**Oracle® Database**

2 Day + Java Developer's Guide

11*g* Release 1

**B28765-03**

February 2012

**ORACLE**®

Oracle Database 2 Day + Java Developer's Guide, 11*g* Release 1

B28765-03

Primary Authors:  Deepa Aswani, Rosslynne Hefferan, Maitreyee Chaliha

Contributing Authors:  Kathleen Heap, Simon Law

Contributors:  Kuassi Mensah, Chris Schalk, Christian Bauwens, Mark Townsend, Paul Lo, Venkatasubramaniam Iyer

# Contents

# 3    Connecting to Oracle Database

# 4    Querying for and Displaying Data

## 5    Updating Data

## 6    Enhancing the Application: Advanced JDBC Features

## 7   Creating a Master-Detail Application Using Oracle ADF

## 8   Getting Unconnected from Oracle Database

## 9   Building Global Applications

## Index

## List of Examples

## List of Figures

# List of Tables

# Preface

This Preface introduces you to *Oracle Database 2 Day + Java Developer's Guide*, by discussing the intended audience and conventions of this document. It also includes a list of related Oracle documents that you can refer to for more information.

## Audience

This guide is intended for application developers using Java to access and modify data in Oracle Database. This guide illustrates how to perform these tasks using a simple Java Database Connectivity (JDBC) application. This guide uses the Oracle JDeveloper integrated development environment (IDE) to create the application. This guide can be read by anyone with an interest in Java programming, but it assumes at least some prior knowledge of the following:

- Java
- Oracle PL/SQL
- Oracle databases

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

### Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

### Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

### Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at

`http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

## Related Documents

For more information, see the following documents in the Oracle Database documentation set:

- *Oracle JDeveloper Installation Guide*, *10g Release 10.1.3* and JDeveloper Online Documentation on Oracle Technology Network at

  `http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html`

- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database Java Developer's Guide*

## Conventions

The following text conventions are used in this document:

| Convention | Meaning |
|------------|---------|
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |

# 1

# Using Java with Oracle Database

Oracle Database is a relational database that you can use to store, use, and modify data. The Java Database Connectivity (JDBC) standard is used by Java applications to access and manipulate data in relational databases.

JDBC is an industry-standard application programming interface (API) developed by Sun Microsystems that lets you embed SQL statements in Java code. JDBC is based on the X/Open SQL Call Level Interface (CLI) and complies with the SQL92 Entry Level standard. Each vendor, such as Oracle, creates its JDBC implementation by implementing the interfaces of the standard `java.sql` package.

> **See Also:**
> - http://java.sun.com/javase/technologies/database/index.jsp

This guide shows you how to use a simple Java application to connect to Oracle Database and access and modify data within the database. Further, it uses the Oracle Application Development Framework (ADF) to develop a master-detail application to display employee data.

This chapter introduces you to the Java application created in this guide, and to the tools you can use to develop the Java application in the following topics:

- Using Java to Connect to Oracle Database
- Using JDeveloper to Create JDBC Applications
- Overview of Sample Java Application

## Using Java to Connect to Oracle Database

 JDBC is a database access protocol that enables you connect to a database and run SQL statements and queries on the database. The core Java class libraries provide the JDBC APIs, `java.sql` and `javax.sql`. However, JDBC is designed to allow vendors to supply drivers that offer the necessary specialization for a particular database.

> **Note:** Oracle Database 11*g* Release 1 support JDK 5 and onward. The JDBC support in this release includes the `ojdbc5.jar` and `ojdbc6.jar` files. The `ojdbc6.jar` file offers JDBC 4.0 compliance. To use this file, you need JDK 6.

Oracle Database provides support for the client-side application development through the JDBC Thin Driver and the  Oracle Call Interface (OCI) Driver, and the `oracle.sql` and `oracle.jdbc` packages. The classes and interfaces in these packages extend the

JDBC standard. They allow you to access and modify Oracle data types and use Oracle performance extensions for JDBC with greater flexibility in a Java application.

The following sections describe Oracle support for the JDBC standard:

- Oracle JDBC Thin Driver
- Oracle JDBC OCI Driver
- Oracle JDBC Packages

> **See Also:**
>
> - *Oracle Database JDBC Developer's Guide and Reference*
> - *Oracle Database Java Developer's Guide*

## Oracle JDBC Thin Driver

Oracle recommends using the JDBC Thin Driver for most requirements. JDBC-OCI is only needed for OCI-specific features.

The JDBC Thin Driver is a pure Java, Type IV driver. It supports the Java$^{TM}$ 2 Platform Standard Edition 5.0, also known as Java Development Kit (JDK) 5. It also includes support for JDK 6. It is platform-independent and does not require any additional Oracle software for client-side application development. The JDBC Thin Driver communicates with the server using SQL*Net to access Oracle Database.

The JDBC Thin Driver allows a direct connection to the database by providing a pure Java implementation of Oracle network protocols (Two-Task Common, also known as the TTC protocol, and SQL*Net). The driver supports the TCP/IP protocol and requires a Transparent Network Substrate (TNS) listener on the TCP/IP sockets on the database server. The Thin driver will work on any machine that has a suitable Java virtual machine (JVM).

You can access the Oracle-specific JDBC features and the standard features by using the `oracle.jdbc` package.

## Oracle JDBC OCI Driver

The JDBC OCI driver is a Type II driver used with Java applications. It requires an Oracle client installation. It supports all installed Oracle Net adapters, including interprocess communication (IPC), named pipes, TCP/IP, and InternetworkPacket Exchange/Sequenced Packet Exchange (IPX/SPX).

OCI is an API that enables you to create applications that use native procedures or function calls. The JDBC OCI driver, written in a combination of Java and C, converts JDBC calls to calls to OCI. It does this by using native methods to call C-entry points. These calls communicate with the database using SQL*Net.

## Oracle JDBC Packages

Oracle support for the JDBC API is provided through the `oracle.jdbc` and `oracle.sql` packages. These packages support all Java Development Kit (JDK) releases from 1.5 through 1.6.

### oracle.sql

The `oracle.sql` package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their

support classes. Essentially, the classes act as Java wrappers for SQL data. The characters are converted to Java `chars` and, then, to bytes in the UCS-2 character set.

Each of the `oracle.sql.*` data type classes extends `oracle.sql.Datum`, a superclass that includes functions and features common to all the data types. Some of the classes are for JDBC 2.0-compliant data types. In addition to data type classes, the `oracle.sql` package supports classes and interfaces for use with objects and collections.

### oracle.jdbc

The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in the `java.sql` package. These extensions provide access to Oracle SQL-format data. They also provide access to other Oracle-specific features, including Oracle performance enhancements.

The key classes and interfaces of this package provide methods that support standard JDBC features and perform tasks such as:

- Returning Oracle statement objects

- Setting Oracle performance extensions for any statement

- Binding `oracle.sql.*` types into prepared and callable statements

- Retrieving data in `oracle.sql` format

- Getting meta information about the database and result sets

- Defining integer constants used to identify SQL types

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

# Using JDeveloper to Create JDBC Applications

The Java application tutorial in this guide uses Oracle JDeveloper 10*g* release 10.1.3 as the integrated development environment (IDE) for developing the Java application and creating Web pages for users to view and change the data.

Oracle JDeveloper is an IDE with support for modeling, developing, debugging, optimizing, and deploying Java applications and Web services.

JDeveloper provides features for you to write and test Java programs that access the database with SQL statements embedded in Java programs. For the database, JDeveloper provides functions and features to do the following:

- Create a connection to a database

- Browse database objects

- Create, edit, or delete database objects

- Create and edit PL/SQL functions, procedures, and packages

## JDeveloper User Interface

Oracle JDeveloper is an IDE that uses windows for various application development tools. You can display or hide any of the windows, and you can dock them or undock them to create a desktop suited to your method of working.

In addition to these tools, JDeveloper provides a range of navigators to help you organize and view the contents of your projects. Application and System navigators show you the files in your projects, and a Structure window shows you the structure of individual items.

You can arrange the windows as you choose, and can close and open them from the
View menu. Figure 1–1 shows the default layout of some of the available navigators,
palettes, and work areas in the JDeveloper user interface (GUI).

*Figure 1–1   JDeveloper User Interface*
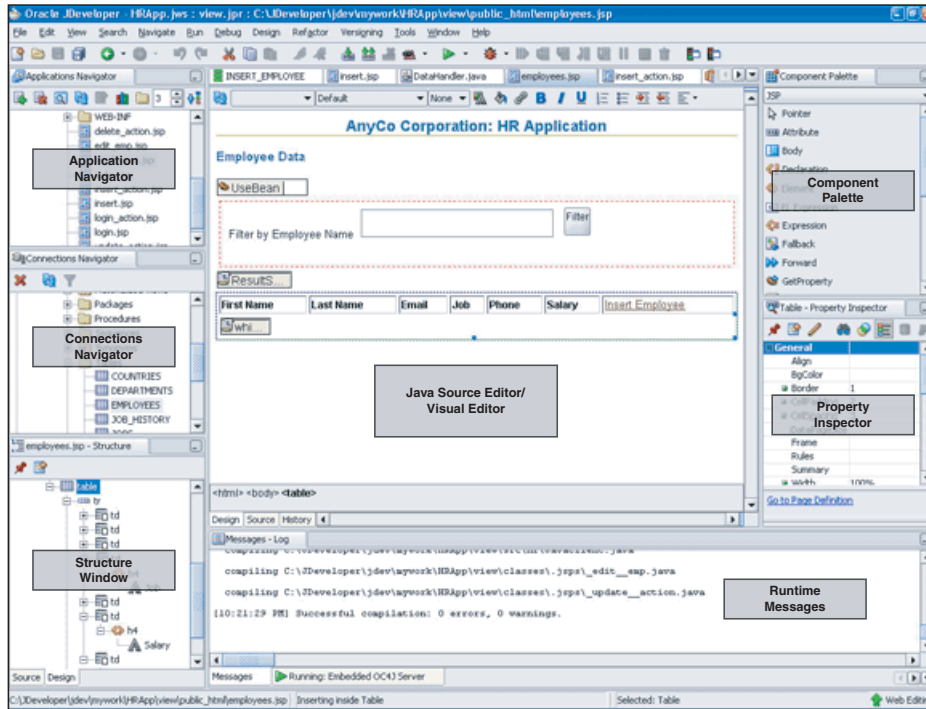


> **See Also:**   *Working with Windows in the IDE*, in the JDeveloper online
> Help

## JDeveloper Tools

For creating a Java application, JDeveloper provides the following tools to simplify the
process:

- **Structure window**, which provides a tree view of all of the elements in the
  application currently being edited be it Java, XML, or JSP/HTML.

- **Java Visual Editor**, which you can use to assemble the elements of a user interface
  quickly and easily.

- **JSP/HTML Visual Editor**, which you can use to visually edit HTML and JSP
  pages.

- **Java Source Editor**, which provides extensive features for helping in writing the
  Java code, such as distinctive highlighting for syntax and semantic errors,
  assistance for adding and sorting import statements, the Java Code Insight feature,
  and code templates.

> **Note:**   The Java Code Insight feature is a facility that provides
> context-specific, intelligent input when creating code in the Java
> Source Editor. In this guide, you will see many instances of how you
> can use Java Code Insight to insert code.

- **Component Palette**, from which you select the user interface components, such as buttons and text areas, that you want to display on your pages.

- **Property Inspector**, which gives a simple way of setting properties of items such as user interface components.

Figure 1–1 might help you get a better idea of where you can access these tools in the JDeveloper UI.

# Overview of Sample Java Application

This guide shows you how to create an application using Java, JDBC and Oracle ADF. In this application, you build in the following functions and features:

1. Allow users to log in and validate the user name and password.

2. Establish a connection to the database.

3. Query the database for data and retrieve the data using a JavaBean.

4. Display the data using JavaServer Pages (JSP) technology.

5. Allow users to insert, update, or delete records.

6. Access and modify information from a master-detail application.

7. Handle exceptions.

> **Note:** The application connects to the HR schema that ships with Oracle Database. Although the Oracle Database client installation comes with both the Thin and OCI drivers, the sample application will use only the JDBC Thin Driver.

### Overview of Application Web Pages (JSP Pages)

Figure 1–2 shows the relationships among the pages developed for this application.

**Figure 1–2   Web Pages in the Sample Application**
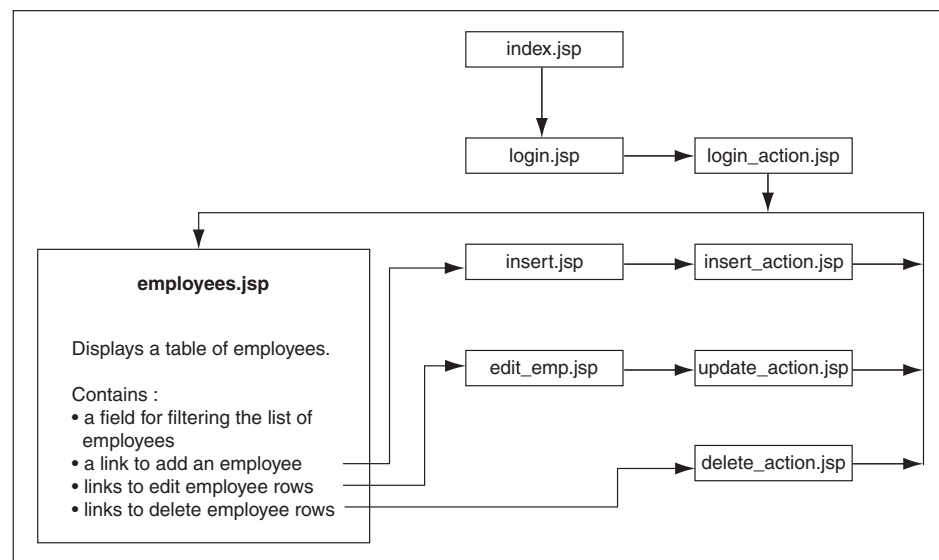


A brief description of the Web pages in the sample application follows:

- `index.jsp`

  This is the starting page of the application. It automatically forwards the user to the login page of the application, `login.jsp`.

- `login.jsp`

  This page allows users to log in to the application. The user name, password, and host information are validated and used to create the connection descriptor to log in to the database.

- `login_action.jsp`

  This is a nonviewable page that handles the authentication of the user-supplied login details from `login.jsp`. If authentication is successful, the page forwards the user to `employees.jsp`. Otherwise, it redisplays the `login.jsp` page including a message.

- `employees.jsp`

  This is the main page of the application. It displays a list of all the employees in the HR schema for AnyCo Corporation and allows the user to filter the list of employees using any string. It also includes links to add, edit, and delete any user data. These actions, however, are handled by other JSP pages that are created specifically for each of these tasks.

- `insert.jsp`

  The link to insert employee data on the `employees.jsp` page redirects the user to this page. This includes a form that accepts all the details for a new employee record. The details entered on this form are processed by the `insert_action.jsp` page.

- `insert_action.jsp`

  This is a nonviewable page that handles the insertion of data for a new employee that is entered on the `insert.jsp` page.

- `edit.jsp`

  The link to edit employee data on the `employees.jsp` page redirects the user to this page. This form displays current data of a single employee in text fields, and the user can edit this information.

- `update_action.jsp`

  The submit action on the `edit.jsp` page directs the data to this nonviewable page, which inserts the edited data into the database.

- `delete_action.jsp`

  The link to delete an employee record on the `employees.jsp` page is handled by this nonviewable page, which deletes the employee data and forwards the user back to the `employees.jsp` page.

**Classes**

The sample application includes the following classes:

- `DataHandler.java`

  This class contains all the methods that are used to implement the important functions of the sample application. It includes methods that validate user credentials, connect to the database, retrieve employee data with and without filters, insert data, update data, handle exceptions, and so on.

- `Employees.java`

  This class is a JavaBean that holds a single employee record. It contains accessor methods to get and set the values of each of the record fields. It also contains accessor methods to retrieve and modify employee records.

- `JavaClient.java`

  This class is used only for testing the `DataHandler` class.

---

> **Note:** This application is developed throughout this guide in the form of a tutorial. It is recommended, therefore, that you read these chapters in sequence.

---

# Advanced Application Development Using Developer Frameworks

To develop enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces, you need to use developer frameworks to simplify your job.

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

The Oracle Application Development Framework (Oracle ADF) is such an end-to-end application framework that builds on Java EE standards and open-source technologies to simplify and accelerate implementing service-oriented applications.

To illustrate how application development can be made easy using a feature-rich environment that facilitates the creation of complex applications, this guide includes a master-detail application in Chapter 7.

# 2

# Getting Started with the Application

To develop a Java application that connects to Oracle Database, you need to ensure that certain components are installed as required. This chapter covers the following topics:

- What You Need to Install
- Verifying the Oracle Database Client Installation
- Installing Oracle JDeveloper

## What You Need to Install

To be able to develop the sample application, you need to install the following products and components:

- Oracle Database Server
- Oracle Database Client
- J2SE or JDK
- Integrated Development Environment
- Web Server

The following subsections describe these requirements in detail.

### Oracle Database **Server**

To develop the Java application, you need a working installation of Oracle Database Server with the HR schema, which comes with the database. If you choose to install the client, then you must install the Oracle Database Server before the Oracle Database Client installation. The installation creates an instance of Oracle Database and provides additional tools for managing this database. The server installation is platform-specific. For more information, refer to the *Oracle Database Installation Guide*.

#### Modifying the HR Schema for the JDBC Application

The HR user account, which owns the sample HR schema used for the Java application in this guide, is initially locked. You must log in as a user with administrative privileges (SYS) and unlock the account before you can log in as HR.

If the database is locally installed, use the command prompt or console window to unlock the account as follows:

1. Log in to SQL*Plus as a user with DBA privileges, for example:

   ```
   > SQLPLUS SYS/ AS SYSDBA
   ```

```
Enter password: password
```

2. Run the following command:

```
> PASSWORD HR
Changing password for HR
New password: password
Retype new password: password
```

3. Test the connection as follows:

```
> CONNECT HR
Enter password: password
```

You should see a message indicating that you have connected to the database.

> **Note:** For information on creating and using secure passwords with Oracle Database, refer to the *Oracle Database Security Guide*.

In addition, some of the constraints and triggers present in the HR schema are not in line with the scope of the Java application created in this guide. You must remove these constraints and triggers as follows using the following SQL statements:

```
DROP TRIGGER HR.UPDATE_JOB_HISTORY;
DROP TRIGGER HR.ADD_JOB_HISTORY;
DROP TRIGGER HR.SECURE_EMPLOYEES;
ALTER TABLE EMPLOYEES  DROP CONSTRAINT JHIST_EMP_FK;
DELETE FROM JOB_HISTORY;
```

## Oracle Database **Client**

Oracle Database Client installation is optional, but recommended. Installing Oracle Database Client on any computer allows easy access from that system to the Oracle Database. The installation also includes the following development tools:

- Oracle JDBC drivers

- Oracle Open Database Connectivity (ODBC) driver

- Oracle Provider for OLE DB

- Oracle Data Provider for .NET (ODP.NET)

- Oracle Services for Microsoft Transaction Server

The client installation is platform-specific. Refer to the *Oracle Database Installation Guide* for more information about installing the client.

## J2SE or JDK

To create and compile Java applications, you need the full Java 2 Platform, Standard Edition, Software Development Kit (J2SE SDK), formerly known as the Java Development Kit (JDK). To create and compile applications that access databases, you must have the full JDBC API that comes with J2SE. This download also includes the Java Runtime Environment (JRE).

> **Note:**
>
> - Oracle Database does not support JDK 1.2, JDK 1.3, JDK 1.4, and all `classes12*.*` files. You need to use the `ojdbc5.jar` and the `ojbc6.jar` files with JDK 5.*n* and JDK 6.*n*, respectively.
>
> - The `oracle.jdbc.driver.*` classes, the `ojdbc4.jar` file, and the `OracleConnectionCacheImpl` class are no longer supported or available.
>
> - JDK versioning conventions have changed from JDK version 1.*n* to JDK *n*. Refer to the Sun Java site at the following location for more information:
>
>   `http://java.sun.com/j2se/1.5.0/docs/relnotes/version-5.0.html`

> **See Also:**  - `http://java.sun.com/javase/index.jsp` for information on installing Java
>
> - `http://java.sun.com/javase/technologies/database.jsp` for information on the JDBC API

## Integrated Development Environment

For ease in developing the application, you can choose to develop your application in an integrated development environment (IDE). This guide uses Oracle JDeveloper to create the files for this application. For more information on installing JDeveloper, refer to Installing Oracle JDeveloper.

## Web Server

The sample application developed in this guide uses JavaServer Pages (JSP) technology to display information and accept input from users. To deploy these pages, you need a Web server with a servlet and JSP container, such as the Apache Tomcat application server.

This guide uses the embedded server in JDeveloper for deploying the JSP pages. It is called the  Oracle Application Server Containers for J2EE server, or the OC4J server for short. If you choose not to install Oracle JDeveloper, then any Web server that allows you to deploy JSP pages should suffice.

JDeveloper supports direct deployment to the following production application servers:

- Oracle Application Server

- BEA WebLogic

- Apache Tomcat

- IBM WebSphere

- JBoss

For more information about these servers, please refer to vendor-specific documentation.

# Verifying the Oracle Database Client Installation

Oracle Database client installation is platform-specific. You need to verify that the client installation was successful before you proceed to create the sample application. This section describes the steps for verifying an Oracle Database client installation.

Verifying a client installation involves the following tasks:

- Checking Installed Directories and Files
- Checking the Environment Variables
- Determining the JDBC Driver Version

## Checking Installed Directories and Files

Installing Oracle Java products creates the following directories:

- *ORACLE_HOME*/jdbc
- *ORACLE_HOME*/jlib

Check if the directories described in Table 2–1 have been created and populated in the *ORACLE_HOME* directory.

**Table 2–1    Directories and Files in the ORACLE_HOME Directory**

| Directory | Description |
|-----------|-------------|
| /jdbc/lib | The lib directory contains the ojdbc5.jar and ojdbc6.jar required Java classes. These contain the JDBC driver classes for use with JDK 5 and JDK 6. |
| /jdbc/Readme.txt | This file contains late-breaking and release-specific information about the drivers, which may not have been included in other documentation on the product. |
| /jlib | This directory contains the orai18n.jar file. This file contains classes for globalization and multibyte character sets support. |

> **Note:** These files can also be obtained from the Sun Microsystems Web site. However, it is recommended to use the files supplied by Oracle, which have been tested with the Oracle drivers.

