain the proxy settings.

A.1.4 Unable to Specify Connection Driver Class to Use with a Web Application in JDeveloper

You have created a web application and you want to specify custom driver classes (such as `oracle.jdbc.pool.OracleConnectionPooldata source` or `oracle.jdbc.pool.OracleConnectionCacheImpl`) for your data sources, but the default connection details specified by the IDE connections are used instead. You want to be able to specify the class and location parameters in a customized manner that does not rely on IDE connection definitions.

Problem

The `data-sources.xml` file located in your project may be used to run and debug the application in the case of embedded OC4J or to initiate deployment to standalone Oracle Application Server installation. In JDeveloper, the `data-sources.xml` file is overwritten with connections defined in the Connection Manager wizard. The action of overwriting the `data-sources.xml` file always deletes any user-defined data sources; it also defines as data sources all database connections defined in the Connection Manager, whether the application requires them or not.

Note that this problem does not exist at the level of the global application when you want to define custom entries that will apply server-wide. You may edit the global application `data-sources.xml` file located in the OC4J directory `<ORACLE_HOME>/j2ee/config/`. Unlike the project-level `data-sources.xml` file, JDeveloper will not overwrite this file.

Solution for JDeveloper 9.0.4.*x* and Earlier

Edit the class drivers in the `data-sources.xml` file to specify the custom entries, and set the file to read-only from the Windows Explorer. This will prevent JDeveloper from overwriting the file again. The file has two locations depending upon what you want to control:

- The project-level `data-sources.xml` file is located in `<`*`JDEV_HOME`*`>/mywork/<`*`project`*`>/src/META-INF/`. Modify this file when you are ready to deploy the application with your data source definitions.
- The embedded OC4J server–level `data-sources.xml` file is located in `<`*`JDEV_HOME`*`>/system`*`XXX`*`/oc4j-config/`. Modify this file when you want to run the application in JDeveloper using your custom data source definitions.

To add user-defined data sources, specify a `name` attribute in the `<data-source>` element. For example:

```
<data-source name="myConnection" ... />
```

In 9.0.3.*x*, JDeveloper recognizes `name` attribute values starting with `jdev-connection`: and maps them to connections defined in the Connection Manager.

In 9.0.4, the prefix used by JDeveloper changed to `jdev-connection-`, but JDev 9.0.4 will also recognize `jdev-connection:`, so both prefixes map to Connection Manager connections.

Based on the `name` attribute, three possible behaviors exist:

- If `name` starts with a recognized prefix, then JDeveloper will automatically update the `<data-source>` element with any changes from the Connection Manager.

- If `name` does not start with a recognized prefix, then JDeveloper will not touch the `<data-source>` element. This is the recommended solution when you want to define your own data source connection information.
- If `name` is not specified at all, then JDeveloper will remove the `<data-source>` element.

JDeveloper packages an `orion-application.xml` file, which in turn points to the `data-sources.xml` file in your project.

To ensure that your `data-sources.xml` file is deployed with your project, you must configure the deployment profile for Standard J2EE:

1. In the navigator, double-click **Xxx.deploy** to display the Deployment Profile Properties dialog.
2. In the dialog, select **Platform** and set **Target Connection** to the value **<None>**.

   The OC4J-specific files will no longer be deployed as a part of the archives.

Solution for JDeveloper 9.0.5.*x* and Later

Starting in JDeveloper 9.0.5.*x*, you can use the JDeveloper IDE to specify the values for data sources and options for synchronizing these data sources with IDE connection definitions. The `data-sources.xml` file that the IDE will update for you has two locations depending upon what you want to control:

- The project-level `data-sources.xml` file is located in `<`*`JDEV_HOME`*`>/mywork/<`*`project`*`>/src/META-INF/`. Update this file when you are ready to deploy the application with your data source definitions.
  1. In the navigator, right-click **data-sources.xml** and choose **Properties**.
  2. In the Data Sources Properties dialog, select **Data Sources**.
  3. Deselect the option **Auto-update data-sources.xml when running or deploying to OC4J**.

     This will prevent JDeveloper from overwriting the file again. Alternatively, you can deselect specific options to create, update, or delete definitions in the `data-sources.xml` file based on the IDE connection.
  4. To specify the desired connection driver class, select the desired connection from the **Data Sources** list on the left.
  5. In the **Connection** tab, specify the desired classname.
- The embedded OC4J server–level `data-sources.xml` file is located in `<`*`JDEV_HOME`*`>/system`*`XXX`*`/oc4j-config/`. Update this file when you want to run the application in JDeveloper using your custom data source definitions.
  1. From the **Tools** menu, choose **Embedded OC4J Server Preferences**.
  2. In the dialog, select **Current Workspaces** and **Data Sources**.