## Checking the Environment Variables

This section describes the environment variables that must be set for the JDBC Thin Driver. You must set the classpath for your installed JDBC Thin Driver. For JDK 5, you must set the following values for the CLASSPATH variable:

```
ORACLE_HOME/jdbc/lib/ojdbc5.jar
ORACLE_HOME/jlib/orai18n.jar
```

Ensure that there is only one JDBC class file, such as ojdbc6.jar, and one globalization classes file, orai18n.jar, in the CLASSPATH variable.

## Determining the JDBC Driver Version

Starting from Oracle Database 11*g* Release 1, you can get details about the JDBC support in the database as follows:

```
> java -jar ojdbc6.jar
```

```
Oracle 11.1.0.0. JDBC 4.0 compiled with JDK6
```

In addition, you can determine the version of the JDBC driver that you installed by calling the `getDriverVersion` method of the `OracleDatabaseMetaData` class.

> **Note:** The JDBC Thin Driver requires a TCP/IP listener to be running on the computer where the database is installed.

Example 2–1 illustrates how to determine the driver version:

***Example 2–1   Determining the JDBC Driver Version***
```java
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

class JDBCVersion
{
  public static void main (String args[]) throws SQLException
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:hr/hr@localhost:1521/XE");
    Connection conn = ods.getConnection();

    // Create Oracle DatabaseMetaData object
    DatabaseMetaData meta = conn.getMetaData();

    // gets driver info:
    System.out.println("JDBC driver version is " + meta.getDriverVersion());
  }
}
```

# Installing Oracle JDeveloper

In this guide, the integrated development environment (IDE) that is used to create the sample Java application using JDBC is Oracle JDeveloper release 10.1.3. This release of JDeveloper is supported on the Microsoft Windows XP, Windows 2000, Windows NT, Linux, Solaris, Mac OS X, and HP-UX operating systems. Installation of JDeveloper is described in detail in *Oracle JDeveloper 10g Release 3 (10.1.3) Installation Guide for the Studio Edition*, which is available online on the Oracle Technology Network at

http://www.oracle.com/technology/documentation/jdev/1013install/install.html

This guide gives a detailed description of the JDeveloper system requirements, and all the details about installing JDeveloper on the supported platforms. You should also read *JDeveloper 10g Release Notes*, which is available online on the Oracle Technology Network at

http://www.oracle.com/technology/products/jdev/htdocs/10.1.3.0.3/readme.html

## JDeveloper Studio Edition: Base Installation and Full Installation

JDeveloper 10.1.3 is available in three editions. Further, each of these editions is available as the base installation and the full installation. The Studio Edition includes

Oracle ADF, which is required for developing the master-detail application created in this guide.

You can install either the base installation or the full installation of the JDeveloper Studio Edition. In addition to JDeveloper, the full installation includes the required version of Java, the specialized Oracle Java Virtual Machine for JDeveloper (OJVM), and the online documentation, so the download file size is larger. For quicker downloading, you can install the JDeveloper base installation.

## Steps to Install JDeveloper

JDeveloper does not require an installer, because it is provided as a ZIP file. In outline, the installation process is as follows:

1. If you are installing the Base version, you need to have J2SE version 1.5.0_05 on your machine. If you are installing the Full version, this J2SE is included.

2. Download JDeveloper version 10.1.3 Studio Edition from the Oracle Technology Network at

   http://www.oracle.com/technology/software/products/jdev/htdocs/soft1013.html

   Download the base installation (`jdevstudiobase1013.zip`), or the full installation (`jdevstudio1013.zip`)

3. Unzip the downloaded file into a target installation directory.

   > **Note:** Do not install JDeveloper in any existing `ORACLE_HOME`. You will not be able to uninstall it using Oracle Universal Uninstaller.

   If you install `jdevstudio1013.zip` on a UNIX or Linux system, you have to modify `jdev.conf` to specify the SDK. Set the variable `SetJavaHome` in the file `<jdev_install>`/jdev/bin/jdev.conf to the location of your Java installation.

   For example, in a UNIX environment, if the location of your Sun J2SE SDK is in a directory called `/usr/local/java`, your entry in `jdev.conf` would be as follows:

   ```
   SetJavaHome /usr/local/java
   ```

   Other tasks that you must perform include setting the permissions for all JDeveloper files to read, and giving all users write and execute permissions to files in a range of JDeveloper directories.

4. If you are using the base installation, there are some additional setup tasks, such as setting the location of your Java installation in the JDeveloper configuration file, optionally installing OJVM, and downloading the online documentation so that it is locally available.

   > **See Also:**
   > - http://www.oracle.com/technology/documentation/jdev/1013install/install.html for the JDeveloper Installation Guide

## Starting JDeveloper

To start JDeveloper on Windows, run the `jdev_install`\jdev\bin\jdevw.exe file, where `jdev_install` is the path to the location where you extract the JDeveloper files. To use a console window for displaying internal diagnostic information, run the `jdev.exe` file in the same directory instead of `jdevw.exe`.

To start JDeveloper on other platforms, run the *jdev_install*/jdev/bin/jdev file.

# 3

# Connecting to Oracle Database

This chapter is the first in a series of five chapters, each of which describes how to create parts of a Java application that accesses Oracle Database and displays, modifies, deletes, and updates data on it. To be able to access the database from a Java application, you must connect to the database using a `java.sql.Connection` object.

This chapter includes the following sections:

- Connecting to Oracle Database from JDeveloper
- Setting Up Applications and Projects in JDeveloper
- Connecting to Oracle Database from a Java Application

## Connecting to Oracle Database from JDeveloper

You can set up and manage database connections in JDeveloper to enable your application to communicate with external data sources, including Oracle Database and offline database objects. This is done using the Connection Navigator. The same navigator is also used to manage other connections your application needs, such as connections to application servers. The following subsections describe how you can use the Connection Navigator to view the database and its objects and to create a connection to the database:

- JDeveloper Connection Navigator
- Creating a Database Connection
- Browsing the Data Using the Connection Navigator

### JDeveloper Connection Navigator

The Connection Navigator displays all currently defined connections. To view the Connection Navigator, select the **Connections** tab in the navigator panel on the top left-hand side of the JDeveloper display, if it is displayed, or use the View menu. For an illustration of the default layout of the JDeveloper IDE, see Figure 1–1.

You can use the Connection Navigator to browse through the connections it displays. In particular, for a database schema, you can also view database objects, tables, views, and their contents.

Database connections are shown under the Database node. To view the objects in the database, expand the connection. Expanding a schema displays nodes for the object types in that schema. Expanding the node for an object type displays its individual objects. When you expand a table node, you can view the structure of the table and the data within the table.

## Creating a Database Connection

You can connect to any database for which you have connection details. When you create a database connection, you must specify a user name and a password. By default, the connection allows you to browse only the schema of the user that you specify in the connection.

To create a connection, follow these steps:

1. Start JDeveloper.

2. From the **View** menu, select **Connection Navigator**. The Connection Navigator is displayed, showing you a list of the available connections.

3. Right-click **Database**, and from the shortcut menu, select **New Database Connection**. The Create Database Connection wizard is displayed. Click **Next** on the Welcome screen. The Type screen of the wizard is displayed.

4. On the Type screen, do not change the default values for the connection name and type, `DBConnection1` and `Oracle (JDBC)`. Click **Next**. The Authentication screen of the wizard is displayed.

5. On the Authentication screen, enter `HR` in both the **Username** and **Password** fields. Do not enter a value for Role, and select **Deploy Password**. Click **Next**. The Connection screen of the wizard is displayed.

6. In the Connection screen, you must provide information about the computer where your database is located. Your database administrator should provide you with this information.

   Enter the following information:

   - **Driver**: `thin`

   - **Host Name**: *Host name of the computer where Oracle Database is installed*

     If database is on the same computer, then for the **Host Name** parameter, enter `localhost`.

   - **JDBC Port**: `1521`

   - **SID**: `ORCL`

     Figure 3–1 shows the Connection screen where you enter these details.

*Figure 3–1   Specifying Connection Details*
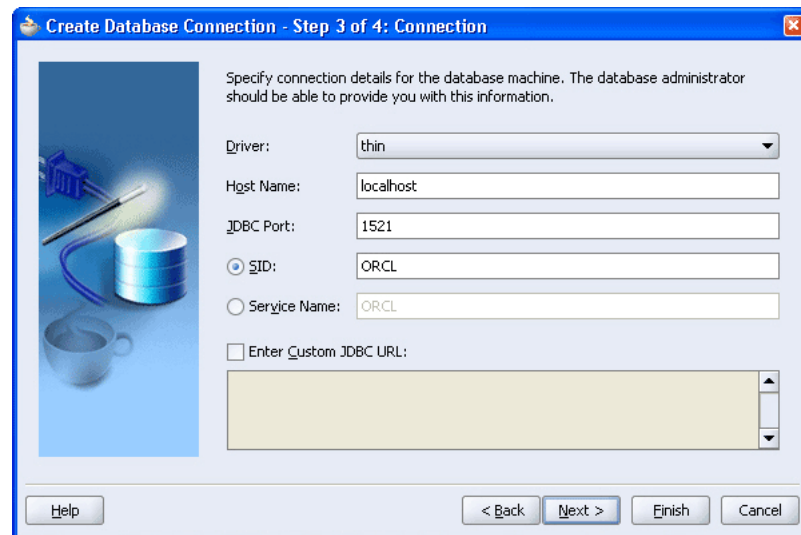


7.  On the Test screen, you can test whether or not you can successfully connect to the database. Click **Test Connection**. If the connection is successful, the word Success! is displayed in the Status field.

8.  Click **Finish** to create the connection and close the wizard.

**Disconnecting and Reconnecting from Oracle Database in JDeveloper**

To disconnect from the database in JDeveloper, in the Connection Navigator, right-click the connection name and select **Disconnect**. The display in the Connection Navigator now shows only the name of the connection, without the plus (+) symbol for expanding the node. To reconnect to the database, right-click the connection name and select **Connect**.

## Browsing the Data Using the Connection Navigator

After you have successfully established a connection to the database, you can browse its contents through the Connection Navigator. The Connection Navigator displays a navigable, hierarchical tree structure for the database, its objects, their instances, and the contents of each. To view the contents at each level of the hierarchy of the database connection that you created, do the following:

1.  The Database node in the Connection Navigator now shows a node with the name of your connection. Click the plus symbol (**+**) to the left of the connection name to expand the navigation tree. The name of the schema to which you connected, in this case HR, is displayed.

2.  To view the list of all the objects in the HR schema, expand the **HR** navigation tree. To display a list of the instances of an object type, for example Tables, expand the **Table** navigation tree.

*Figure 3–2   Accessing Database Objects in the Connection Navigator*
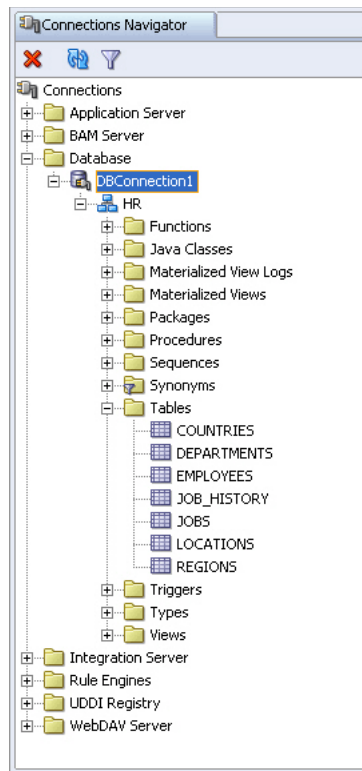


3. The Structure window below the navigator shows the detailed structure of any object selected in the navigator. Select a table in the navigator (for example **Employees**) to see the columns of that table in the Structure window.

*Figure 3–3   Viewing the Table Structure and Data*
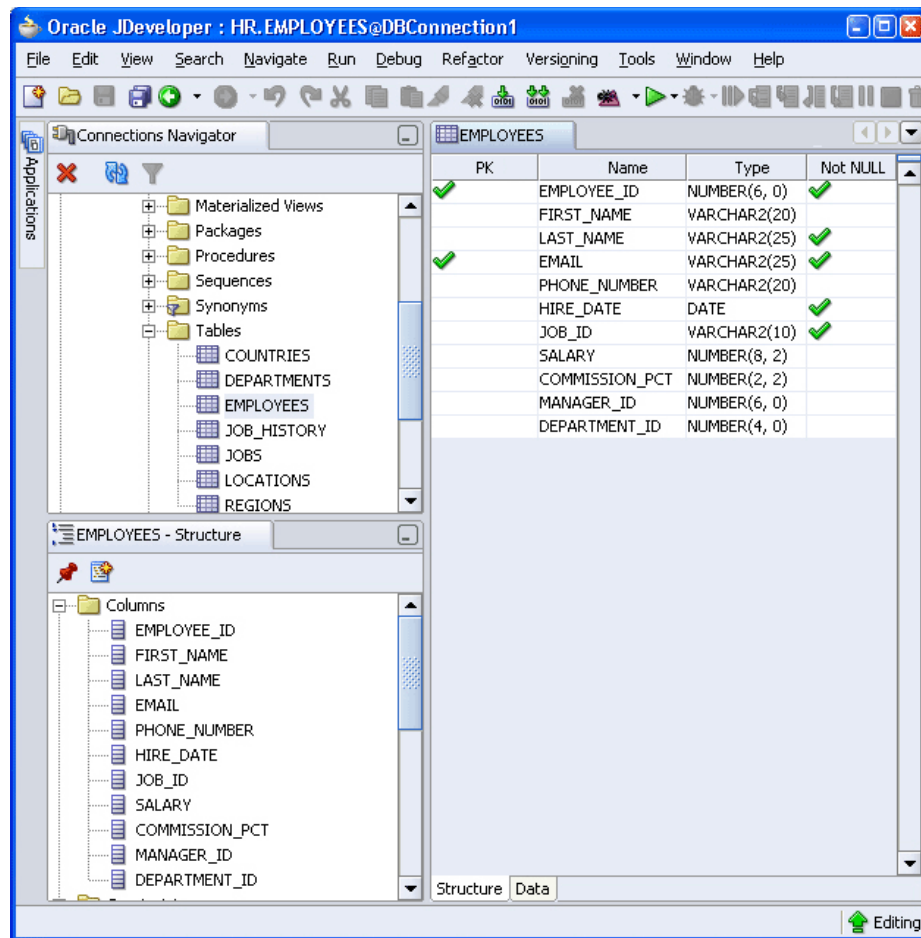


4.  If you double-click a table in the navigator, the structure of that table is displayed in the main editing area of the window. It includes details about all the columns, such as Name, Type, and Size, so you can browse the table definition.

    To view the data from a table, select the **Data** tab below the table structure. You can now view and browse through the table data.

5.  You can also edit the objects in the Connection Navigator. To edit a table, right-click the table and select **Edit** from the shortcut menu. A dialog box allows you to make changes to the selected table.

## Setting Up Applications and Projects in JDeveloper

In JDeveloper, you create your work in an application, within which you can organize your work into a number of projects. JDeveloper provides a number of application templates, to help you to create the project structure for standard types of application relatively quickly and easily. At the time you create your application in JDeveloper, you can choose the application template that matches the type of application you will be building.

The application template you select determines the initial project structure (the named project folders within the application) and the application technologies that will be included. You can then add any extra libraries or technologies you need for your particular application, and create additional projects if you need them.

## Using the JDeveloper Application Navigator

The Application Navigator displays all your applications and projects. When you first start JDeveloper, the Application Navigator is displayed by default on the left side of the JDeveloper IDE.

To view the Application Navigator when it is not displayed, you can click the **Applications** tab in the navigator panel on the top left-hand side of the JDeveloper display, or select **Application Navigator** from the View menu.

The Application Navigator shows a logical grouping of the items in your projects. To see the structure of an individual item, you can select it and the structure is displayed in the Structure window.

From the Application Navigator, you can display items in an appropriate default editor. For example, if you double-click a Java file, the file opens in the Java Source Editor, and if you double-click a JavaServer Pages (JSP) file, it opens in the JSP/HTML Visual Editor.

## Creating an Application and a Project

To get started with JDeveloper, you must create an application and at least one project in which to store your work, as follows:

1.  In the Application Navigator, right-click **Applications**, and select **New Application** from the shortcut menu. The Create Application Workspace dialog box is displayed, which is shown in Figure 3–4.

2.  Enter `HRApp` in the Application Name field, and in the Application Template list, select **No Template [All Technologies]**. Click **OK**. The Create Project dialog box is displayed.

*Figure 3–4   Creating an Application*
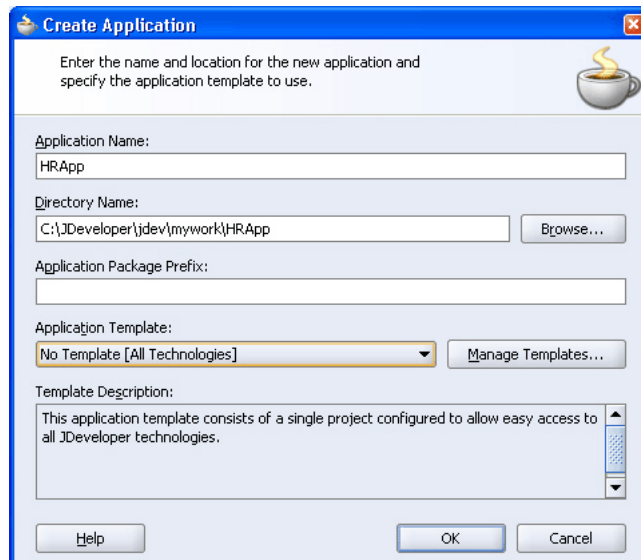


3.  On the Create Project dialog box, enter `view` as the name of the project.

4.  The new `HRApp` application is displayed in the Application Navigator.

5.  Save your application. To do this, from the **File** menu, select **Save All**.

## Viewing the Javadoc and Source Code Available in the Project Scope

You can view the Javadoc or the code for any of the classes available in the project technology scope within JDeveloper. In addition, you can view the details of all the methods available for those classes.

For example, to see the code or Javadoc for the `Connection` class, do the following:

1. With your project selected in the Application Navigator, from the **Navigate** menu select **Go to Java Class**. You can also do this for a specific file in your project.

2. In the Go to Java Class dialog box, select **Source** or **Javadoc**.

3. Enter the name of the class you want to view in the **Name** field, or click **Browse** to find the class. For the `Connection` class, start to enter `Connection`, and from the displayed list select **Connection (java.sql)**.

*Figure 3–5   Selecting the Class to View the Javadoc in JDeveloper*
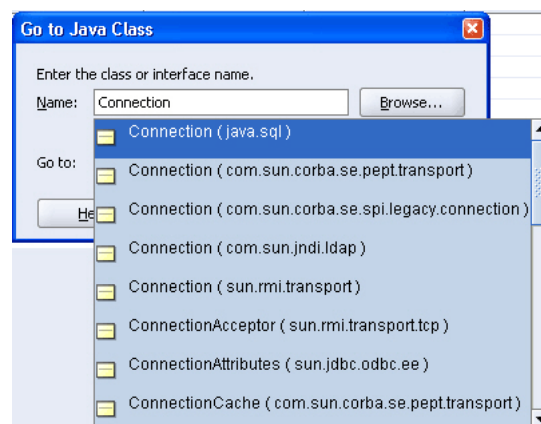


4. Click **OK**.

**Figure 3–6   Javadoc Display in JDeveloper**
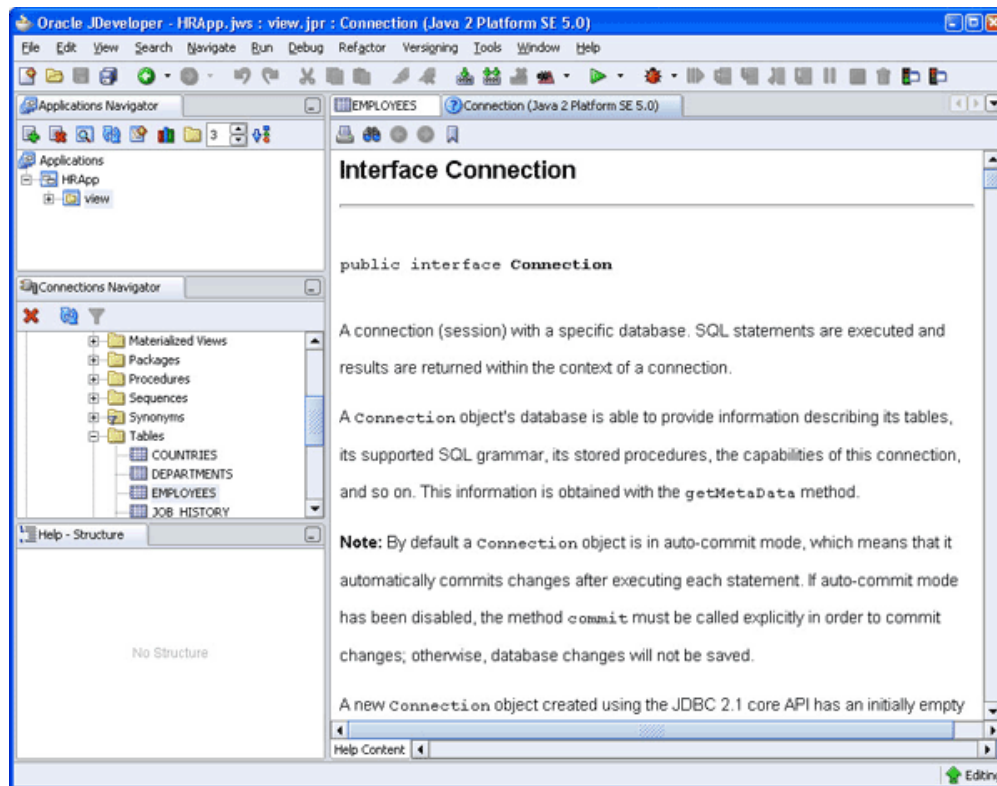


# Connecting to Oracle Database from a Java Application

So far, you have seen how to connect to the database from JDeveloper. To initiate a connection from the Java application, you use the Connection object from the JDBC application programming interface (API).