You must deselect each option to create, update, or delete definitions in the `data-sources.xml` file based on the IDE connection. This will prevent JDeveloper from overwriting the file again.

3. To specify the desired connection driver class, select the desired connection from the **Data Sources** list on the left.
4. In the **Connection** tab, specify the desired classname.

JDeveloper packages an `orion-application.xml` file, which in turn points to the `data-sources.xml` file in your project.

To ensure that your `data-sources.xml` file is deployed with your project, you must configure the deployment profile for Standard J2EE:

1. In the navigator, double-click **Xxx.deploy** to display the Deployment Profile Properties dialog.
2. In the dialog, select **Platform** and set **Target Connection** to the value **<None>**.

   The OC4J-specific files will no longer be deployed as a part of the archives.

A.1.5 Unable to Establish Connection Upon EJB Lookup

You have created a J2EE application that relies on EJB lookup on Oracle Application Server, but the connection fails when you:

- Try to connect to the server using the Create Application Server Connection wizard
- Try to deploy the WAR file to the server using the connection
- Try to run the application and the client attempts to access a component on the server

The error might be `Connection refused: connect` or it might be `java.net.ConnectException: Connection refused: connect Io exception: Connection refused: connect (DESCRIPTION=(TMP=)(VSNNUM=135286784)(ERR=12505)(ERROR_STACK=(ERROR=(CODE=12505)(EMFI=4))))`.

Problem 1

If the problem is not a SQL exception, and the message `Connection refused: connect` is displayed, then it is possible that the OC4J server is listening on a different RMI port. Otherwise, the OC4J container is either not up or it is not listening on the host specified.

Not being able to establish a connection is a common problem on the standalone Oracle Application Server installation because RMI ports are not supplied as defaults but are picked up from a specified range that you configure.

Solution for Embedded OC4J Server in JDeveloper

To obtain and verify the port that JDeveloper is using when you run or deploy your application, view the message log window. One of the first messages to appear should look like this:

```
[Starting OC4J using the following ports: HTTP=8988, RMI=23891,
JMS=9227.]
```

Alternatively, you can check the `rmi.xml` file to ensure that the OC4J server is listening on the same RMI port. You can also check for errors in the `rmi.log` file in `<`*`JDEV_HOME`*`>/system`*`XXX`*`/oc4j-config/log/`.

To modify the port in JDeveloper versions 9.0.2 through 9.0.4:

1. From the **Tools** menu, choose **Preferences**.
2. In the dialog, select **Embedded OC4J** to make your changes.

To modify the port in JDeveloper 9.0.5.*x*:

1. From the **Tools** menu, choose **Preferences**.
2. In the dialog, select **Embedded OC4J Server Preferences**.
3. In the dialog, select **Global** and **Startup** to make your changes.

The new server port settings will be updated in the `rmi.xml` file located in `<`*`JDEV_HOME`*`>/system`*`XXX`*`/oc4j-config/`.

Solution for Standalone OC4J on Oracle Application Server

Check the `rmi.xml` file to ensure that the OC4J server is listening on the same RMI port. You can also check for errors in the `rmi.log` file in `<`*`OC4J_HOME`*`>/j2ee/home/log/`. Edit the server port in the `rmi.xml` file located in `<`*`OC4J_HOME`*`>/j2ee/home/config/`.

Alternatively, access Enterprise Manager’s web site for administering the Oracle Application Server installation (type `http://host:port` — defaults to `1810`). Click the **Ports** link in the first page for the OC4J instance. Then use the page to check and configure the RMI ports for each of the OC4J instances.

Appendix B, Oracle ADF Problems and Solutions

This appendix describes common problems and solutions. It contains the following topics:

- ADF Runtime Installer Fails with Error
- Previously Working Application Using ADF Business Components Starts Throwing JDBC Errors
- Changes to ADF Business Components Parameters Have No Effect
- ADF Business Components Throw `ClassNotFoundException`
- ADF Business Components Deployed with Libraries Throw Exceptions

B.1.1 Oracle ADF Runtime Installer Fails With Error

When using the Oracle ADF Runtime Installer in JDeveloper 9.0.5.*x* from a remote machine to upgrade the Oracle ADF libraries on Oracle Application Server, you may receive the error `The selection is not an Oracle Application Server home directory.` in the ADF Runtime Installer wizard.