This section describes connecting to the database from the Java application in the following subsections:

- Overview of Connecting to Oracle Database

- Specifying Database URLs

- Creating a Java Class in JDeveloper

- Java Libraries

- Adding JDBC and JSP Libraries

- Importing JDBC Packages

- Declaring Connection-Related Variables

- Creating the Connection Method

## Overview of Connecting to Oracle Database

In Java, you use an instance of the DataSource object to get a connection to the database. The DataSource interface provides a complete replacement for the previous JDBC DriverManager class. Oracle implements the javax.sql.DataSource interface with the OracleDataSource class in the oracle.jdbc.pool package. The overloaded getConnection method returns a physical connection to the database.

> **Note:** The use of the `DriverManager` class to establish a connection to a database is deprecated.

You can either set properties using appropriate set*xxx* methods for the `DataSource` object or use the `getConnection` method that accepts these properties as input parameters.

Important `DataSource` Properties are listed in Table 3–1.

*Table 3–1    Standard Data Source Properties*

| Name | Type | Description |
| --- | --- | --- |
| databaseName | String | Name of the particular database on the server. Also known as the service name (or SID) in Oracle terminology. For Oracle Database, this is `ORCL` by default. |
| dataSourceName | String | Name of the underlying data source class. |
| description | String | Description of the data source. |
| networkProtocol | String | Network protocol for communicating with the server. For Oracle, this applies only to the JDBC Oracle Call Interface (OCI) drivers and defaults to `tcp`. |
| password | String | Password for the connecting user. |
| portNumber | int | Number of the port where the server listens for requests |
| serverName | String | Name of the database server |
| user | String | User name to be used for login |
| driverType | String | Specifies the Oracle JDBC driver type. It can be either `oci` or `thin`.<br>This is an Oracle-specific property. |
| url | String | Specifies the URL of the database connect string. You can use this property in place of the standard `portNumber`, `networkProtocol`, `serverName`, and `databaseName` properties.<br>This is an Oracle-specific property. |

If you choose to set the `url` property of the `DataSource` object with all necessary parameters, then you can connect to the database without setting any other properties or specifying any additional parameters with the `getDBConnection` method. For more information on setting the database URL, refer to the Specifying Database URLs section.

> **Note:** The parameters specified through the `getConnection` method override all property and `url` parameter settings previously specified in the application.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

## Specifying Database URLs

Database URLs are strings that you specify for the value of the `url` property of the `DataSource` object. The complete URL syntax is the following:

```
jdbc:oracle:driver_type:[username/password]@database_specifier
```

The first part of the URL specifies which JDBC driver is to be used. The supported *driver_type* values for client-side applications are `thin` and `oci`. The brackets indicate

that the user name and password pair is optional. The *database_specifier* value identifies the database to which the application is connected.

The following is the syntax for thin-style service names that are supported by the Thin driver:

```
jdbc:oracle:driver_type:[username/password]@//host_name:port_number:SID
```

For the sample application created in this guide, if you include the user name and password, and if the database is hosted locally, then the database connection URL is as shown in Example 3–1.

**Example 3–1   Specifying the url Property for the DataSource Object**

```
jdbc:oracle:thin:hr/hr@localhost:1521:UORCL
```

### Using the Default Service Feature of the Oracle Database Client

Oracle Database comes with a new connection feature. If you install the Oracle Database client, then you need not specify all the details in the database specifier part of the connection URL. Under certain conditions, the Oracle Database connection adapter requires only the host name of the computer where the database is installed.

Because of this feature introduced in Oracle Database, some parts of the JDBC connection URL syntax become optional:

```
jdbc:oracle:driver_type:[username/password]@[//]host_name[:port][:ORCL]
```

In this URL:

- // is optional.

- *:port* is optional.

  Specify a port only if the default Oracle Net listener port (1521) is not used.

- :ORCL (or the service name) is optional.

  The connection adapter for the Oracle Database Client connects to the default service on the host. On the host, this is set to ORCL in the listener.ora file.

Example 3–2 shows a basic configuration of the listener.ora file, where the default service is defined.

**Example 3–2   Default Service Configuration in listener.ora**

```
MYLISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcp)(HOST=test555)(PORT=1521))
)
DEFAULT_SERVICE_MYLISTENER=dbjf.regress.rdbms.dev.testserver.com

SID_LIST_MYLISTENER = (SID_LIST=
(SID_DESC=(SID_NAME=dbjf)(GLOBAL_
DBNAME=dbjf.regress.rdbms.dev.testserver.com)(ORACLE_HOME=/test/oracle))
)
```

After making changes to the listener.ora file, you must restart the listener with the following command:

```
> lsnrctl start mylistener
```

The following URLs should work with this configuration:

```
jdbc:oracle:thin:@//test555.testserver.com
jdbc:oracle:thin:@//test555.testserver.com:1521
```

```
jdbc:oracle:thin:@test555.testserver.com
jdbc:oracle:thin:@test555.testserver.com:1521
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com
)(PORT=1521)))
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com
)))
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com
)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=)))
```

> **Note:** Default service is a new feature in Oracle Database 11*g* Release
> 1. If you use any other version of the Oracle Database Client to
> connect to the database, then you must specify the SID and port
> number.

## Creating a Java Class in JDeveloper
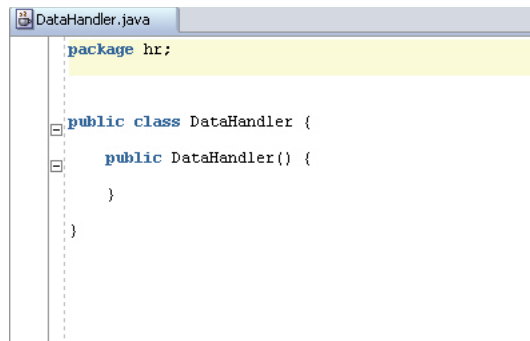
The first step in building a Java application is to create a Java class. The following
instructions describe how you create a class called `DataHandler`, which will contain the
methods for querying the database and modifying the data in it.

1.  In the Application Navigator, right-click the **View** project, and from the shortcut
    menu, select **New**.

2.  In the **New Gallery**, select the **General** category. In the **Items** list, select **Java
    Class**, and click **OK**. The Create Java Class dialog box is displayed.

3.  In the Create Java Class dialog box, enter `DataHandler` as the class **Name**, and `hr`
    as the **Package**. Do not change the default values of the Optional Attributes, and
    click **OK**. The Create Java Class dialog box with the appropriate values specified is
    shown in Figure 3–7.

*Figure 3–7   Creating a Java Class*
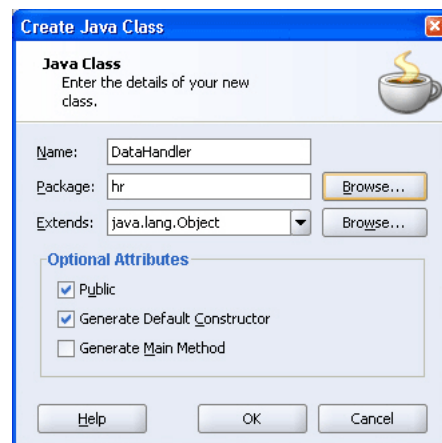


4.  The skeleton `DataHandler` class is created and is displayed in the Java Source
    Editor. The package declaration, the class declaration, and the default constructor
    are created by default. Figure 3–8 shows the class displayed in the Java Source
    Editor, ready for you to add your Java code:

*Figure 3–8   Java Source Editor*



## Java Libraries

Oracle JDeveloper comes with standard libraries to help Java application programming. These libraries include API support for Application Development Framework (ADF), Oracle libraries for JDBC, JSP, and so on.

To use JDBC in your project, you import the Oracle JDBC library into the project. Similarly, to use JSP technology, you import the JSP Runtime library.

### Overview of the Oracle JDBC Library

Important packages of the Oracle JDBC library include the following:

- `oracle.jdbc`: The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in the `java.sql` package. These extensions provide access to Oracle SQL-format data and other Oracle-specific features, including Oracle performance enhancements.

- `oracle.sql`: The `oracle.sql` package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes.

- `oracle.jdbc.pool`: This package includes the `OracleDataSource` class that is used to get a connection to the database. The overloaded `getConnection` method returns a physical connection to the database.

### Overview of the JSP Runtime Library

This library includes the classes and tag libraries required to interpret and run JSP files on the OC4J server that comes with JDeveloper.

## Adding JDBC and JSP Libraries

To include libraries in your project, perform the following steps:

1. Double-click the **View** project in the Application Navigator to display the Project Properties dialog box.

2. Click **Libraries**, and then click **Add Library**. The Add Library dialog box is displayed with a list of the available libraries for the Java2 Platform, Standard Edition (J2SE) version is displayed.

3. In the Add Library dialog box, scroll through the list of libraries in the Extension folder. Select the **JSP Runtime** library and click OK to add it to the list of selected libraries for your project. Similarly, add the Oracle JDBC library. Figure 3–9 shows the Oracle JDBC library added to the `view` project.

*Figure 3–9   Importing Libraries*
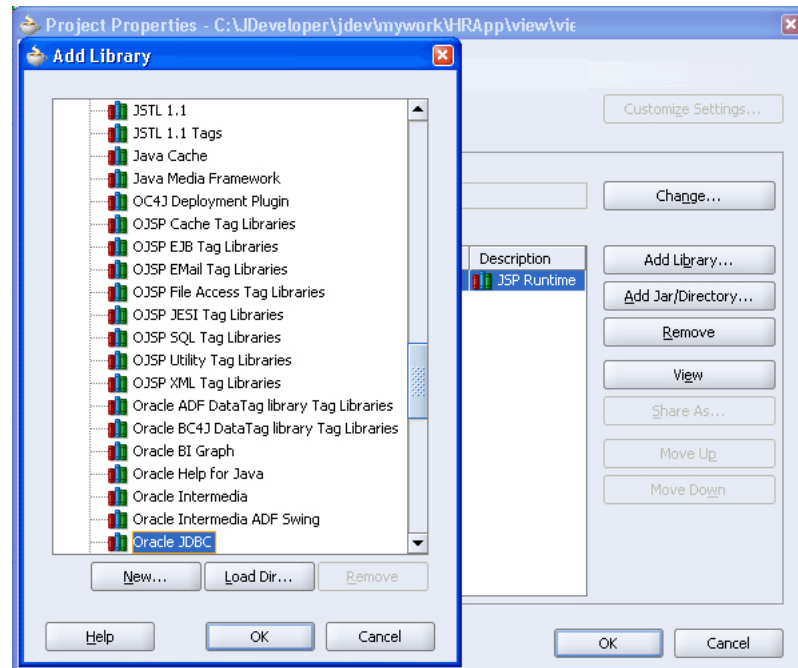


4.   Click **OK**.

## Importing JDBC Packages

To use JDBC in the Java application, import the following JDBC packages:

1.   If the `DataHandler.java` class is not already open in the Java Source Editor, in the Application Navigator, expand the **View** project, **Application Sources**, and your package (**hr**) and double-click **DataHandler.java**.

2.   At the end of the generated package declaration, on a new line, enter the `import` statements shown in Example 3–3.

*Example 3–3   Importing Packages in a Java Application*

```
package hr;
import java.sql.Connection;
import oracle.jdbc.pool.OracleDataSource;
```

## Declaring Connection-Related Variables

Connection information is passed to the connection method by using the following connection variables: the connection URL, a user name, and the corresponding password.

Use the Java Source Editor of JDeveloper to edit the `DataHandler.java` class as follows:

1.   After the `DataHandler` constructor, on a new line, declare the three connection variables as follows:

```
String jdbcUrl = null;
String userid = null;
String password = null;
```

These variables will be used in the application to contain values supplied by the user at login to authenticate the user and to create a connection to the database. The `jdbcUrl` variable is used to hold the URL of the database that you will connect to. The `userid` and `password` variables are used to authenticate the user and identify the schema to be used for the session.

---

**Note:** The login variables have been set to null to secure the application. At this point in the guide, application login functionality is yet to be built into the application. Therefore, to test the application until login functionality is built in, you can set values in the login variables as follows:

Set the `jdbcUrl` variable to the connect string for your database.

```
String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
```

Set the variables `userid` and `password` to `hr` as follows:

```
String userid = "hr";
String password = "hr";
```

Make sure you reset these to `null` as soon as you finish testing.

For more information on security features and practices, refer to *Oracle Database Security Guide* and the vendor-specific documentation for your development environment.

---

2. On a new line, declare a connection instance as follows:

```
Connection conn;
```

Your Java class should now contain the code in Example 3–4.

***Example 3–4   Declaring Connection Variables and the Connection Object***

```
package hr;
import java.sql.Connection;
import oracle.jdbc.pool.OracleDataSource;

public class DataHandler {
    public DataHandler() {
    }
    String jdbcUrl = null;
    String userid = null;
    String password = null;
    Connection conn;
}
```

## Creating the Connection Method

To connect to the database, you must create a method as follows:

1. Add the following method declaration after the connection declaration:

```
public void getDBConnection() throws SQLException
```

The Java Code Insight feature displays a message reminding you to import the `SQLException` error handling package. Press the Alt+Enter keys to import it. The `import java.sql.SQLException` statement is added to the list of import packages.

2. At the end of the same line, add an open brace ({) and then press the Enter key. JDeveloper automatically creates the closing brace, and positions the cursor in a new empty line between the braces.

3. On a new line, declare an `OracleDataSource` instance as follows:

```
OracleDataSource ds;
```

4. Enter the following to create a new `OracleDataSource` object:

```
ds = new OracleDataSource();
```

5. Start to enter the following to set the URL for the `DataSource` object:

```
ds.setURL(jdbcUrl);
```

Java Code Insight prompts you by providing you with a list of available `OracleDataSource` methods. Scroll through the list to select the `setURL(String)` method, and press the Enter key to select it into your code. In the parentheses for this function, enter `jdbcUrl`.

Figure 3–10 shows how the Java Code Insight feature in JDeveloper helps you with inserting code.

**Figure 3–10   Java Code Insight**
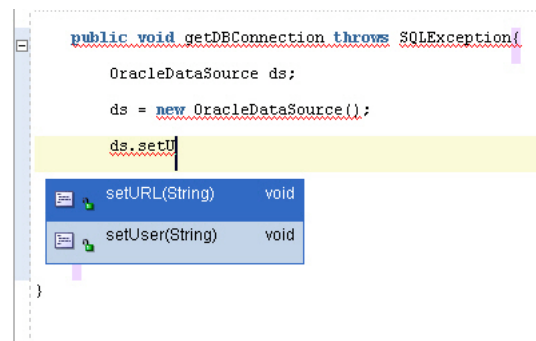


6. On the next line, enter the following:

```
conn = ds.getConnection(userid,password);
```

As usual, Java Code Insight will prompt you with a list of methods for `ds`. This time, select `getConnection(String,String)`. In the parentheses, enter `userid,password`. End the line with a semicolon (;).

Your code should look similar to the code in Example 3–5.

**Example 3–5   Adding a Method to Connect to the Database**

```
package hr;
import java.sql.Connection;
import java.sql.SQLException;

import oracle.jdbc.pool.OracleDataSource;

public class DataHandler {
    public DataHandler() {
    }
    String jdbcUrl = null;
    String userid = null;
```

```
                String password = null;
                Connection conn;
                public void getDBConnection() throws SQLException{
                    OracleDataSource ds;
                    ds = new OracleDataSource();
                    ds.setURL(jdbcUrl);
                    conn=ds.getConnection(userid,password);

                }
            }
```

7. Compile your class to ensure that there are no syntax errors. To do this, right-click in the Java Source Editor, and select **Make** from the shortcut menu. A `Successful compilation` message is displayed in the Log window below the Java Source Editor window.

# 4

# Querying for and Displaying Data

This chapter adds functions and code to the `DataHandler.java` file for querying the database. This chapter has the following sections:

- Overview of Querying for Data in Oracle Database
- Querying Data from a Java Application
- Creating JSP Pages
- Adding Dynamic Content to the JSP Page: Database Query Results
- Filtering a Query Result Set
- Adding Login Functionality to the Application
- Testing the JSP Page

## Overview of Querying for Data in Oracle Database

In outline, to query Oracle Database from a Java class to retrieve data, you must do the following:

1. Create a connection by using the `OracleDataSource.getConnection` method. This is covered in Chapter 3, "Connecting to Oracle Database".

2. Define your SQL statements with the methods available for the connection object. The `createStatement` method is used to define a SQL query statement.

3. Using the methods available for the statement, run your queries. You use the `executeQuery` method to run queries on the database and produce a set of rows that match the query conditions. These results are contained in a `ResultSet` object.

4. You use a `ResultSet` object to display the data in the application pages.

The following sections describe important Java Database Connectivity (JDBC) concepts related to querying the database from a Java application:

- SQL Statements
- Query Methods for the Statement Object
- Result Sets

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

## SQL Statements

Once you connect to the database and, in the process, create a `Connection` object, the next step is to create a `Statement` object. The `createStatement` method of the JDBC

Connection object returns an object of the JDBC Statement type. Example 4–1 shows how to create a Statement object.

### Example 4–1    Creating a Statement Object

```
Statement stmt = conn.createStatement();
```

The Statement object is used to run static SQL queries that can be coded into the application.

In addition, for scenarios where many similar queries with differing update values must be run on the database, you use the OraclePreparedStatement object, which extends the Statement object. To access stored procedures on Oracle Database, you use the OracleCallableStatement object.

> **See Also:**
>
> - Using OraclePreparedStatement
> - Using OracleCallableStatement
> - *Oracle Database JDBC Developer's Guide and Reference*

## Query Methods for the Statement Object

To run a query embedded in a Statement object, you use variants of the execute method. Important variants of this method are listed in Table 4–1.

*Table 4–1    Key Query Execution Methods for java.sql.Statement*

| Method Name | Return Type | Description |
| --- | --- | --- |
| execute(String sql) | Boolean | Runs the given SQL statement, which returns a Boolean response: true if the query runs successfully and false if it does not. |
| addBatch() | void | Adds a set of parameters to a PreparedStatement object batch of commands. |
| executeBatch() | int[] | Submits a batch of commands to the database for running, and returns an array of update counts if all commands run successfully. |
| executeQuery(String sql) | ResultSet | Runs the given SQL statement, which returns a single ResultSet object. |
| executeUpdate(String sql) | int | Runs the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or a SQL statement that returns nothing, such as a SQL DDL statement. |

> **See Also:**
> - http://java.sun.com/j2se/1.5.0/docs/api/index.html

## Result Sets

A ResultSet object contains a table of data representing a database result set, which is generated by executing a statement that queries the database.

A cursor points to the current row of data in a ResultSet object. Initially, it is positioned before the first row. You use the next method of the ResultSet object to

move the cursor to the next row in the result set. It returns `false` when there are no more rows in the `ResultSet` object. Typically, the contents of a `ResultSet` object are read by using the `next` method within a loop until it returns `false`.

The `ResultSet` interface provides accessor methods (`getBoolean`, `getLong`, `getInt`, and so on) for retrieving column values from the current row. Values can be retrieved by using either the index number of the column or the name of the column.

By default, only one `ResultSet` object per `Statement` object can be open at the same time. Therefore, to read data from multiple `ResultSet` objects, you must use multiple `Statement` objects. A `ResultSet` object is automatically closed when the `Statement` object that generated it is closed, rerun, or used to retrieve the next result from a sequence of multiple results.

> **See Also:**
>
> - http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/mapping.html for more information on mapping SQL types and Java types
>
> - *Oracle Database JDBC Developer's Guide and Reference* for more information on result sets and their features

### Features of ResultSet Objects

**Scrollability** refers to the ability to move backward as well as forward through a result set. You can also move to any particular position in the result set, through either **relative positioning** or **absolute positioning**. Relative positioning lets you move a specified number of rows forward or backward from the current row. Absolute positioning lets you move to a specified row number, counting from either the beginning or the end of the result set.

When creating a scrollable or positionable result set, you must also specify **sensitivity**. This refers to the ability of a result set to detect and reveal changes made to the underlying database from outside the result set. A sensitive result set can see changes made to the database while the result set is open, providing a dynamic view of the underlying data. Changes made to the underlying column values of rows in the result set are visible. **Updatability** refers to the ability to update data in a result set and then copy the changes to the database. This includes inserting new rows into the result set or deleting existing rows. A result set may be updatable or read-only.

### Summary of Result Set Object Types

Scrollability and sensitivity are independent of updatability, and the three result set types and two concurrency types combine for the following six result set categories:

- Forward-only/read-only
- Forward-only/updatable
- Scroll-sensitive/read-only
- Scroll-sensitive/updatable
- Scroll-insensitive/read-only
- Scroll-insensitive/updatable

Example 4–2 demonstrates how to declare a scroll-sensitive and read-only `ResultSet` object.

***Example 4–2   Declaring a Scroll-Sensitive, Read-Only ResultSet Object***

```
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_
READ_ONLY);
```

> **Note:**   A forward-only updatable result set has no provision for
> positioning at a particular row within the `ResultSet` object. You
> can update rows only as you iterate through them using the `next`
> method.

# Querying Data from a Java Application

This section discusses how you can use JDeveloper to create a Java class that queries
data in Oracle Database in the following sections:

- Creating a Method in JDeveloper to Query Data

- Testing the Connection and the Query Methods

## Creating a Method in JDeveloper to Query Data

The following steps show you how to add a simple query method to your
`DataHandler.java` class. If `DataHandler.java` is not open in the JDeveloper integrated
development environment (IDE), double-click it in the Application Navigator to
display it in the Java Source Editor.

1. In the `DataHandler` class, add the following `import` statements after the existing
   `import` statements to use the `Statement` and `ResultSet` JDBC classes:

   ```
   import java.sql.Statement;
   import java.sql.ResultSet;
   ```

2. After the `connection` declaration, declare variables for `Statement`, `ResultSet`, and
   `String` objects as follows:

   ```
   Statement stmt;
   ResultSet rset;
   String query;
   String sqlString;
   ```

3. Create a method called `getAllEmployees`, which will be used to retrieve employee
   information from the database. Enter the signature for the method:

   ```
   public ResultSet getAllEmployees() throws SQLException{
   ```

4. Press Enter to include a closing brace for this method, and a new line in which to
   start entering the method code.

5. Call the `getDBConnection` method created earlier:

   ```
   getDBConnection();
   ```

6. Use the `createStatement` method of the `Connection` instance to provide context
   for executing the SQL statement and define the `ResultSet` type. Specify a
   read-only, scroll-sensitive `ResultSet` type:

   ```
   stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_
   READ_ONLY);
   ```

   The Java Code Insight feature can help you ensure that the statement syntax is
   correct.

7. Define the query and print a trace message. The following code uses a simple query: it returns all the rows and columns in the `Employees` table and the data is ordered by the Employee ID:

```
query = "SELECT * FROM Employees ORDER BY employee_id";
System.out.println("\nExecuting query: " + query);
```

8. Run the query and retrieve the results in the `ResultSet` instance as follows:

```
rset = stmt.executeQuery(query);
```

9. Return the `ResultSet` object:

```
return rset;
```

10. Save your work. From the **File** menu, select **Save All**.

The code for the `getAllEmployees` method should be as shown in Example 4–3.

**Example 4–3   *Using the Connection, Statement, Query, and ResultSet Objects***

```
public ResultSet getAllEmployees() throws SQLException{
    getDBConnection();
    stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
    query = "SELECT * FROM Employees ORDER BY employee_id";
    System.out.println("\nExecuting query: " + query);
    rset = stmt.executeQuery(query);
    return rset;
}
```

## Testing the Connection and the Query Methods

In the following steps, you create a simple Java class to test the methods in the `DataHandler.java` class. To test your application at this stage, you can temporarily set the value of the `jdbcUrl` variable to the connection string for your database and set the values of the `userid` and `password` variables to the values required to access the `HR` schema ("hr" in each case).

1. Open the `DataHandler.java` class in the Java Visual Editor from the Application Navigator.

2. Change the `jdbcUrl`, `userid` and `password` variables to contain the values required for the `HR` schema as follows:

```
String jdbcUrl = "connect-string"
String userid = "hr";
String password = "hr";
```

where *connect-string* is, for example:

```
jdbc:oracle:thin:@dbhost.companyname.com:1521:ORCL
```

> **See Also:**   Declaring Connection-Related Variables in Chapter 3

3. Create a new Java class in the `hr` package. Name it `JavaClient`, make it a public class, and generate a default constructor and a `main` method. The skeleton `JavaClient.java` class is created and displayed in the Java Source Editor.

> **See Also:**   Chapter 3 for information on creating a Java class file

**4.** Import the `ResultSet` package:

```
import java.sql.ResultSet;
```

**5.** In the `main` method declaration, add exception handling as follows:

```
public static void main(String[] args) throws Exception{
```

**6.** Replace the `JavaClient` object created by default with a `DataHandler` object. Locate the following line:

```
JavaClient javaClient = new JavaClient();
```

Replace this with:

```
DataHandler datahandler = new DataHandler();
```

**7.** Define a `ResultSet` object to hold the results of the `getAllEmployees` query, and iterate through the rows of the result set, displaying the first four columns, `Employee Id`, `First Name`, `Last Name`, and `Email`. To do this, add the following code to the `main` method:

```
ResultSet rset = datahandler.getAllEmployees();

while (rset.next()) {
System.out.println(rset.getInt(1) + " " +
  rset.getString(2) + " " +
  rset.getString(3) + " " +
  rset.getString(4));
}
```

**8.** Compile the `JavaClient.java` file to check for compilation errors. To do this, right-click in the Java Source Editor, and select **Make** from the shortcut menu.

If there are no errors in compilation, you should see the following message in the Log window:

```
 Successful compilation: 0 errors, 0 warnings
```

**9.** Run the `JavaClient.java` file. To do this, right-click in the Java Source Editor window and select **Run** from the shortcut menu.

**10.** Examine the output in the Log window. Notice the trace message, followed by the four columns from the `Employees` table as shown in Figure 4–1.

*Figure 4–1   Test Output for Query Method in Log Window*
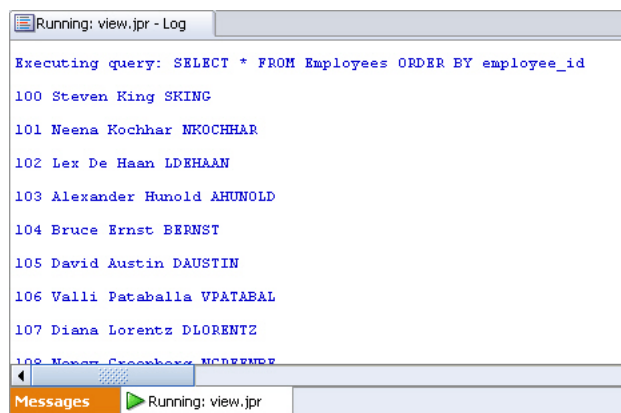
11. When you finish testing the application, set the `jdbcUrl`, `userid` and `password` variables in `DataHandler.java` back to `null`.

> **See Also:** Declaring Connection-Related Variables

# Creating JSP Pages

The `HRApp` application uses JavaServer Pages (JSP) technology to display data. JSP technology provides a simple, fast way to create server-independent and platform-independent dynamic Web content. A JSP page has the `.jsp` extension. 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.

To develop JSP pages, you use some or all of the following:

- HTML tags to design and format the dynamically generated Web page
- Standard JSP 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

> **See Also:** Sun Microsystems documentation for JSP at
>
> http://java.sun.com/products/jsp/

In this section, you will see how you can create JSP pages for the application in this guide in the following sections:

- Overview of Page Presentation
- Creating a Simple JSP Page
- Adding Static Content to a JSP Page
- Adding a Style Sheet to a JSP Page

## Overview of Page Presentation

In the application created in this guide, JSP pages are used to do the following tasks:

- Display data.
- Hold input data entered by users adding employees and editing employee data.
- Hold the code needed to process the actions of validating user credentials and adding, updating, and deleting employee records in the database.

Because JSP pages are presented to users as HTML or XML, you can control the presentation of data in the same way as you would for static HTML and XML pages. You can use standard HTML tags to format your page, including the `title` tag in the header to specify the title to be displayed for the page.

You use HTML tags for headings, tables, lists and other items on your pages. Style sheets can also be used to define the presentation of items. If you use JDeveloper to develop your application, you can select styles from a list.

The following sections describe the main elements used in the JSP pages of the sample application:

- JSP Tags
- Scriptlets

- [HTML Tags](#)
- [HTML Forms](#)

### JSP Tags

JSP tags are used in the sample application in this guide for the following tasks: to initialize Java classes that hold the application methods and the JavaBean used to hold a single employee record, and to forward the user to either the same or another page in the application.

The `jsp:useBean` tag is used in pages to initialize the class that contains all the methods needed by the application, and the `jsp:forward` tag is used to forward the user to a specified page. You can drag the tags you need from the Component Palette of JSP tags, and enter the properties for the tag in the corresponding dialog box that is displayed.

> **See Also:** ▪ http://java.sun.com/products/javabeans/ for more information on JavaBeans

### Scriptlets

Scriptlets are used to run the Java methods that operate on the database and to perform other processing in JSP pages. You can drag a scriptlet tag component from the Component Palette and drop it onto your page, ready to enter the scriptlet code. In JDeveloper, the code for scriptlets is entered in the Scriptlet Source Editor dialog box.

In this application, you use scriplets for a variety of tasks. As an example, one scriptlet calls the `DataHandler` method that returns a `ResultSet` object containing all the employees in the `Employees` table, which you can use to display that data in your JSP page. As another example, a scriplet is used to iterate through the same `ResultSet` object to display each item in a row of a table.

### HTML Tags

HTML tags are typically used for layout and presentation of the nondynamic portions of the user interface, for example headings and tables. In JDeveloper, you can drag and drop a Table component from the Component Palette onto your page. You must specify the number of rows and columns for the table, and all the table tags are automatically created.

### HTML Forms

HTML forms are used to interact with or gather information from the users on Web pages. The `FORM` element acts as a container for the controls on a page, and specifies the method to be used to process the form input.

For the filter control to select which employees to display, the `employees.jsp` page itself processes the form. For login, insert, edit, and delete operations, additional JSP pages are created to process these forms. To understand how the JSP pages in this application are interrelated, refer to Figure 1–2.

You can add a form in a JSP page by selecting it from the Component Palette of HTML tags. If you attempt to add a control on a page outside of the form component or in a page that does not contain a form, then JDeveloper prompts you to add a form component to contain it.

## Creating a Simple JSP Page

The following steps describe how to create a simple JSP page:

1. In the Application Navigator, right-click the **View** project and choose **New** from the shortcut menu.

2. In the New Gallery, from the **Filter By:** list, select **All Technologies**.

3. Expand the **Web Tier** node and select **JSP**.

4. In the **Items** list, select **JSP** and click **OK**. The Create JSP Wizard is displayed.

5. On the JSP File screen, enter a name for the JSP page and select **JSP Page**.

6. On the Error Page Options screen, select **Do Not Use an Error Page to Handle Uncaught Exceptions in this File**.

---

**Note:** You create an error page to make Java exceptions thrown by an application more readable for the user. When you create an error page, you specify a user-friendly interpretation of possible application errors. However, functionally, this is not a required feature in the application and has not been covered in this guide.

---

7. Do not add any JSP tag libraries at this stage. Click **Next** on the Tag Libraries screen.

8. Retain the default settings on the **HTML Options** screen and click **Finish**. The new page opens in the JSP/HTML Visual Editor and is ready for you to start adding text and components to your web page.

## Adding Static Content to a JSP Page

JDeveloper provides the Component Palette and the Property Inspector on the right hand side of the JSP/HTML Visual Editor. You can also use the JSP Source Editor by clicking the Source Editor tab next to the Design tab at the bottom of the page. The Component Palette allows you to add components to the page and the Property Inspector allows you to set the properties of the components. A blank page in the Visual Editor is shown in Figure 4–2.

Figure 4–2  Adding Content to JSP Pages in the JDeveloper Visual Source Editor
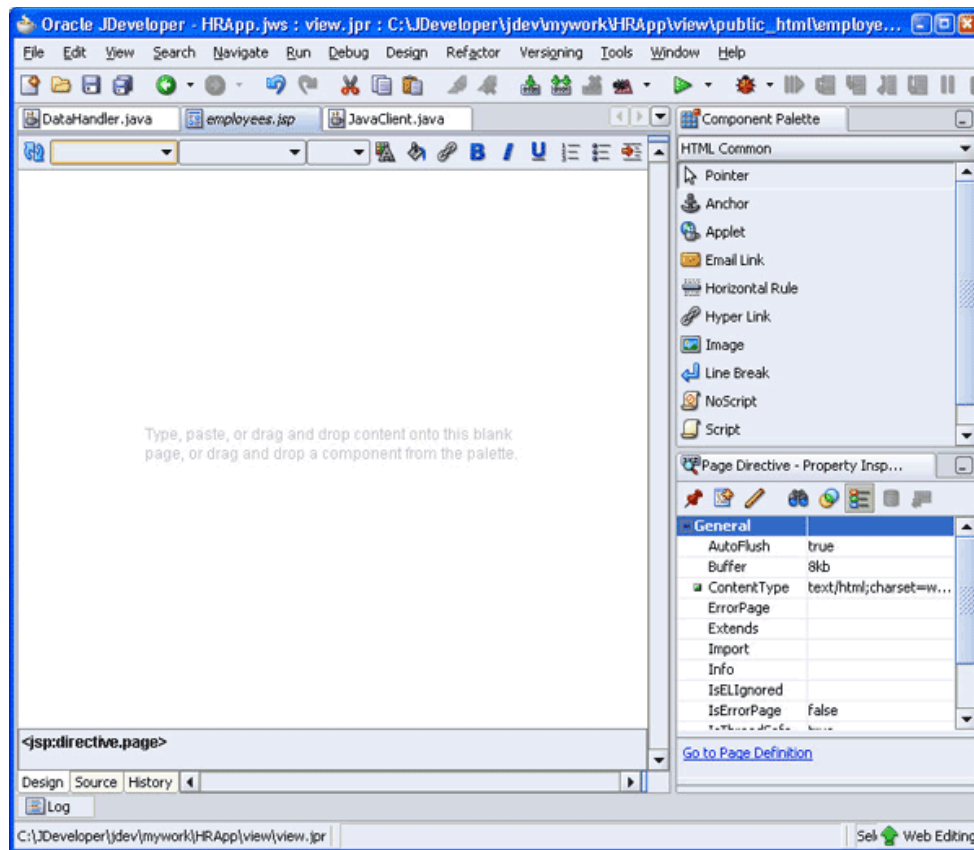


The following steps show how you can add text to the `employees.jsp` page. They use the Visual Editor to modify the JSP. The Visual Editor is like a WYSIWYG editor and you can use it to modify content.

1. With `employees.jsp` open in the Visual Editor, in the top line of your page, enter **AnyCo Corporation: HR Application**. From the list of styles at the top of the page, on the left-hand side, select **Heading 2**.

2. With the cursor still on the heading you added, from the **Design** menu select **Align**, and then **Center**.

3. In a similar way, on a new line, enter **Employee Data**, and format it with the **Heading 3** style. Position it on the left-hand side of the page.

## Adding a Style Sheet to a JSP Page

You can add a style sheet reference to your page, so that your headings, text, and other elements are formatted in a consistent way with the presentation features, such as the fonts and colors used in the Web pages. You can add a style sheet to the page as follows:

1. With `employees.jsp` open in the Visual Editor, click the list arrow at the top right of the Component Palette, and select **CSS**.

2. From the **CSS** list, drag **JDeveloper** onto your page. As soon as you select the style sheet it is added to your page and formats the page with the JDeveloper styles. Figure 4–3 shows the JSP Page with the content added to it in the previous section and the JDeveloper stylesheet applied to it.

*Figure 4–3   Adding Static Content to the JSP Page*
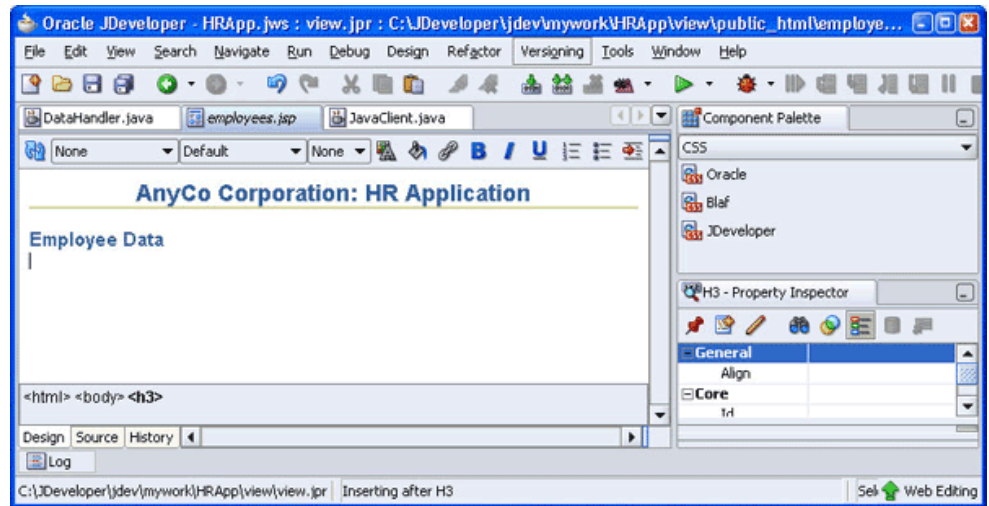


> **Note:**   In JDeveloper version 10.1.3, you can associate a stylesheet
> with the JSP page while creating it in the JSP Creation Wizard. The
> only difference is that you need to browse and locate the stylesheet to
> be applied to the JSP page, instead of just dragging and dropping it
> onto the page.

## Adding Dynamic Content to the JSP Page: Database Query Results

This section includes the following subsections:

- Adding a JSP useBean Tag to Initialize the DataHandler Class
- Creating a Result Set
- Adding a Table to the JSP Page to Display the Result Set

### Adding a JSP useBean Tag to Initialize the DataHandler Class

A `jsp:useBean` tag identifies and initializes the class that holds the methods that run
in the page. To add a `jsp:useBean` tag, follow these steps:

1.  Open `employees.jsp` in the Visual Editor.

2.  In the Component Palette, select the **JSP** set of components. Scroll through the list
    to select **UseBean**. Then, drag and drop it onto your JSP page, below the headings.

3.  In the Insert UseBean dialog box, enter `empsbean` as the **ID**, and for the **Class**,
    browse and select the `hr.DataHandler` class. Set the **Scope** to `session`, and leave
    the Type and BeanName fields blank.

4.  Click **OK** to create the tag in the page.

Figure 4–4 shows the representation of the `useBean` tag in the `employees.jsp` page.

**Figure 4–4   useBean Representation in the employees.jsp File**
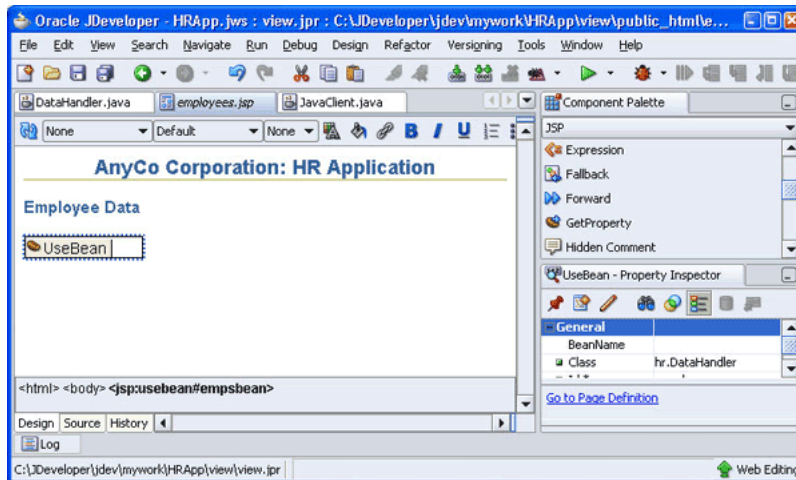


## Creating a Result Set

The following steps describe how you can add a scripting element to your page to call the `getAllEmployees` method and hold the result set data that is returned. This query is defined in the `DataHandler` class, and initialized in the page by using the `jsp:useBean` tag.

1. Open the `employees.jsp` page in the Visual Editor. In the JSP part of the Component Palette, select **Scriptlet** and drag and drop it onto the JSP page next to the representation of the UseBean.

2. In the Insert Scriptlet dialog box, enter the following lines of code, which will call the `getAllEmployees` method and produce a `ResultSet` object:

```
ResultSet rset;
rset = empsbean.getAllEmployees();
```

Click **OK**. A representation of the scriptlet is displayed on the page as shown in Figure 4–5.

**Figure 4–5   Scriptlet Representation in a JSP Page**
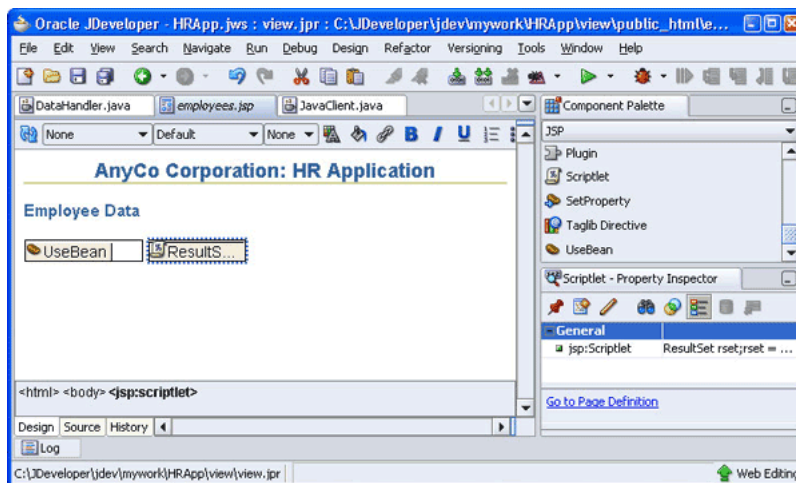
**3.** Select the **Source** tab at the bottom of the Visual Editor to see the code that has been created for the page so far. A wavy line under `ResultSet` indicates that there are errors in the code.

**4.** The Structure window on the left-hand side also indicates any errors in the page. Scroll to the top of the window and expand the **JSP Errors** node. Figure 4–6 shows how the error in the code is shown in the Structure window.

*Figure 4–6   Viewing Errors in the Structure Window*
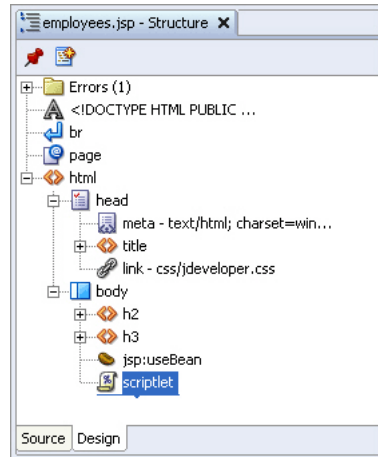


**5.** You must import the `ResultSet` package. To do this, click the **page** node in the Structure window to display the page properties in the Property Inspector.

**6.** Click in the empty box to the right of the **import** property. Click the ellipsis symbol (**...**). The import dialog box is displayed, which is shown in Figure 4–7.