Problem

Before you deploy and run your application, you must ensure that the ADF runtime libraries that reside on the target Oracle Application Server installation are the same version, or later, as the libraries that were used to develop the application in JDeveloper. While you attempt to run the ADF Runtime Installer, the ADF runtime libraries are not available on the same machine as the target Oracle Application Server installation. In this release, the ADF Runtime Installer is designed to run on the same machine as the target Oracle Application Server installation. In order to run the ADF Runtime Installer, you must be able to obtain the libraries from the correct JDeveloper installation.

Before performing any updates, verify that your application server is supported by JDeveloper 9.0.5. See the chart provided in the document at this link:

http://otn.oracle.com/products/jdev/collateral/papers/10g/as_supportmatrix.html

Solution 1

Install JDeveloper on the same machine as the target Oracle Application Server installation and rerun the ADF Runtime Installer. In this case, because JDeveloper resides on the same machine as the Oracle Application Server installation, the ADF Runtime Installer can find Oracle Home and will automatically locate the ADF runtime libraries from JDeveloper.

It is important that the ADF runtime libraries that reside on the target Oracle Application Server installation are the same version, or later, as the libraries that were used to develop the application in JDeveloper. Be sure to install any maintenance releases of JDeveloper before you upgrade the target environment libraries.

Solution 2

Install JDeveloper on any machine, and then move the required ADF runtime libraries to the target Oracle Application Server machine.

It is important that the ADF runtime libraries that reside on the target Oracle Application Server installation be the same version, or later, as the libraries that were used to develop the application in JDeveloper. Be sure to install any maintenance releases of JDeveloper before you upgrade the target environment libraries.

Note: This solution does not rely on the ADF Runtime Installer. Instead use the following list of runtime libraries to update the application server.

Before you install the ADF runtime libraries:

1. Optionally, create a backup directory of each directory that you plan to update (see the list below).
2. Stop *all* OC4J instances, including the Enterprise Manager instance. Only after completing the installation should you restart the server.
3. If your application works with *TopLink* mapping objects, you must edit the `<ORACLE_Home>/j2ee/home/config/application.xml` file to include the following library paths:

   ```
   <library path="../../../jlib/ojmisc.jar" />
   <library path="../../../toplink/jlib/toplink.jar" />
   <library path="../../../toplink/jlib/antrl.jar" />
   ```

   Only TopLink users must edit the `application.xml` file on the server; the other supported business services do not require this modification.

After you have completed the above steps, you must remove these four files from the existing Oracle Application Server installation in the OC4J root directory:

```
<ORACLE_Home>/BC4J/lib/datatags.jar
<ORACLE_Home>/BC4J/lib/bc4juixtags.jar
<ORACLE_Home>/BC4J/lib/bc4jhtml.jar
<ORACLE_Home>/BC4J/lib/bc4j_jclient_common.jar
```

Then you can copy the files shown in the following tables from the JDeveloper source location to the OC4J root directory.

Copy these ADF runtime libraries:

From JDeveloper	To Server
<*JDEV_Home*>/BC4J/lib/adfm.jar	<*ORACLE_Home*>/BC4J/lib/adfm.jar
<*JDEV_Home*>/BC4J/lib/adfmweb.jar	<*ORACLE_Home*>/BC4J/lib/adfmweb.jar
<*JDEV_Home*>/BC4J/lib/bc4jct.jar	<*ORACLE_Home*>/BC4J/lib/bc4jct.jar
<*JDEV_Home*>/BC4J/lib/bc4jctejb.jar	<*ORACLE_Home*>/BC4J/lib/bc4jctejb.jar
<*JDEV_Home*>/BC4J/lib/bc4jdomorcl.jar	<*ORACLE_Home*>/BC4J/lib/bc4jdomorcl.jar
<*JDEV_Home*>/BC4J/lib/bc4jimdomains.jar	<*ORACLE_Home*>/BC4J/lib/bc4jimdomains.jar
<*JDEV_Home*>/BC4J/lib/bc4jmt.jar	<*ORACLE_Home*>/BC4J/lib/bc4jmt.jar
<*JDEV_Home*>/BC4J/lib/bc4jmtejb.jar	<*ORACLE_Home*>/BC4J/lib/bc4jmtejb.jar
<*JDEV_Home*>/BC4J/jlib/adfjclient.jar	<*ORACLE_Home*>/BC4J/jlib/adfjclient.jar
<*JDEV_Home*>/BC4J/lib/collections.jar	<*ORACLE_Home*>/BC4J/lib/collections.jar
<*JDEV_Home*>/BC4J/jlib/bc4jdomgnrc.jar	<*ORACLE_Home*>/BC4J/jlib/bc4jdomgnrc.jar
<*JDEV_Home*>/jlib/jdev-cm.jar	<*ORACLE_Home*>/jlib/jdev-cm.jar
<*JDEV_Home*>/BC4J/lib/adfmtl.jar	<*ORACLE_Home*>/BC4J/lib/adfmtl.jar