*Figure 4–7   Importing Packages in JDeveloper*
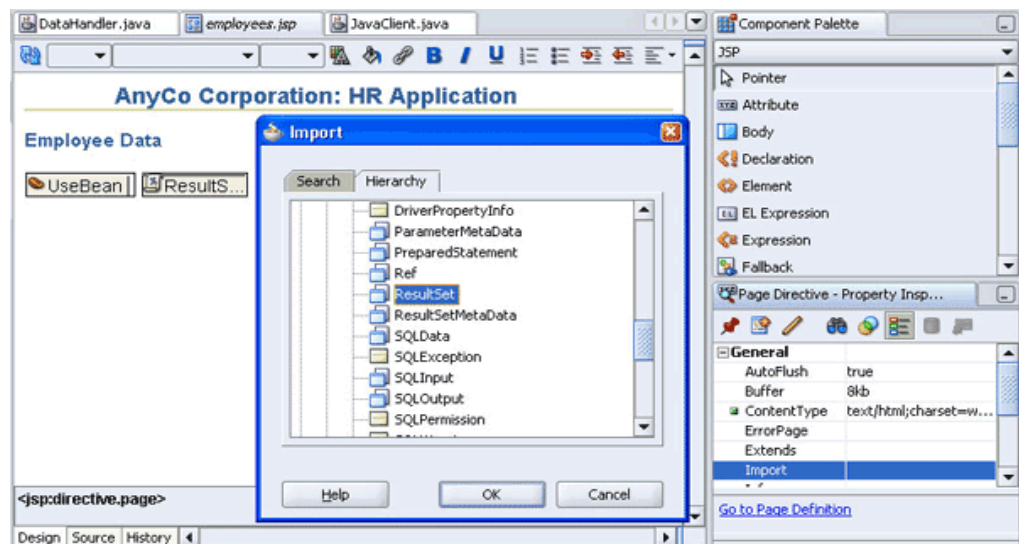


**7.** In the import list, select the **Hierarchy** tab, expand the **java** node, then the **sql** node, and then select **ResultSet**. Click **OK**.

**8.** On the Source tab, examine the code to see if the `import` statement has been added to the code for your page. The error should disappear from the list in the Structure

window. Before continuing with the following sections, return to the design view of the page by selecting the **Design** tab.

## Adding a Table to the JSP Page to Display the Result Set

The following steps describe how you can add a table to the JSP page to display the results of the getAllEmployees query:

1.  If the employees.jsp page is not open in the Visual Editor, double-click it in the Application Navigator to open it, and work in the Design tab. With the employees.jsp file open in the Visual Editor, position the cursor after the scriptlet and from the HTML Common page of the Component Palette, select the **Table** component.

2.  In the Insert Table dialog box, specify 1 row and 6 columns. Leave all Layout properties as defaults. Click **OK**.

3.  In the table row displayed on the page, enter text as follows for the headings for each of the columns: **First Name**, **Last Name**, **Email**, **Job**, **Phone**, **Salary**. Use **Heading 4** to format the column names.

4.  Add a scripting element for output, this time to display the values returned for each of the columns in the table. To do this, select the table as follows. Position the cursor on the top border of the table, and click when the cursor image changes to a table image. From the JSP Component Palette, select **Scriptlet**. (You need not drag the scriptlet into your table; it is inserted automatically.)

5.  In the Insert Scriptlet dialog box, enter the following lines of code:

```
while (rset.next ())
{
out.println("<tr>");
out.println("<td>" +
  rset.getString("first_name") + "</td><td> " +
  rset.getString("last_name") + "</td><td> " +
  rset.getString("email") + "</td><td> " +
  rset.getString("job_id") + "</td><td>" +
  rset.getString("phone_number") + "</td><td>" +
  rset.getDouble("salary") + "</td>");
out.println("</tr>");
}
```

6.  Click **OK**.

The JSP page created is shown in Figure 4–8.

*Figure 4–8   Table in a JSP Page*
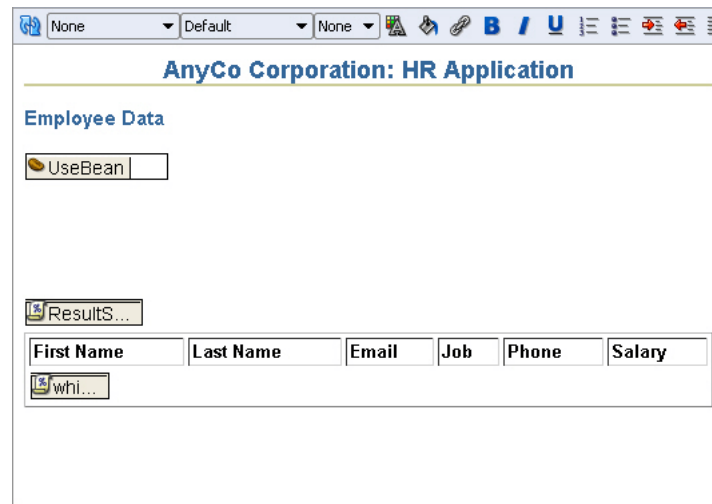


## Filtering a Query Result Set

You can filter the results of a query by certain parameters or conditions. You can also allow users of the application to customize the data filter. In the sample application created in this guide, the procedure of filtering the query result consists of the following tasks:

1.  Determining what filtered set is required

    Users can specify the set of employee records that they want to view by entering a filter criterion in a query field, in this case, a part of the name that they want to search for. The employees.jsp page accepts this input through form controls, and processes it.

2.  Creating a method to return a query ResultSet

    The user input string is used to create the SQL query statement. This statement selects all employees whose names include the sequence of characters that the user enters. The query searches for this string in both the first and the last names.

3.  Displaying the results of the query

    This is done by adding code to the employees.jsp page to use the method that runs the filtered query.

This section describes filtering query data in the following sections:

- Creating a Java Method for Filtering Results

- Testing the Query Filter Method

- Adding Filter Controls to the JSP Page

- Displaying Filtered Data in the JSP Page

### Creating a Java Method for Filtering Results

The following steps describe how you can create the getEmployeesByName method. This method allows users to filter employees by their first or last name.

1.  From the Application Navigator, open the DataHandler.java class in the Java Visual Editor.

2. After the `getAllEmployees` method, declare the `getEmployeesByName` method as follows:

```
public ResultSet getEmployeesByName(String name) throws SQLException {

}
```

3. Within the body of the method, add the following code to convert the name to uppercase to enable more search hits:

```
name = name.toUpperCase();
```

4. Call the method to connect to the database:

```
getDBConnection();
```

5. Specify the `ResultSet` type and create the query:

```
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_READ_ONLY);
query =
"SELECT * FROM Employees WHERE UPPER(first_name) LIKE \'%" + name + "%\'" +
" OR UPPER(last_name) LIKE \'%" + name + "%\' ORDER BY employee_id";
```

6. Print a trace message:

```
System.out.println("\nExecuting query: " + query);
```

7. Run the query and return a result set as before:

```
rset = stmt.executeQuery(query);
return rset;
```

8. Save the file and compile it to ensure there are no compilation errors.

## Testing the Query Filter Method

You can use the `JavaClient.java` class created in Testing the Connection and the Query Methods to test the `getEmployeesByName` method. You must add the `getEmployeesByName` method to display the query results as described in the following steps:

1. Open the `JavaClient.java` class in the Java Source Editor.

2. After the result set displaying the results from the `getAllEmployees` query, define a result set for the conditional query as follows:

```
rset = datahandler.getEmployeesByName("King");

System.out.println("\nResults from query: ");

while (rset.next()) {
  System.out.println(rset.getInt(1) + " " +
  rset.getString(2) + " " +
  rset.getString(3) + " " +
  rset.getString(4));
}
```

3. To test your application at this stage, you can temporarily adjust the values of the `jdbcUrl`, `userid` and `password` variables in the `DataHandler` class to provide the values required for the `HR` schema. Save the file, and compile it to check for syntax errors.

> **Note:** Make sure you change the values of `userid`, `password`, and `jdbcUrl` back to `null` after testing. For more information, refer to Declaring Connection-Related Variables.

4. To test-run the code, right-click in the Java Source Editor and select **Run** from the shortcut menu. In the Log window, you will first see the results of the `getAllEmployees` method, then the results from the `getEmployeesByName("xxx")` query. Here, xxx is set to "King" to test the filtering functionality. In actual operation, this parameter will be set to the value provided by the user of the application to filter the search.

## Adding Filter Controls to the JSP Page

To accept the filter criterion and to display the filter results, you must modify the `employees.jsp` page. In the following steps, you add a form element and controls to the `employees.jsp` page that accepts input from users to filter employees by name:

1. With the `employees.jsp` page displayed in the Visual Editor, position the cursor between the `useBean` tag and the scriptlet.

2. In the HTML Forms page of the Component Palette, select **Form**.

3. In the Insert Form dialog box, use the down arrow for the Action field and select **employees.jsp**. Leave the other fields empty and click **OK**.

   The form is displayed on the page in the Visual Editor, represented by a dotted-line rectangle.

4. In the HTML Forms page of the Component Palette, scroll to **Text Field**. Select it and drag and drop it inside the Form component. In the Insert Text Field dialog, enter `query` as the value of the **Name** field and click **OK**. The text field box is displayed within the form. This field allows users to enter filter criteria.

5. Position the cursor to the left of the Text Field and add the following text:

   **Filter by Employee name:**

6. In the HTML Forms page of the Component Palette, scroll to **Submit Button**. Select it and drop it inside the **Form** component to the right of the Text Field.

7. In the Insert Submit Button dialog box, leave the **Name** field empty and enter `Filter` as the value of the **Value** field, and click **OK**.

   Figure 4–9 shows these HTML Form components in the `employees.jsp` file.

*Figure 4–9   HTML Form Components in the JSP Page*



## Displaying Filtered Data in the JSP Page

In the previous section, you created a text field component on the JSP page that accepts user inputs. In this text field, users can specify a string with which to filter employee names. You also added a submit button.

In the following steps, you add code to the scriptlet in the employees.java file to enable it to use the getEmployeesByName method. This method is used only if a user submits a value for filtering the results. If this filter criterion is not specified, the getAllEmployees method is used.

1.  Open the employees.jsp file in the Visual Editor.

2.  Double-click the **Scriptlet** tag on the page (not the one inside the table) to open the Properties dialog box. Modify the code as follows:

```
ResultSet rset;
String query = request.getParameter("query");
if (query != null && query != null)
  rset = empsbean.getEmployeesByName(query);
else
  rset = empsbean.getAllEmployees();
```

Figure 4–10 shows how you can use the Scriptlet Properties dialog box to modify the code.

*Figure 4–10   Using the Scriptlet Properties Dialog Box*
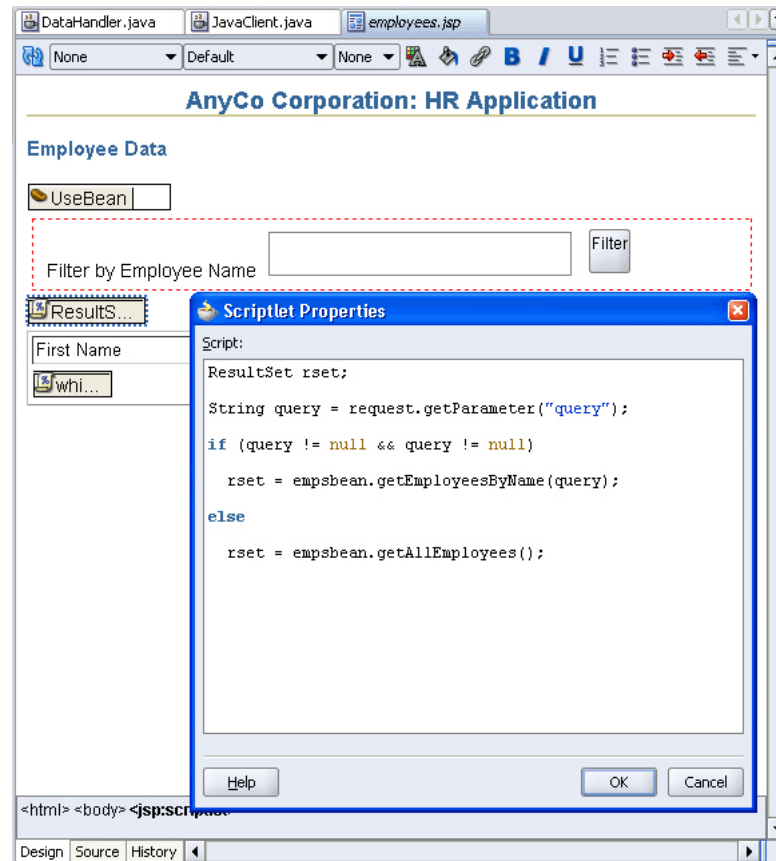


3. Click **OK**.

4. Save the file.

# Adding Login Functionality to the Application

The login functionality used in the sample application is a simple example of application-managed security. It is not a full Java EE security implementation, but simply used as an example in the sample application.

To implement this simple login functionality, you must perform the following tasks:

- Creating a Method to Authenticate Users

- Creating a Login Page

- Preparing Error Reports for Failed Logins

- Creating the Login Interface

- Creating a JSP Page to Handle Login Action

## Creating a Method to Authenticate Users

In the following steps, you create a method in the `DataHandler.java` class that authenticates users by checking that the values they supply for the `userid` and `password` match those required by the database schema.

1. Open the `DataHandler.java` class in the Source Editor.

**2.** Create a method called `authenticateUser` that checks if the `userid`, `password`, and `host` values supplied by a user are valid:

```
public boolean authenticateUser(String jdbcUrl, String userid, String password,
  HttpSession session) throws SQLException {


}
```

**3.** JDeveloper prompts you with a wavy underline and a message that you need to import a class for `HttpSession`. Press the Alt+Enter keys to import the `javax.servlet.http.HttpSession` class.

**4.** Within the body of the method, assign the `jdbcUrl`, `userid`, and `password` values from the call to the attributes of the current object as follows:

```
this.jdbcUrl= jdbcUrl;
this.userid = userid;
this.password = password;
```

**5.** Attempt to connect to the database using the values supplied, and if successful, return a value of `true`. Enclose this in a `try` block as follows:

```
try {
  OracleDataSource ds;
  ds = new OracleDataSource();
  ds.setURL(jdbcUrl);
  conn = ds.getConnection(userid, password);
  return true;
}
```

> **See Also:** For information about using `try` and `catch` blocks, refer to Exception Handling in Chapter 5.

**6.** To handle the case where the login credentials do not match, after the `try` block, add a `catch` block. The code in this block prints out a log message and sets up an error message. This error message can be displayed to the user if a login attempt fails. The `jdbcUrl`, `userid` and `password` variables are set back to `null`, and the method returns the value `false`. To do this, enter the following code:

```
catch ( SQLException ex )  {
  System.out.println("Invalid user credentials");
  session.setAttribute("loginerrormsg", "Invalid Login. Try Again...");
  this.jdbcUrl = null;
  this.userid = null;
  this.password = null;
  return false;
}
```

The complete code is shown in Example 4–4.

**Example 4–4   Implementing User Validation**

```
public boolean authenticateUser(String userid, String password,
  HttpSession session) throws SQLException {

  this.jdbcUrl = jdbcUrl;
  this.userid = userid;
  this.password = password;
  try {
    OracleDataSource ds;
    ds = new OracleDataSource();
```

```
    ds.setURL(jdbcUrl);
    conn = ds.getConnection(userid, password);
    return true;
} catch ( SQLException ex )  {
System.out.println("Invalid user credentials");
session.setAttribute("loginerrormsg", "Invalid Login. Try Again...");
this.jdbcUrl = null;
this.userid = null;
this.password = null;
return false;
}
}
```

## Creating a Login Page

The following steps create a `login.jsp` page, on which users enter the login details for the schema they are going to work on:

1.  In the View project, create a new JSP page. Change the Name to `login.jsp` and accept all other defaults. The new page opens in the JSP/HTML Visual Editor and is ready for you to start adding text and components to your Web page.

2.  Apply the JDeveloper style sheet to the page.

3.  Give the page the same heading as earlier, **AnyCo Corporation: HR Application**, apply the **Heading 2** style to it, and align it to the center of the page.

4.  On the next line, enter **Application Login**, with the **Heading 3** style applied. Align this heading to the left-hand side of the page.

## Preparing Error Reports for Failed Logins

The following steps add functions to the `login.jsp` page for displaying error messages when a user login fails. The scriptlets and expression used in the `login.jsp` page set up a variable to hold any error message. If the user login fails, the connection method sets a message for the session. This page checks to see if there is such a message, and if present, it displays the message.

1.  With the `login.jsp` page open in the Visual Editor, position the cursor after the text on this page. Then, from the JSP page of the Component Palette, drag and drop the **Scriptlet** element from the palette onto the page.

2.  In the Insert Scriptlet dialog box, enter the following code:

    ```
    String loginerrormsg = null;
    loginerrormsg = (String) session.getAttribute("loginerrormsg");
    if (loginerrormsg != null) {
    ```

3.  Add another scriptlet in exactly the same way, and this time enter only a single closing brace (}) in the Insert Scriptlet dialog box.

4.  Place the cursor between the two scriptlets and press Enter to create a new line. Apply the **Heading 4** style to the new line.

5.  With the cursor still on the new line, in the JSP page of the Component Palette, click **Expression**.

6.  In the Insert Expression dialog box, enter `loginerrormsg`.

7.  To see the code that has been added to your `login.jsp` page, below the Visual Editor, select the **Source** tab. The code should appear as follows:

    ```
    <%
    ```

```
    String loginerrormsg = null;
    loginerrormsg = (String) session.getAttribute("loginerrormsg");
    if (loginerrormsg != null) {
%>
<h4>
  <%=  loginerrormsg %>
</h4>
<%
}
%>
```

Before continuing with the following sections, return to the design view of the page by selecting the **Design** tab.

## Creating the Login Interface

In these steps, you add fields to the login.jsp page on which users enter their login details.

1. If the login.jsp page is not open in the Visual Editor, double-click it in the Application Navigator to open it, and check that the Design tab is selected.

2. Position the cursor after the second scriptlet and, in the HTML Forms page of the Component Palette, select **Form**. The Form is displayed on the page in the Visual Editor, represented by a dotted-line rectangle.

3. In the HTML Forms page of the Component Palette, select **Form**. In the Insert Form dialog box, enter login_action.jsp as the value for the **Action** field. This file will be used to process the user input in the **login.jsp** file. (You cannot select this page from a list as it is not created yet.) Leave the other fields empty and click **OK**.

   The Form is displayed on the page in the Visual Editor, represented by a dotted rectangle.

4. Add a **Table** to the page. Position it inside the Form. Specify a 3-row and 2-column layout, and accept other layout defaults.

5. In the first column of the three rows, enter the following as the text to display for users:

   **User ID:**

   **Password:**

   **Host:**

6. From the HTML page of the Component Palette, drag a **Text Field** into the table cell to the right of the User ID: cell. In the Insert Text Field dialog box, enter userid as the value of the **Name** property. Leave the other fields empty and click **OK**.

7. In the same way, add a **Text Field** to the table cell to the right of the Password: cell and enter password as the value of the **Name** property. Similarly, add a **Text Field** to the table cell to the right of the Host: cell and enter host as the value of the **Name** property.

8. Drag a **Submit** button to the Form below the table. Enter Submit for the **Value** property of the button.

Your login.jsp page should now appear as shown in Figure 4–11.

*Figure 4–11   Login Page*
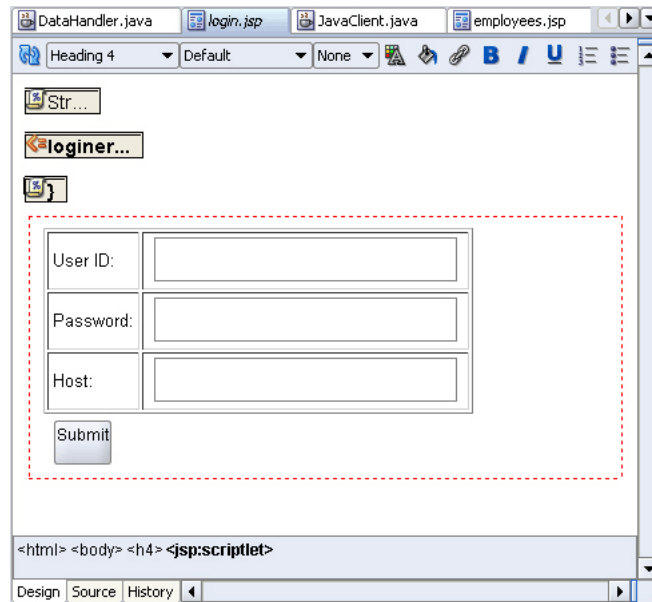


## Creating a  JSP Page to Handle Login Action

In the following steps, you create the `login_action.jsp` page, which is a nonviewable page that processes the login operation.

1. Create a JSP page and call it `login_action.jsp`. Accept all default settings for the JSP page.

2. With `login_action.jsp` open in the Visual Editor, from the JSP page of the Component Palette, drag a Page Directive component to the page. In the Insert Page Directive dialog box, for the **Import** field, browse to import **java.sql.ResultSet**. Click **OK**.

3. Drag a **jsp:usebean** tag onto the page. Enter `empsbean` as the **ID** and browse to select **hr.DataHandler** as the **Class**. Set the **Scope** to `session`, and click **OK**.

4. Position the cursor after the useBean tag and add a **Scriptlet** to the page. Enter the following code into the Insert Scriptlet dialog box and click **OK**.

```
boolean userIsValid = false;
String host = request.getParameter("host");
String userid = request.getParameter("userid");
String password = request.getParameter("password");
String jdbcUrl = "jdbc:oracle:thin:@" + host;
userIsValid = empsbean.authenticateUser(jdbcUrl, userid, password, session);
```

5. Add another **Scriptlet**, and add the following code to it:

```
if (userIsValid){
```

6. In the JSP page of the Component Palette, find **Forward** and drag it onto the page to add a **jsp:forward** tag onto the page. In the Insert Forward dialog box, enter **employees.jsp**.

7. Add another scriptlet, and enter the following code:

```
} else {
```

8. Add another **jsp:forward** tag, and this time move forward to `login.jsp`.

9. Add a final **Scriptlet**, and enter a closing brace (}).

10. Save your work.

To see the code that has been added to login_action.jsp, select the **Source** tab. The code displayed is similar to the following:

```
<body>
<%@ page import="java.sql.ResultSet"%><jsp:useBean id="empsbean"
                                                   class="hr.DataHandler"
                                                   scope="session"/>
<%boolean userIsValid = false;
String host = request.getParameter("host");
String userid = request.getParameter("userid");
String password = request.getParameter("password");
String jdbcUrl = "jdbc:oracle:thin:@" + host + ":1521:ORCL";
userIsValid = empsbean.authenticateUser(jdbcUrl, userid, password, session);%><%if
(userIsValid){%><jsp:forward page="employees.jsp"/><%if
(userIsValid){%><jsp:forward page="login.jsp"/><%}%>
</body>
```

# Testing the JSP Page

To test the login page and the filtering of employees, do the following:

1. In the Application Navigator, right-click the view **project**, and select **Run**.

   You might be prompted to specify a Default Run Target for the project. For now, set this to login.jsp. You can later change the project properties for the default run target page to be any page of your choice.