Copy this OJMisc runtime library:

From JDeveloper	To Server
<*JDEV_Home*>/jlib/ojmisc.jar	<*ORACLE_Home*>/jlib/ojmisc.jar

Install these *inter*Media Runtime Libraries:

From JDeveloper	To Server
<*JDEV_Home*>/ord/jlib/ordhttp.jar	<*ORACLE_Home*>/ord/jlib/ordhttp.jar
<*JDEV_Home*>/ord/jlib/ordim.jar	<*ORACLE_Home*>/ord/jlib/ordim.jar

Copy these TopLink runtime libraries:

From JDeveloper	To Server
<*JDEV_Home*>/toplink/jlib/toplink.jar	<*ORACLE_Home*>/toplink/jlib/toplink.jar
<*JDEV_Home*>/toplink/jlib/antlr.jar	<*ORACLE_Home*>/toplink/jlib/antlr.jar

Copy these BC4J EAR application files:

From JDeveloper	To Server
<*JDEV_Home*>/BC4J/redist/bc4j.ear	<*ORACLE_Home*>/BC4J/redist/bc4j.ear
<*JDEV_Home*>/BC4J/redist/bc4j.ear	<*ORACLE_Home*>/j2ee/home/applications/BC4J.ear

Solution 3

Map a local drive from a machine that already has JDeveloper installed to the target Oracle Application Server machine and rerun the ADF Runtime Installer. Use the mapped drive for the JDeveloper installation.

Be sure to install any maintenance releases of JDeveloper when you want to upgrade the target environment libraries.

B.1.2 Previously Working Application Using ADF Business Components Starts Throwing JDBC Errors

An application that previously successfully retrieved data suddenly starts throwing JDBC errors such as `Connection Reset By Peer`, `Connection Closed`, or `Socket Reset By Peer`.

Problem

The connections in the pool have become stale. This can happen for any of the following reasons:

- The database was shut down or restarted without a corresponding restart of the JVM running the business components.
- The connections were timed out by a firewall.
- There were network problems.

Stale connections, when accessed, will throw errors.

Solution

If your ADF Business Components are deployed to Oracle Application Server 10*g*, you can set the parameter `clean-available-connections-threshold` to periodically clean up stale connections.

B.1.3 Changes to ADF Business Components Parameters Have No Effect

You have changed ADF Business Components runtime parameters, but the new parameters appear not to have taken effect.

Problem

ADF Business Components runtime parameters can be specified in several separate locations. A location with a higher precedence is overriding your changes. Runtime properties can be specified in the following locations, in descending order of precedence:

- The ADF application module configuration being used by the client application

- Applet tags
- `-D` flags passed to the JVM
- The `bc4j.properties` file in the directory holding your business components
- The `/oracle/jbo/BC4J.properties` resource
- The `/oracle/jbo/common.jboserver.properties` resource
- The `/oracle/jbo/common.Diagnostic.properties` resource
- The ADF BC library's own defaults

Solution

Check the locations with higher precedence to ensure that they are not overriding your changes.

B.1.4 ADF Business Components Throw ClassNotFoundException

When your application attempts to access business components, it throws a class not found exception that mentions an ADF BC framework class. You can diagnose the cause of this problem by searching for the JAR file which contains the class mentioned in the exception.

Problem 1

Your application was designed against a newer version of the ADF BC libraries than is available on the server, and the old version does not contain some of the classes your application is expecting.

Solution 1

Use the ADF Runtime Installer to install a newer version of the ADF BC libraries on the server.

Problem 2

A JClient application has been distributed without the required libraries in its archive.

Solution 2

Redeploy the JClient application with the missing libraries.

Problem 3

The archive containing the needed class is not on the classpath.

Solution 3

Ensure that the OC4J classpath includes the archive containing the needed class.

B.1.5 ADF Business Components Deployed with Libraries Throw Exceptions

You have deployed an application with ADF Business Components to a version of Oracle Application Server 10*g* with a different version of the ADF Business Components libraries installed. Even though you deployed the appropriate version of the libraries with your application, the application continues to throw exceptions as if it were attempting to run against the incorrect version.

Problem

The libraries installed on the application server appear earlier in the classpath than the appropriate libraries included in the application's EAR file.

Solution

Check the class loader hierarchy in the `server.xml` and `orion-web.xml` files to ensure that the libraries your application needs are loaded first. Ideally, you should avoid this problem by installing the latest version of the libraries using the ADF Runtime Installer.