   The login page is displayed in your browser, as shown in Figure 4–12.

*Figure 4–12   Login Page for Sample Application in the Browser*
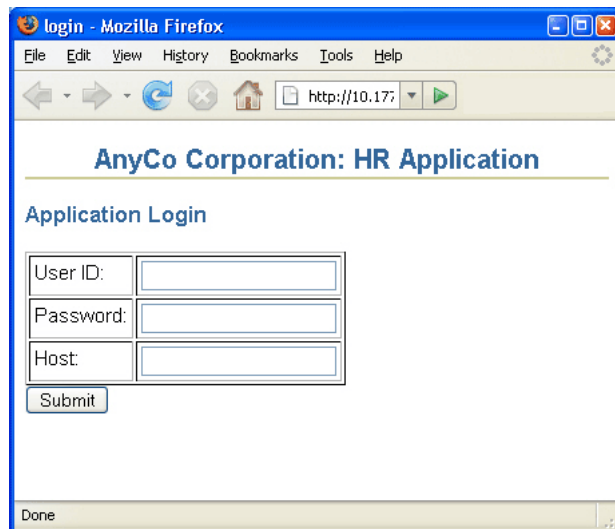


2. Enter the following login details for your database, and then click **Submit**.
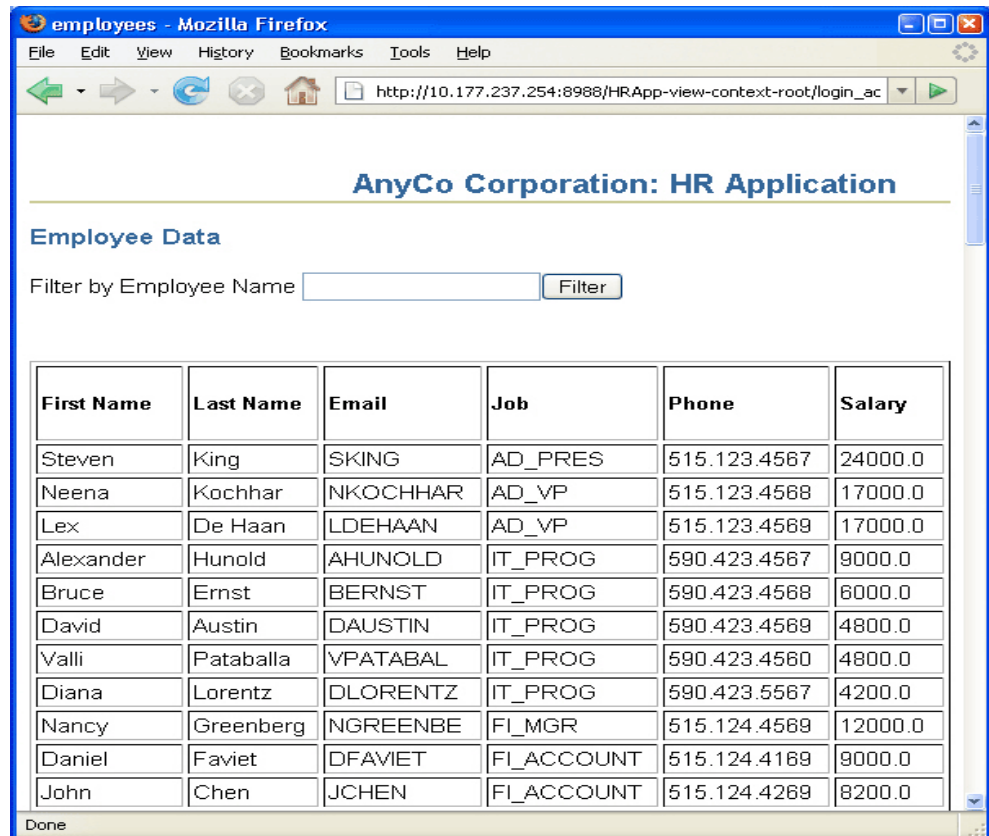
   **User ID:** hr

   **Password:** hr

   **Host:** *Host name of the machine with Oracle Database*

The `Employee.java` file is displayed in your browser as shown in Figure 4–13.

*Figure 4–13    Unfiltered Employee Data in employee.jsp*



3. Enter a string of letters by which you want to filter employee data. For example, enter `ing` in the **Filter by Employee Name** field, and click **Filter**. A filtered list is displayed, which is shown in:

*Figure 4–14    Filtered Employee Data in employee.jsp*

# 5

# Updating Data

In this chapter, you will see how you can modify the sample application and add functionality that allows users to edit, update, and delete data in Oracle Database. This chapter includes the following sections:

- Creating a JavaBean
- Updating Data from a Java Class
- Inserting an Employee Record
- Deleting an Employee Record
- Exception Handling
- Navigation in the Sample Application

## Creating a JavaBean

In outline, a bean is a Java class that has properties, events and methods. For each of its properties, the bean also includes accessors, that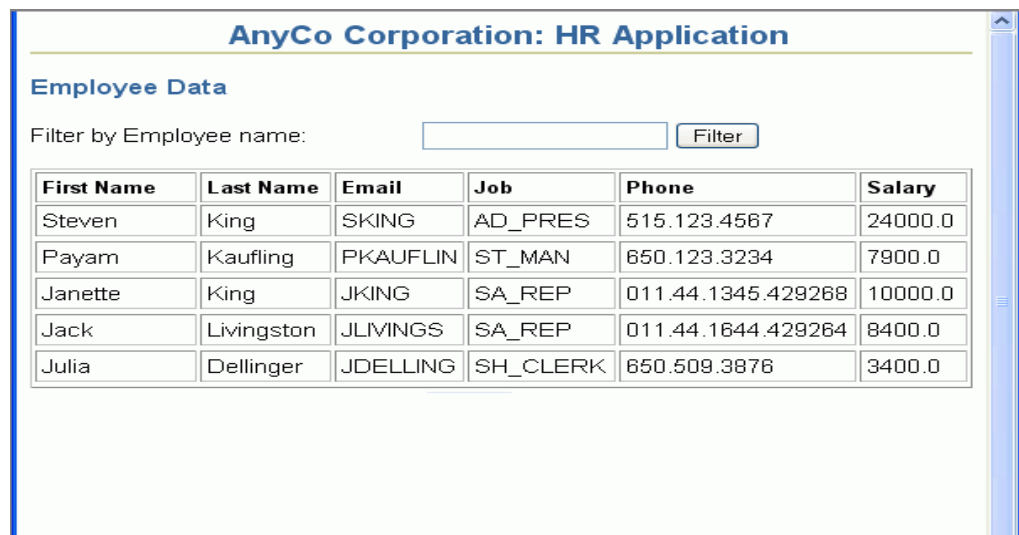 is `get` and `set` methods. Any object that conforms to certain basic rules can be a bean. There is no special class that has to be extended to create a bean.

In the steps for creating a sample application in this chapter, a JavaBean is used to hold a single employee record. When a user wants to edit an existing record or add a new one, it is used as a container to hold the changed or new values for a single row of a table to prepare the row for using to update the database.

The bean contains properties for each field in an employee record, and then JDeveloper creates the accessors (`get` and `set` methods) for each of those properties. You will see how to create a JavaBean for the sample application in the following subsections:

- Creating a JavaBean in JDeveloper
- Defining the JavaBean Properties and Methods

## Creating a JavaBean in JDeveloper

`Employee.java` is the JavaBean that is used in the sample application to hold a single employee record and modify its contents. To create a JavaBean, do the following:

1. Right-click the **View** project, and from the shortcut menu, click **New**.

2. In the New Gallery dialog box, in the Filter By field select **All Technologies**.

3. Expand the General category and select **JavaBeans** in the **General** category. From the **Items** list, select **Bean**. Click **OK**.

**4.** In the Create Bean dialog box, enter `Employee` as the name, `hr` as the package, and ensure that the **Extends:** field is set to `java.lang.Object`. Click **OK** to create the bean.

**5.** Save the file. The `Employee.java` file should now contain the following code:

```
package hr;

public class Employee {
  public Employee(){
  }
}
```

## Defining the JavaBean Properties and Methods

In the JavaBean, you must create one field for each column in the `Employees` table, and accessor methods (`get` and `set` methods) for each field.

**1.** Add an import statement for `java.sql.Date`, which is the field type for one of the fields:

```
import java.sql.Date;
```

**2.** Add a field to the `Employee` class for each of the columns in the `Employees` table. Each field is `private`, and the field types are as follows:

```
private Integer employeeId;
private String firstName;
private String lastName;
private String email;
private String phoneNumber;
private Date hireDate;
private String jobId;
private Double salary;
private Double commissionPct;
private Integer departmentId;
```

**3.** Right-click on the Source Editor page and select **Generate Accessors** from the shortcut menu. In the Generate Accessors dialog box, select the top-level **Employee** node. A check mark is displayed for that node and for all the fields. Click **OK**. Figure 5–1 shows the Generate Accessors dialog box with all the fields selected.

*Figure 5–1   Generate Accessors Dialog Box*
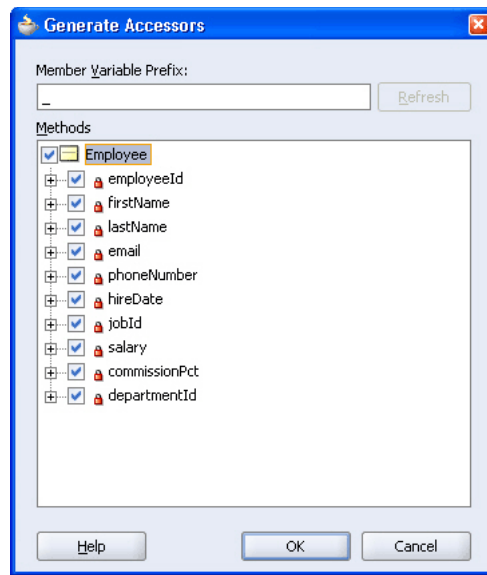


4. Save the file. The `Employee.java` file should now contain the following code:

*Example 5–1   Skeleton Code for a Basic Java Bean with Accessor Methods*

```
package hr;
import java.sql.Date;

public class Employee {
    public Employee() {
    }
    private Integer employeeId;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    private Date hireDate;
    private String jobId;
    private Double salary;
    private Double commissionPct;
    private Integer departmentId;

    public void setEmployeeId(Integer employeeId) {
        this.employeeId = employeeId;
    }

    public Integer getEmployeeId() {
        return employeeId;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return firstName;
    }
...
...
```

```
...
...
// This list has been shortened and is not comprehensive. The actual code contains
// accessor methods for all the fields declared in the bean.

    public void setDepartmentId(Integer departmentId) {
        this.departmentId = departmentId;
    }

    public Integer getDepartmentId() {
        return departmentId;
    }
}
```

# Updating Data from a Java Class

Updating a row in a database table from a Java application requires you to do the following tasks:

1. Create a method that finds a particular employee row. This is used to display the values for a particular employee on an edit page.

2. Create a method that takes the updated employee data from the bean and updates the database.

3. On the main application page, in every row of employee data, include a link that allows a user to edit the data for that employee. The links take the user to the edit.jsp file with the data for that employee displayed, ready for editing.

4. Create a JSP page called edit.jsp, that includes a form and a table to display all the data of a single employee and allows a user to change the values.

5. Create a JSP page that processes the form on the edit.jsp page, writes the updated values to the Employee.java bean and calls the updateEmployee method.

You will see how to do this in the following sections:

- Creating a Method to Identify an Employee Record

- Creating a Method to Update Employee Data

- Adding a Link to Navigate to an Update Page

- Creating a JSP Page to Edit Employee Data

- Creating a JSP Page to Handle an Update Action

## Creating a Method to Identify an Employee Record

The method you create in these steps is used to find the record for a particular employee. It is used when a user wants to edit or delete a particular employee record, and selects a link for that employee on the Employee.java page.

1. If the DataHandler class is not already open in the Java Source Editor, double-click it in the Application Navigator to open it.

2. In the DataHandler class, declare a new method that identifies the employee record to be updated:

```
public Employee findEmployeeById(int id) throws SQLException {

}
```

**3.** Within the body of this method, create a new instance of the `Employee` bean called `selectedEmp`.

```
Employee selectedEmp = new Employee();
```

**4.** Connect to the database.

```
getDBConnection();
```

**5.** Create a `Statement` object, define a `ResultSet` type, and formulate the query. Add a trace message to assist with debugging.

```
stmt =
  conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                       ResultSet.CONCUR_READ_ONLY);
query = "SELECT * FROM Employees WHERE employee_id = " + id;
System.out.println("\nExecuting: " + query);
```

**6.** Run the query and use a ResultSet object to contain the result.

```
rset = stmt.executeQuery(query);
```

**7.** Use the result set returned in `rset` to populate the fields of the employee bean using the `set` methods of the bean.

```
while (rset.next()) {
  selectedEmp.setEmployeeId(new Integer(rset.getInt("employee_id")));
  selectedEmp.setFirstName(rset.getString("first_name"));
  selectedEmp.setLastName(rset.getString("last_name"));
  selectedEmp.setEmail(rset.getString("email"));
  selectedEmp.setPhoneNumber(rset.getString("phone_number"));
  selectedEmp.setHireDate(rset.getDate("hire_date"));
  selectedEmp.setSalary(new Double(rset.getDouble("salary")));
  selectedEmp.setJobId(rset.getString("job_id"));
}
```

**8.** Return the populated object.

```
return selectedEmp;
```

## Creating a Method to Update Employee Data

In the following steps, you will see how to create a method to update employee data in the database:

**1.** Open the `DataHandler` class.

**2.** Declare an `updateEmployee` method as follows:

```
public String updateEmployee(int employee_id, String first_name,
                             String last_name, String email,
                             String phone_number, String salary,
                             String job_id) throws SQLException {

}
```

**3.** Within the body of this method, create an instance of the `Employee` bean, containing details for the selected employee:

```
Employee oldEmployee = findEmployeeById(employee_id);
```

**4.** Connect to the database.

```
getDBConnection();
```

5. Create a `Statement` object and specify the `ResultSet` type as before.

```
stmt =
   conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);
```

6. Create a `StringBuffer` to accumulate details of the SQL `UPDATE` statement that needs to be built:

```
StringBuffer columns = new StringBuffer( 255 );
```

7. For each field in an employee record, check whether the user has changed the value and if so, add relevant code to the `StringBuffer`. For each item added after the first one, add a comma to separate the items. The following code checks if the `first_name` variable changed, and if so, adds details to the SQL in the `StringBuffer` that will be used to update the database:

```
if ( first_name != null &&
     !first_name.equals(oldEmployee.getFirstName() ) )
{
   columns.append( "first_name = '" + first_name + "'" );
}
```

For the `last_name`, before appending the new last name, check to see whether there are already some changes in the `StringBuffer` and if so, append a comma to separate the new change from the previous one. Use the following code:

```
if ( last_name != null &&
     !last_name.equals(oldEmployee.getLastName() ) ) {
   if ( columns.length() > 0 ) {
      columns.append( ", " );
   }
   columns.append( "last_name = '" + last_name + "'" );
}
```

Use the samecode logic to check for changes made to `email`, and `phone_number`.

---

**Note:** Only significant parts of the code are included within this procedure. Example 5–2 contains the complete code for this method.

---

For the `salary` field, obtain a `String` value to add to the `StringBuffer` as follows:

```
if ( salary != null &&
     !salary.equals( oldEmployee.getSalary().toString() ) ) {
   if ( columns.length() > 0 ) {
      columns.append( ", " );
   }
   columns.append( "salary = '" + salary + "'" );
```

8. When the whole set of changes has been assembled, check to see whether there are in fact any changes, that is, whether the `StringBuffer` contains anything. If so, construct a SQL `UPDATE` statement using the information in the `StringBuffer` and execute it. If the StringBuffer does not contain any changes, output a message saying so:

```
if ( columns.length() > 0 )
{
   sqlString = "update Employees SET " + columns.toString() +
            " WHERE employee_id = " + employee_id;
```

```
      System.out.println("\nExecuting: " + sqlString);
      stmt.execute(sqlString);
    }
    else
    {
      System.out.println( "Nothing to do to update Employee Id: " +
                          employee_id);
    }
```

**9.** Return the word "success".

```
    return "success";
```

**10.** Save your work and make the file to check there are no syntax errors.

Example 5–2 contains the complete code for this method.

### Example 5–2   Method for Updating a Database Record

```
public String updateEmployee(int employee_id, String first_name,
                             String last_name, String email,
                             String phone_number, String salary,
                             String job_id) throws SQLException {

  Employee oldEmployee = findEmployeeById(employee_id);
  getDBConnection();
  stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_READ_ONLY);

  StringBuffer columns = new StringBuffer( 255 );
  if ( first_name != null &&
      !first_name.equals( oldEmployee.getFirstName() ) )
  {
    columns.append( "first_name = '" + first_name + "'" );
  }
  if ( last_name != null &&
      !last_name.equals( oldEmployee.getLastName() ) ) {
    if ( columns.length() > 0 ) {
      columns.append( ", " );
    }
    columns.append( "last_name = '" + last_name + "'" );
  }
  if ( email != null &&
      !email.equals( oldEmployee.getEmail() ) ) {
    if ( columns.length() > 0 ) {
      columns.append( ", " );
    }
    columns.append( "email = '" + email + "'" );
  }
  if ( phone_number != null &&
      !phone_number.equals( oldEmployee.getPhoneNumber() ) ) {
    if ( columns.length() > 0 ) {
      columns.append( ", " );
    }
    columns.append( "phone_number = '" + phone_number + "'" );
  }
  if ( salary != null &&
      !salary.equals( oldEmployee.getSalary().toString() ) ) {
    if ( columns.length() > 0 ) {
      columns.append( ", " );
    }
```

```
      columns.append( "salary = '" + salary + "'" );
    }
    if ( job_id != null &&
         !job_id.equals( oldEmployee.getJobId() ) ) {
      if ( columns.length() > 0 ) {
        columns.append( ", " );
      }
      columns.append( "job_id = '" + job_id + "'" );
    }

    if ( columns.length() > 0 )
    {
    sqlString =
      "UPDATE Employees SET " + columns.toString() +
        " WHERE employee_id = " + employee_id;
      System.out.println("\nExecuting: " + sqlString);
      stmt.execute(sqlString);
    }
    else
    {
      System.out.println( "Nothing to do to update Employee Id: " +
                          employee_id);
    }
    return "success";
}
```

## Adding a Link to Navigate to an Update Page

In the following steps, you add a link to each row of the employees table on the
`employees.jsp` page, that users will click to edit that row.

1. Open `employees.jsp` in the Visual Editor.

2. Add an extra column to the table that displays employee details. To do this,
   position the cursor in the last column of the table, right-click and select **Table** from
   the shortcut menu, then select **Insert Rows Or Columns**. In the Insert Rows or
   Columns dialog box, select **Columns** and **After Selection** and click **OK**.

3. This extra column will contain the link that reads Edit for each row. Each of these
   links leads to a separate page where the selected employee record can be edited.
   To do this, double-click the scriptlet that is inside the `Employees` table, to display
   the Scriptlet Properties dialog box.

4. Modify the scriptlet to include a link to the `edit.jsp` page. The modified scriptlet
   should contain the following code:

```
    while (rset.next ())
    {
    out.println("<tr>");
      out.println("<td>" +
      rset.getString("first_name") + "</td><td> " +
      rset.getString("last_name") + "</td><td> " +
      rset.getString("email") + "</td><td> " +
      rset.getString("job_id") + "</td><td>" +
      rset.getString("phone_number") + "</td><td>" +
      rset.getDouble("salary") +
      "</td><td> <a href=\"edit.jsp?empid=" + rset.getInt(1) +
      "\">Edit</a></td>");
      out.println("<tr>");
      }
```

When the edit link is clicked for any employee, this code passes the employee ID to the `edit.jsp` page, which will handle the employee record updates. The `edit.jsp` page will use this to search for the record of that particular employee in the database.

5. Save `employees.jsp`. Figure 5–2 shows `employees.jsp` when it is run and displayed in a browser, illustrating the link users can click to edit employee data.

*Figure 5–2    Link to Edit Employees in employees.jsp*



## Creating a JSP Page to Edit Employee Data

In this section, you will create the `edit.jsp` file that allows users to update an employee record.

1. Create a new JSP page and name it `edit.jsp`. Accept all other defaults.

2. Give the page the same heading as earlier, **AnyCo Corporation: HR Application**, apply the **Heading 2** style to it, and align it to the center of the page.

3. On the next line, type **Edit Employee Record**, with the **Heading 3** style applied. Align this heading to the left of the page.

4. Add the **JDeveloper** style sheet to the page.

5. Add a `jsp:usebean` tag. Enter `empsbean` as the **ID**, and `hr.DataHandler` as the **Class**. Set the **Scope** to `session`, and click **OK**.

6. Position the cursor after the useBean tag and add another `jsp:usebean` tag. This time enter `employee` as the **ID**, browse to select **hr.Employee** as the class, and leave the Scope as page. Click **OK**.

7. Add a **Scriptlet** to the page. The scriptlet code passes the employee ID to the `findEmployeeById` method and retrieves the data inside the `Employee` bean. Enter the following code in the Insert Scriptlet dialog box:

```
Integer employee_id = new Integer(request.getParameter("empid"));
employee = empsbean.findEmployeeById(employee_id.intValue());
```

8. Add a **Form** to the page. In the Insert Form dialog, enter `update_action.jsp` for the **Action** field. You cannot select this page from the drop down list as you have not yet created it.

9. Add a **Table** to the page. Position it inside the Form. Specify a 6-row and 2-column layout, and accept other layout defaults.

10. In the first column, enter the following headings, each on a separate row: `First Name`, `Last Name`, `Email`, `Phone`, `Job`, `Monthly Salary`.

11. Drag a **Hidden Field** component from the HTML Forms page of the Component Palette. Drop it in the second column, adjacent to the First Name heading. In the Insert Hidden Field dialog, enter `employee_id` as the **Name** property and enter `<%= employee.getEmployeeId() %>` as the **Value** property.

12. Drag a **Text Field** component to this column, adjacent to the First Name heading. In the Insert Text Field dialog, enter `first_name` in the **Name** field, and `<%= employee.getFirstName() %>` in the **Value** field. Click **OK**.

13. Drag a second **Text Field** component to this column, adjacent to the Last Name heading. In the Insert Text Field dialog, enter `last_name` in the **Name** field, and `<%= employee.getLastName() %>` in the `Value` field. Click **OK**.

14. In a similar way, add text fields adjacent to each of the remaining column headings, using `email`, `phone_number`, `job_id`, and `salary` as the field names and the corresponding getter method for each field. These are specified in the following table.

15. Add a **Submit** button in the form, below the table. Enter `Update` as its **Value**.

16. Save the application.

The resultant `edit.jsp` page should look similar to the page shown in Figure 5–3.

*Figure 5–3   Creating a JSP Page to Edit Employee Details*
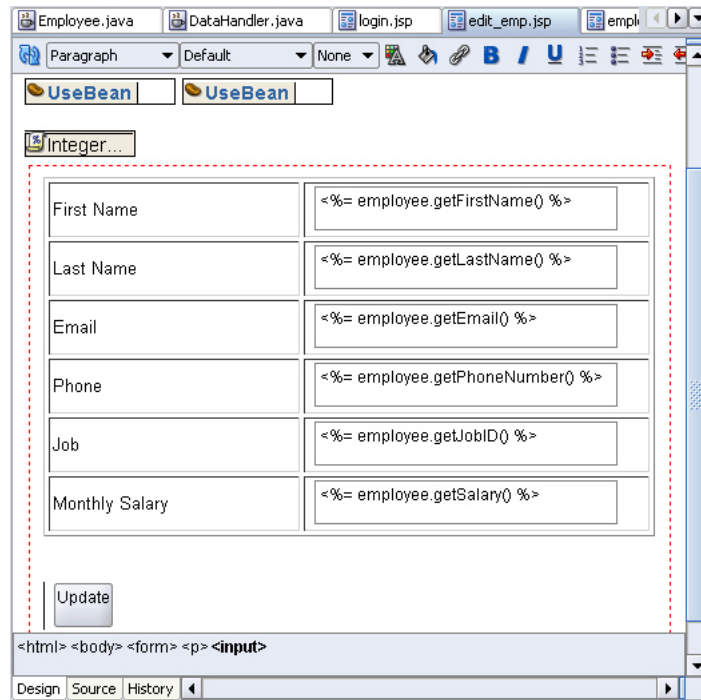


## Creating a JSP Page to Handle an Update Action

In this section, you will see how to create the `update_action.jsp` file. This page processes the form on the `edit.jsp` page that allows users to update an employee record. There are no visual elements on this page, this page is used only to process the `edit.jsp` form and returns control to the `employees.jsp` file.
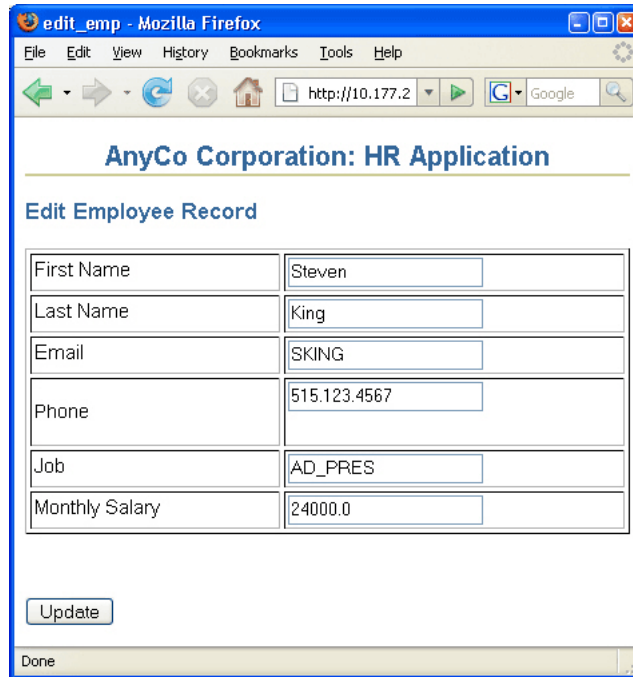
1. Create a new JSP page and call it `update_action.jsp`. Accept all other defaults for the page in the JSP Creation Wizard.

2. Drag a **Page Directive** component from the JSP page of the Component Palette onto the page. In the Insert Page Directive dialog box, browse to import **java.sql.ResultSet**. Click **OK**.

3. Add a **jsp:usebean** tag. Enter `empsbean` as the **ID**, and `hr.DataHandler` as the **Class**. Set the **Scope** to `session`, and click **OK**.

4. Add a **Scriptlet** to the page. Enter the following code into the Insert Scriptlet dialog box:

```
Integer employee_id = new Integer(request.getParameter("employee_id"));
String first_name = request.getParameter("first_name");
String last_name = request.getParameter("last_name");
String email = request.getParameter("email");
String phone_number = request.getParameter("phone_number");
String salary = request.getParameter("salary");
String job_id = request.getParameter("job_id");
empsbean.updateEmployee(employee_id.intValue(), first_name, last_name, email,
phone_number, salary, job_id );
```

5. Drag a **jsp:forward** tag onto the page. In the Insert Forward dialog box, enter `employees.jsp` for the **Page** property.

6. Save your work.

**7.** Run the project and test whether you can edit an employee record. Click **Edit** for any employee on the `employees.jsp` page, and you should be directed to the page shown in Figure 5–4. Modify any of the employee details and check whether the change reflects in the `employees.jsp` page.

*Figure 5–4   Editing Employee Data*



# Inserting an Employee Record

The steps for inserting a new employee record to the Employees table are similar to the process for updating an employee record:

**1.** Create a method to insert a new employee row into the `Employees` table.

**2.** Add a link to the main application page, allowing a user to click to insert a new employee. The link takes the user to an `insert.jsp` with an empty form ready for the user to enter details for the new row.

**3.** Create a JSP page to process the form on the `insert.jsp` page.

**4.** Create a JSP page with form controls for users to enter the values for the new employee.

This section covers the creation of Java application code for inserting new employee data in the following subsections:

- Creating a Method to Insert Data
- Adding a Link to Navigate to an Insert Page
- Creating a JSP Page to Handle an Insert Action
- Creating a JSP Page to Enter New Data

## Creating a Method to Insert Data

In the following steps, you will create a method for inserting a new employee record.

1. Open `DataHandler.java` in the Java Source Editor.

2. Declare a method to add a new employee record.

   ```
   public String addEmployee(String first_name,
     String last_name, String email,
     String phone_number, String job_id, int salary) throws SQLException {

   }
   ```

3. Add a line to connect to the database.

   ```
   getDBConnection();
   ```

4. Create a `Statement` object, define a `ResultSet` type as before, and formulate the SQL statement.

   ```
   stmt =
     conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                          ResultSet.CONCUR_READ_ONLY);
   sqlString =
     "INSERT INTO Employees VALUES (EMPLOYEES_SEQ.nextval, '" +
     first_name + "','" +
     last_name + "','" +
     email + "','" +
     phone_number + "'," +
     "SYSDATE, '" +
     job_id + "', " +
     salary + ",.30,100,80)";
   ```

   > **Note:** The last three columns (`Commission`, `ManagerId`, and `DepartmentId`) contain hard-coded values for the sample application.

5. Add a trace message, and then run the SQL statement.

6. Return a message that says "success" if the insertion was successful.

7. Make the file to check for syntax errors.

Example 5–3 shows the code for the `addEmployee()` method.

***Example 5–3   Method for Adding a New Employee Record***

```
public String addEmployee(String first_name,
  String last_name, String email,
  String phone_number, String job_id, int salary) throws SQLException {
  getDBConnection();
  stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_READ_ONLY);
  sqlString =
    "INSERT INTO Employees VALUES (EMPLOYEES_SEQ.nextval, '" +
     first_name + "','" +
    last_name + "','" +
    email + "','" +
    phone_number + "'," +
    "SYSDATE, '" +
    job_id + "', " +
    salary + ",.30,100,80)";

  System.out.println("\nInserting: " + sqlString);
  stmt.execute(sqlString);
```

```
                     return "success";
                 }
```

## Adding a Link to Navigate to an Insert Page

In these steps, you add a link to the header row of the employees table that users can click to add a new employee.

1. Open employees.jsp in the Visual Editor.

2. Drag a **Hyper Link** component from the HTML Common page of the Component Palette into the empty column header cell at the end of the header row. In the Insert HyperLink dialog box, enter insert.jsp in the **HyperLink** field, and Insert Employee in the **Text** field. You cannot browse to find insert.jsp as you have not yet created it. Click **OK**.

3. Save employees.jsp.

## Creating a JSP Page to Enter New Data

In these steps, you create the insert.jsp page, which allows users to enter details of a new employee record.

1. Create a new JSP page and call it insert.jsp.

2. Give the page the same heading as before, **AnyCo Corporation: HR Application**, and format it as **Heading 2**, and center it.

3. On the next line enter **Insert Employee Record**, and apply the **Heading 3** format. Align this heading to the left of the page.

4. Add the **JDeveloper** stylesheet to the page.

5. Add a **Form**. In the Insert Form dialog box, enter insert_action.jsp for the **Action** property, and click **OK**.

6. Add a **Table** inside the **Form**. Specify that you want 6 rows and 2 columns and accept all other layout defaults.

7. In the first column, enter the following headings, each on a separate row: **First Name**, **Last Name**, **Email**, **Phone**, **Job**, **Monthly Salary**.

8. Drag and drop a **Text Field** into the column to the right of the **First Name** header. In the Insert Field dialog box, type first_name in the **Name** property.

9. Drag a **Text** Field next to each of the **Last Name**, **Email**, **Phone**, and **Monthly Salary** headers. Specify the values for each of these text fields for the **Name** property in the Insert Field dialog box. The values are indicated in the following table:

| Text Field For | Set the Name Property To |
| --- | --- |
| Last Name | last_name |
| Email | email |
| Phone | phone_number |
| Monthly Salary | salary |

This procedure is different for the Job row.

10. Drag a **Combo Box** component from the HTML Forms page of the Component Palette to the column next to the **Job** heading.

11. In the Insert Select dialog box, enter `job_id` as the name, and `1` as the size. Click **Add**. Click in the **Value** field and enter `SA_REP`, and in the **Caption** field, enter `Sales Representative`. Click **Add** to add each of the following job titles, then click **OK**.

| Value | Caption |
|---|---|
| HR_REP | HR Representative |
| PR_REP | PR Representative |
| MK_MAN | Marketing Manager |
| SA_MAN | Sales Manager |
| FI_MAN | Finance Manager |
| IT_PROG | Software Developer |
| AD_VIP | Vice President |

12. Drag a **Submit** button to the Form below the table. In the Insert Submit Button dialog box, enter `Add Employee` for the **Value** property.

13. Save your work.

Figure 5–5 shows the `insert.jsp` page in the Visual Editor.

**Figure 5–5    Form to Insert Employee Data**
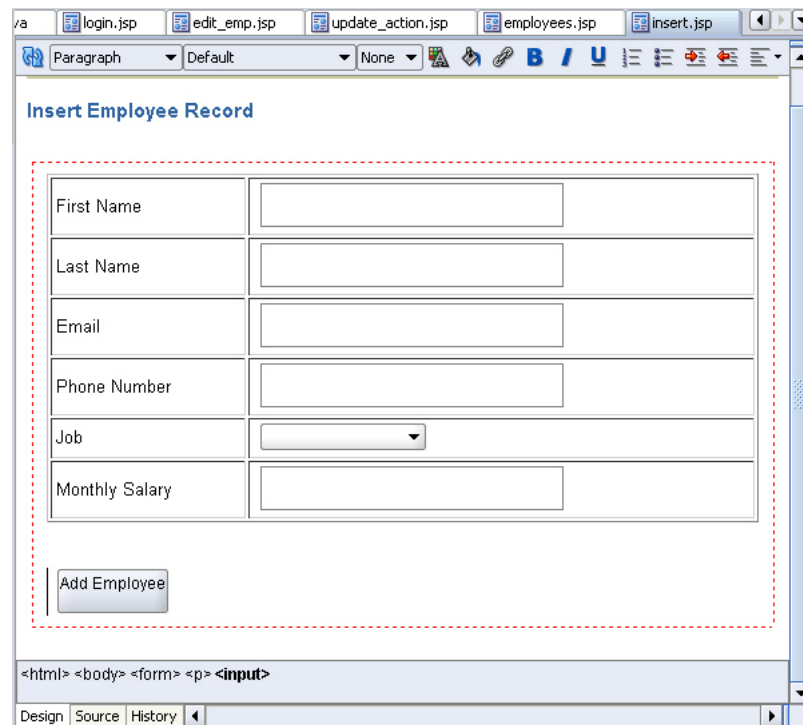
## Creating a JSP Page to Handle an Insert Action

In these steps, you create the `insert_action.jsp` page. This is a page that processes the form input from `insert.jsp`, which is the page on which users enter a new employee record. There are no visual elements on this page, and it is only used to process the `insert.jsp` form and return control to the `employees.jsp` file.

1. Create a JSP page as before. Call it `insert_action.jsp`.

2. Add a **jsp:usebean** tag. As before, enter `empsbean` as the **ID**, and `hr.DataHandler` as the **Class**. Set the **Scope** to `session`, and click **OK**.

3. Position the cursor after the useBean tag and add a **Scriptlet** to the page. Enter the following code into the Insert Scriptlet dialog box:

```
String first_name = request.getParameter("first_name");
String last_name = request.getParameter("last_name");
String email = request.getParameter("email");
String phone_number = request.getParameter("phone_number");
String job_id = request.getParameter("job_id");
Integer salary = new Integer(request.getParameter("salary"));

empsbean.addEmployee(first_name, last_name, email, phone_number, job_id,
salary.intValue());
```

4. Drag a **jsp:forward** tag onto the page. In the Insert Forward dialog box, enter `employees.jsp`.

5. Save your work.

6. Run the **View** project to test whether you can insert a new employee record.

To insert an employee, click `Insert Employee` on the `employees.jsp` page shown in Figure 5–6.

**Figure 5–6    Inserting New Employee Data**
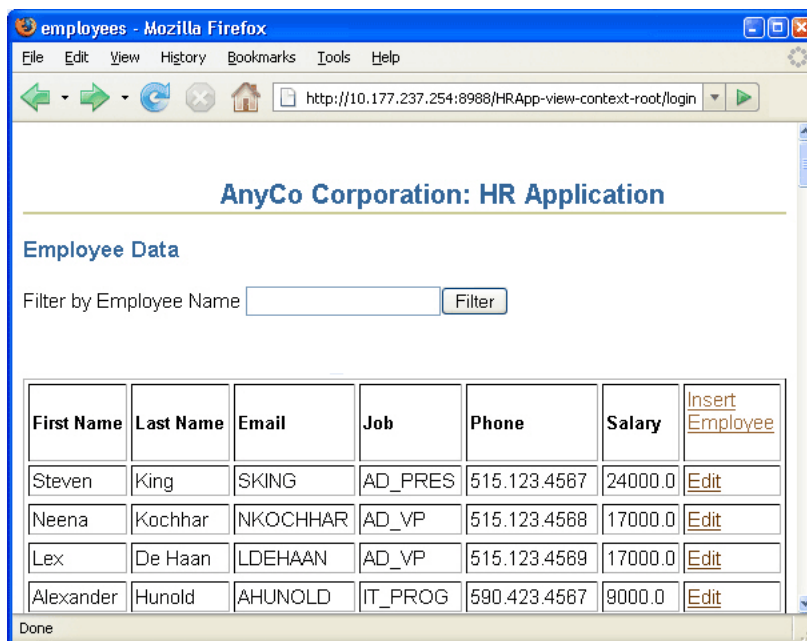


Figure 5–7 shows the page where you can insert new employee data with some data filled in, and the list of jobs being used to select a job.

*Figure 5–7   Inserting Employee Data*



## Deleting an Employee Record

The steps for deleting a record are similar to those for editing and inserting a record:

1. Use the method created in Creating a Method to Identify an Employee Record to identify a particular employee row. This is used to identify the row to be deleted.

2. Create a method that deletes an employee record from the database.

3. Add a link to the main application page for each row, allowing a user to click to delete the employee in that row. The link takes the user to a `delete_action.jsp`, with the ID of the employee whose record is to be deleted.

4. To delete the employee from the database, create a JSP page to call the delete method created in Step 2.

This section discusses the following tasks related to deleting employee data:

- Creating a Method for Deleting Data

- Adding a Link to Delete an Employee

- Creating a JSP Page to Handle a Delete Action

### Creating a Method for Deleting Data

The method created in the following steps is used to delete employee records by ID:

1. Open `DataHandler.java` in the Java Source Editor.

2. Declare a new method that identifies the employee record to be deleted:

```
public String deleteEmployeeById(int id) throws SQLException {

}
```

3. Connect to the database as before.

```
getDBConnection();
```

4. Create a `Statement` object, define a `ResultSet` type as before, and formulate the SQL statement. Add a trace message to assist with debugging.

```
stmt =
  conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                       ResultSet.CONCUR_READ_ONLY);
sqlString = "DELETE FROM Employees WHERE employee_id = " + id;
System.out.println("\nExecuting: " + sqlString);
```

5. Run the SQL statement.

```
stmt.execute(sqlString);
```

6. If the SQL statement runs without any errors, return the word, `Success`.

```
return "success";
```

Example 5–4 shows the code for the `deleteEmployeeById()` method.

**Example 5–4   Method for Deleting an Employee Record**

```
public String deleteEmployeeById(int id) throws SQLException {
getDBConnection();
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_READ_ONLY);
    sqlString = "DELETE FROM Employees WHERE employee_id = " + id;
    System.out.println("\nExecuting: " + sqlString);
    stmt.execute(sqlString);
    return "success";
}
```

## Adding a Link to Delete an Employee

In the following instructions, you add a link to each row of the employees table on the `employees.jsp` page. Clicking on that link will delete all employee data for that row.

1. Open `employees.jsp` in the Visual Editor.

2. In the column you created to contain the Edit link, add another link for deleting the row. To do this, double-click the scriptlet that is inside the `Employees` table, to display the Scriptlet Properties dialog box.

3. Modify the scriptlet to include a link to a `delete_action.jsp` page. The modified scriptlet should contain the following code:

```
while (rset.next ())
{
out.println("<tr>");
  out.println("<td>" +
  rset.getString("first_name") + "</td><td> " +
  rset.getString("last_name") + "</td><td> " +
  rset.getString("email") + "</td><td> " +
  rset.getString("job_id") + "</td><td>" +
  rset.getString("phone_number") + "</td><td>" +
  rset.getDouble("salary") +
  "</td><td> <a href=\"edit.jsp?empid=" + rset.getInt(1) +
  "\">Edit</a>   <a href=\"delete_action.jsp?empid=" +
  rset.getInt(1) + "\">Delete</a></td>");
out.println("<tr>");
}
```

> **4.** Save `employees.jsp`.

## Creating a JSP Page to Handle a Delete Action

In the following steps, you create the `delete_action.jsp` page, which is a page that only processes the delete operation. There are no visual elements on this page.

**1.** Create a JSP page and call it `delete_action.jsp`.

**2.** Add a **jsp:usebean** tag. As before, enter `empsbean` as the **ID**, and `hr.DataHandler` as the **Class**. Set the **Scope** to `session`, and click **OK**.

**3.** Add a **Scriptlet** to the page. Enter the following code into the Insert Scriptlet dialog box:

```
Integer employee_id =
 new Integer(request.getParameter("empid"));
empsbean.deleteEmployeeById(employee_id.intValue());
```

**4.** Drag Forward from the Component Palette to add a **jsp:forward** tag to the page. In the Insert Forward dialog box, enter **employees.jsp**.

**5.** Save your work.

**6.** Run the project and try deleting an employee. Figure 5–8 shows the links for deleting employee records from the `employees.jsp`.

*Figure 5–8    Link for Deleting an Employee from employees.jsp*
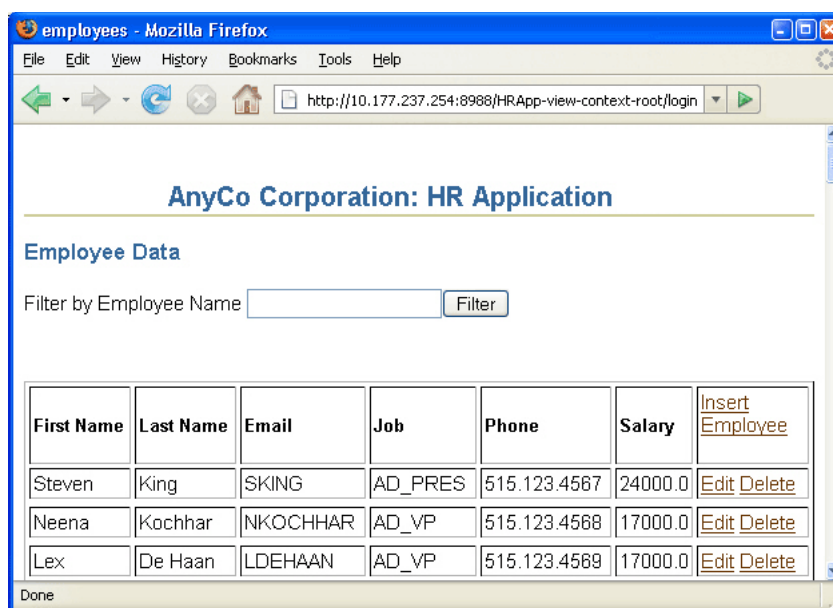


If you click **Delete** for any of the employee records, then that employee record will be deleted.

# Exception Handling

A `SQLException` object instance provides information on a database access error or other errors. Each `SQLException` instance provides many types of information,

including a string describing the error, which is used as the Java Exception message, available via the `getMessage` method.

The sample application uses `try` and `catch` blocks, which are the Java mechanism for handling exceptions. With Java, if a method throws an exception, there needs to be a mechanism to handle it. Generally, a `catch` block catches the exception and specifies the course of action in the event of an exception, which could simply be to display the message.

Each JDBC method throws a `SQLException` if a database access error occurs. For this reason, any method in an application that executes such a method must handle the exception.

All the methods in the sample application include code for handling exceptions. For example, the `getDBConnection`, which is used to get a connection to the database, throws SQLException, as does the `getAllEmployees` method as follows:

```
public ResultSet getAllEmployees() throws SQLException {
}
```

For an example of code used to catch and handle SQLExceptions, refer to the code in the `authenticateUser` method in the `DataHandler.java` class. In this example, a `try` block contains the code for the work to be done to authenticate a user, and a `catch` block handles the case where the authentication fails.

The following sections describe how to add code to the sample application to catch and handle SQLExceptions.

## Adding Exception Handling to Java Methods

To handle SQL exceptions in the methods in the sample application, do the following:

1.  Ensure that the method throws `SQLException`. For example, the method:

    ```
    public ResultSet getAllEmployees() throws SQLException
    ```

2.  Use `try` and `catch` blocks to catch any SQLExceptions. For example, in the `getAllEmployees` method, enclose your existing code in a `try` block, and add a `catch` block as follows:

    ```
    public ResultSet getAllEmployees() throws SQLException {
      try {
        getDBConnection();
        stmt =
          conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                               ResultSet.CONCUR_READ_ONLY);
        sqlString = "SELECT * FROM Employees order by employee_id";
        System.out.println("\nExecuting: " + sqlString);
        rset = stmt.executeQuery(sqlString);
      }
      catch (SQLException e) {
        e.printStackTrace();
      }
      return rset;
    }
    ```

3.  As another example, the `deleteEmployee` method rewritten to use `try` and `catch` blocks would return "success" only if the method was successful, that is, the `return` statement is enclosed in the `try` block. The code could be as follows:

    ```
    public String deleteEmployeeById(int id) throws SQLException {
    ```

```
            try {
              getDBConnection();
              stmt =
                conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
              sqlString = "delete FROM Employees where employee_id = " + id;
              System.out.println("\nExecuting: " + sqlString);

              stmt.execute(sqlString);
              return "success";
            }
            catch (SQLException e) {
              e.printStackTrace();
            }
          }
```

## Creating a Method for Handling Any SQLException

As a refinement to the code for the sample application, you can create a method that can be used in any method that might throw a SQLException, to handle the exception. As an example, the following method could be called in the catch block of any of the methods in the sample application. This method cycles through all the exceptions that have accumulated, printing a stack trace for each.

***Example 5–5    Adding a Method to Handle Any SQLException in the Application***

```
public void logException( SQLException ex )
{
  while ( ex != null ) {
    ex.printStackTrace();
    ex = ex.getNextException();
  }
}
```

In addition, in the catch block, you can return text that explains why the method has failed. The catch block of a method could therefore be written as follows:

```
catch ( SQLException ex )  {
  logException( ex );
  return "failure";
}
```

To add this feature to your application:

1.  In the DataHandler.java, add a logException method.

2.  Edit each of the methods to include try and catch blocks.

3.  In the catch block of each method, run the logException method.

4.  For methods that have a return value of String, include a return statement to return a message indicating that the method has failed such as:

    ```
    return "failure";
    ```

## Navigation in the Sample Application

The web.xml file is the deployment descriptor file for a web application. One section of the web.xml file can be used for defining a start page for the application, for example:

```
<web-app>
...
  <welcome-file>
    myWelcomeFile.jsp
  </welcome-file>
...
</web-app>
```

If you do not define a welcome page in your web.xml file, generally a file with the name index, with extension .html, .htm, or .jsp if there is one, is used as the starting page. With JDeveloper, you can define which page is to be the default run target for the application, that is, the page of the application that is displayed first, by defining it in the properties of the project.

Once the application has started, and the start page has been displayed, navigation through the application is achieved using the following scheme:

- Links, in the form of HTML anchor tags, define a target for the link, usually identifying another JSP page to which to navigate, and some text for the link.

- HTML submit buttons, are used to submit forms on the pages, such as forms for entering new or changed data.

- jsp:forward tags, which are executed on JSP pages that handle queries and forms, to forward to either the same JSP page again, or another JSP page.

## Creating a Starting Page for an Application

In the following steps, you create the index.jsp page, which will be the default starting page for the application. The page does not include any display elements, and simply forwards the user to the application login page, login.jsp. To do this you use the jsp:forward tag. A jsp:forward tag runs on JSP pages that handle queries and forms, to forward to either the same JSP page again, or another JSP page.

1. Create a new JSP page and call it index.jsp.

2. For the sample application, we will not add any text to this page. From the JSP page of the Component Palette, drag **Forward** to include a jsp:forward tag in the page.

3. In the Insert Forward dialog box for the forward tag, enter login.jsp as the **Page**.

You can now specify this new page as the default target for the application as follows:

1. In the Application Navigator, right-click the View project and choose Project Properties.

2. In the displayed tree, select **Run/Debug**. In the Run/Debug area, ensure that Use Project Settings is selected, and in the Run Configurations area, ensure that Default Configurations is selected. Click **Edit**.

3. In the Edit Launch Settings dialog box, select **Launch Settings**. In the Launch Settings area on the right, click **Browse** next to the Default Run Target field and navigate to find the new index.jsp page you just created and click **OK**. Then click **OK** again to close the dialog box.

You can now run your application by right-clicking in the **View** project and select **Run** from the shortcut menu. The application runs and runs index.jsp, which has been set as the default launch target for the application. The index.jsp forwards you directly to the login page, login.jsp, which is displayed in your browser.

# 6

# Enhancing the Application: Advanced JDBC Features

This chapter describes additional functionality that you can use in your Java application. Some of these features have not been implemented in the sample application, while some features are enhancements you can use in your code to improve performance.

This chapter includes the following sections:

- Using Dynamic SQL
- Calling Stored Procedures
- Using Cursor Variables

## Using Dynamic SQL

Dynamic SQL, or generating SQL statements on the fly, is a constant need in a production environment. Very often, and especially in the matter of updates to be performed on a database, the final query is not known until run time.

For scenarios where many similar queries with differing update values must be run on the database, you can use the `OraclePreparedStatement` object, which extends the `Statement` object. This is done by substituting the literal update values with bind variables. You can also use stored PL/SQL functions on the database by calling stored procedures through the `OracleCallableStatement` object.

This section discusses the following topics:

- Using OraclePreparedStatement
- Using OracleCallableStatement
- Using Bind Variables

## Using OraclePreparedStatement

To run static SQL queries on the database, you use the `Statement` object. However, to run multiple similar queries or perform multiple updates that affect many columns in the database, it is not feasible to hard-code each query in your application.

You can use `OraclePreparedStatement` when you run the same SQL statement multiple times. Consider a query like the following:

```
SELECT * FROM Employees WHERE ID=xyz;
```

Every time the value of xyz in this query changes, the SQL statement needs to be compiled again.

If you use OraclePreparedStatement functionality, the SQL statement you want to run is precompiled and stored in a PreparedStatement object, and you can run it as many times as required without compiling it every time it is run. If the data in the statement changes, you can use bind variables as placeholders for the data and then provide literal values at run time.

Consider the following example of using OraclePreparedStatement:

### Example 6–1   Creating a PreparedStatement

```
OraclePreparedStatement pstmt = conn.prepareStatement("UPDATE Employees
                            SET salary = ? WHERE ID = ?");
   pstmt.setBigDecimal(1, 153833.00)
   pstmt.setInt(2, 110592)
```

The advantages of using the OraclePreparedStatement interface include:

- You can batch updates by using the same PreparedStatement object

- You can improve performance because the SQL statement that is run many times is compiled only the first time it is run.

- You can use bind variables to make the code simpler and reusable.

## Using OracleCallableStatement

You can access stored procedures on databases using the OracleCallableStatement interface. This interface extends the OraclePreparedStatement interface. The OracleCallableStatement interface consists of standard JDBC escape syntax to call stored procedures. You may use this with or without a result parameter. However, if you do use a result parameter, it must be registered as an OUT parameter. Other parameters that you use with this interface can be either IN, OUT, or both.

These parameters are set by using accessor methods inherited from the OraclePreparedStatement interface. IN parameters are set by using the set*XXX* methods and OUT parameters are retrieved by using the get*XXX* methods, *XXX* being the Java data type of the parameter.

A CallableStatement can also return multiple ResultSet objects.

As an example, you can create an OracleCallableStatement to call the stored procedure called foo, as follows:

### Example 6–2   Creating a CallableStatement

```
OracleCallableStatement cs = (OracleCallableStatement)
conn.prepareCall("{call foo(?)}");
```

You can pass the string bar to this procedure in one of the following two ways:

```
cs.setString(1,"bar"); // JDBC standard
// or...
cs.setString("myparameter","bar"); // Oracle extension
```

## Using Bind Variables

Bind variables are variable substitutes for literals in a SQL statement. They are used in conjunction with OraclePreparedStatement and OracleCallableStatement to specify

parameter values that are used to build the SQL statement. Using bind variables has remarkable performance advantages in a production environment.

For PL/SQL blocks or stored procedure calls, you can use the following qualifiers to differentiate between input and output variables: `IN`, `OUT`, and `IN OUT`. Input variable values are set by using set*XXX* methods and `OUT` variable values can be retrieved by using get*XXX* methods, where *XXX* is the Java data type of the values. This depends on the SQL data types of the columns that you are accessing in the database.

## Calling Stored Procedures

Oracle Java Database Connectivity (JDBC) drivers support the processing of PL/SQL stored procedures and anonymous blocks. They support Oracle PL/SQL block syntax and most of SQL92 escape syntax. The following PL/SQL calls would work with any Oracle JDBC driver:

***Example 6–3   Calling Stored Procedures***

```
// SQL92 syntax
CallableStatement cs1 = conn.prepareCall
                        ( "{call proc (?,?)}" ) ; // stored proc
CallableStatement cs2 = conn.prepareCall
                        ( "{? = call func (?,?)}" ) ; // stored func

// Oracle PL/SQL block syntax
CallableStatement cs3 = conn.prepareCall
                        ( "begin proc (?,?); end;" ) ; // stored proc
CallableStatement cs4 = conn.prepareCall
                        ( "begin ? := func(?,?); end;" ) ; // stored func
```

As an example of using the Oracle syntax, here is a PL/SQL code snippet that creates a stored function. The PL/SQL function gets a character sequence and concatenates a suffix to it:

***Example 6–4   Creating a Stored Function***

```
create or replace function foo (val1 char)
return char as
begin
return val1 || 'suffix';
end;
```

You can call this stored function in a Java program as follows:

***Example 6–5   Calling a Stored Function in Java***

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@<hoststring>");
ods.setUser("hr");
ods.setPassword("hr");
Connection conn = ods.getConnection();
CallableStatement cs = conn.prepareCall ("begin ? := foo(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, "aa");
cs.executeUpdate();
String result = cs.getString(1);
```
The following sections describe how you can use stored procedures in the sample application in this guide:

- Creating a PL/SQL Stored Procedure in JDeveloper
- Creating a Method to Use the Stored Procedure
- Allowing Users to Choose the Stored Procedure
- Calling the Stored Procedure from the Application

## Creating a PL/SQL Stored Procedure in JDeveloper

JDeveloper allows you to create stored procedures in the database through the Connection Navigator. In these steps, you create a stored procedure that can be used as an alternative way of inserting an employee record in the sample application.

1. Select the **Connections** tab to view the Connection Navigator.

2. Expand the database connection node (by default called DBConnection1), and the **HR** node to see the objects in the HR database.

3. Right-click **Procedures**, and select **New PL/SQL Procedure**.

4. In the Create PL/SQL Procedure dialog, enter insert_employee as the object name. Click **OK**.

   The skeleton code for the procedure is displayed in the Source Editor.

5. After the procedure name, enter the following lines of code:

```
PROCEDURE    "INSERT_EMPLOYEE" (p_first_name  employees.first_name%type,
  p_last_name    employees.last_name%type,
  p_email        employees.email%type,
  p_phone_number employees.phone_number%type,
  p_job_id       employees.job_id%type,
  p_salary       employees.salary%type
)
```

6. After the BEGIN statement, replace the line that reads NULL with the following:

```
INSERT INTO Employees VALUES (EMPLOYEES_SEQ.nextval, p_first_name ,
  p_last_name , p_email , p_phone_number, SYSDATE, p_job_id,
  p_salary,.30,100,80);
```

   You can see that the statement uses the same hard-coded values that are used for the last three columns in the addEmployee method in the DataHandler.java class.

7. Add the procedure name in the END statement:

```
END insert_employee;
```

8. Save the file, and check whether there are any compilation errors.

The complete code for the stored procedure is shown in Example 6–6.

**Example 6–6   Creating a PL/SQL Stored Procedure to Insert Employee Data**

```
PROCEDURE    "INSERT_EMPLOYEE" (p_first_name  employees.first_name%type,
  p_last_name    employees.last_name%type,
  p_email        employees.email%type,
  p_phone_number employees.phone_number%type,
  p_job_id       employees.job_id%type,
  p_salary       employees.salary%type
)
AS
BEGIN
  INSERT INTO Employees VALUES (EMPLOYEES_SEQ.nextval, p_first_name ,
```

```
      p_last_name , p_email , p_phone_number, SYSDATE, p_job_id,
    p_salary,.30,100,80);
END insert_employee;
```

## Creating a Method to Use the Stored Procedure

In these steps, you add a method to the DataHandler.java class that can be used as an alternative to the addEmployee method. The new method you add here makes use of the insert_employee stored procedure.

1.  Select the **Applications** tab to display the Application Navigator.

2.  If the DataHandler.java file is not already open in the Java Source Editor, double-click it to open it.

3.  Import the CallableStatement interface as follows:

    ```
    import java.sql.CallableStatement;
    ```

4.  After the addEmployee method, add the declaration for the addEmployeeSP method.

    ```
    public String addEmployeeSP(String first_name, String last_name,
      String email, String phone_number, String job_id,
      int salary) throws SQLException {
    }
    ```

    The method signature is the same as that for addEmployee.

5.  Inside the method, add a try block, and inside that, connect to the database.

    ```
    try {
      getDBConnection();
    }
    ```

6.  In addition, inside the try block, create the SQL string:

    ```
    sqlString = "begin hr.insert_employee(?,?,?,?,?,?); end;";
    ```

    The question marks (?) in the statement are bind variables, acting as placeholders for the values of first_name, last_name, and so on expected by the stored procedure.

7.  Create the CallableStatement:

    ```
    CallableStatement callstmt = conn.prepareCall(sqlString);
    ```

8.  Set the IN parameters:

    ```
      callstmt.setString(1, first_name);
      callstmt.setString(2, last_name);
      callstmt.setString(3, email);
      callstmt.setString(4, phone_number);
      callstmt.setString(5, job_id);
      callstmt.setInt(6, salary);
    ```

9.  Add a trace message, and run the callable statement.

    ```
      System.out.println("\nInserting with stored procedure: " +
                          sqlString);
      callstmt.execute();
    ```

10. Add a return message:

```
                                 return "success";
```

**11.** After the `try` block, add a `catch` block to trap any errors. Call the `logException` created in Example 5–5.

```
catch ( SQLException ex ) {
  System.out.println("Possible source of error: Make sure you have created the
stored procedure");
  logException( ex );
  return "failure";
}
```

**12.** Save `DataHandler.java`.

The complete method is shown in Example 6–7.

> **Note:** If you have not added the `logException()` method (see Example 5–5), JDeveloper will indicate an error by showing a red curly line under `logException(ex)`. This method must be present in the `DataHandler.java` class before you proceed with compiling the file.

**Example 6–7   Using PL/SQL Stored Procedures in Java**

```
public String addEmployeeSP(String first_name, String last_name,
  String email, String phone_number, String job_id,
  int salary) throws SQLException {

  try {
    getDBConnection();
    sqlString = "begin hr.insert_employee(?,?,?,?,?,?); end;";
    CallableStatement callstmt = conn.prepareCall(sqlString);
    callstmt.setString(1, first_name);
    callstmt.setString(2, last_name);
    callstmt.setString(3, email);
    callstmt.setString(4, phone_number);
    callstmt.setString(5, job_id);
    callstmt.setInt(6, salary);
    System.out.println("\nInserting with stored procedure: " +
                       sqlString);

    callstmt.execute();
    return "success";
  }
  catch ( SQLException ex ) {
    System.out.println("Possible source of error: Make sure you have created the
stored procedure");
    logException( ex );
    return "failure";
  }
}
```

## Allowing Users to Choose the Stored Procedure

The steps in this section add a radio button group to the `insert.jsp` page, which allows a user to choose between inserting an employee record using the stored procedure, or by using a SQL query in Java code.

**1.** Open `insert.jsp` in the Visual Editor, if it is not already open.

2. Create a new line after the Insert Employee Record heading. With the cursor on this new line, drag **UseBean** from the JSP page of the Component Palette to add a `jsp:useBean` tag to the page. Enter `empsbean` as the ID, browse to select `hr.DataHandler` as the **Class**, and set the **Scope** to `session`. With the UseBean still selected on the page, set the style of this line to `None` instead of Heading 3.

3. Drag a **Radio Button** component from the HTML Forms page of the Component Palette onto the page inside the form above the table. In the Insert Radio Button dialog, enter `useSP` as the **Name**, `false` as the **Value**, and select **Checked**. Click **OK**.

4. In the Visual Editor, position the cursor to the right of the button, and enter text to describe the purpose of the button, for example, 'Use only JDBC to insert a new record'.

5. Press Enter at the end of the current line to create a new line.

6. Drag a second **Radio Button** below the first one. In the Insert Radio Button dialog, use `useSP` as the **Name**, `true` as the **Value**, and ensure that the Checked checkbox is not selected.

7. In the Visual Editor, position the cursor directly to the right of the button, and enter text to describe the purpose of the button, for example, 'Use stored procedure called via JDBC to insert a record'.

8. Save the page.

Figure 6–1 shows `insert.jsp` with the radio button that provides the option to use a stored procedure.

*Figure 6–1   Adding a Link to Provide the Stored Procedure Option*
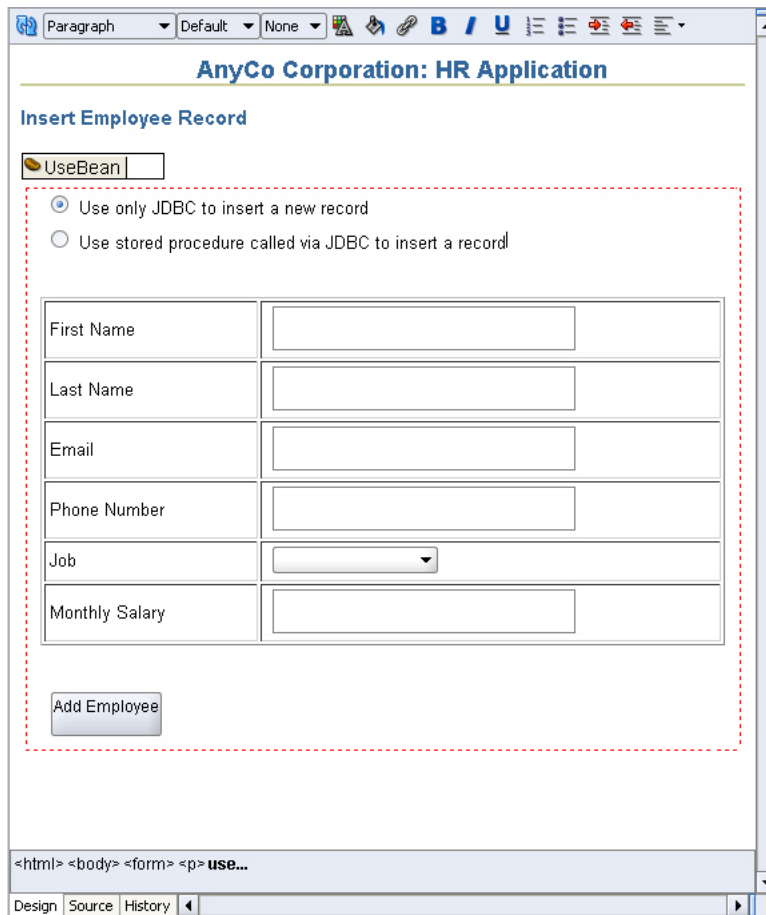


## Calling the Stored Procedure from the Application

The steps in this section modify the `insert_action.jsp` file, which processes the form on the `insert.jsp` page, to use the radio button selection and select the appropriate method for inserting a new employee record.

1. Open `insert_action.jsp` in the Visual Editor, if it is not already open.

2. Double-click the scriptlet to invoke the Scriptlet Properties dialog box and add a new variable after the salary variable, as follows:

   ```
   String useSPFlag = request.getParameter("useSP");
   ```

3. Below that, still in the Scriptlet Properties dialog box, replace the existing `empsbean.addEmployee` line with the following lines of code to select the `addEmployeeSP` method or the pure JDBC `addEmployee` method to insert the record.

   ```
   if ( useSPFlag.equalsIgnoreCase("true"))
     empsbean.addEmployeeSP(first_name, last_name, email,
     phone_number, job_id, salary.intValue());
   // otherwise use pure JDBC insert
   else
     empsbean.addEmployee(first_name, last_name, email,
     phone_number, job_id, salary.intValue());
   ```

4. Save `insert_action.jsp`.

You can now run the application and use the radio buttons on the insert page to choose how you want to insert the new employee record. In a browser, the page will appear as shown in Figure 6–2.

*Figure 6–2   Using Stored Procedures to Enter Records*



## Using Cursor Variables

Oracle JDBC drivers support cursor variables with the REF CURSOR types, which are not a part of the JDBC standard. REF CURSOR types are supported as JDBC result sets.

A cursor variable holds the memory location of a query work area, rather than the contents of the area. Declaring a cursor variable creates a pointer. In SQL, a pointer has the data type REF *x*, where REF is short for REFERENCE and *x* represents the entity being referenced. A REF CURSOR, then, identifies a reference to a cursor variable. Because many cursor variables might exist to point to many work areas, REF CURSOR can be thought of as a category or data type specifier that identifies many different types of cursor variables. A REF CURSOR essentially encapsulates the results of a query.

Oracle does not return ResultSets. To access data returned by a query, you use CURSORS and REF CURSORS. CURSORS contain query results and metadata. A REF CURSOR (or CURSOR variable) data type contains a reference to a cursor. It can be passed between the RDBMS and the client, or between PL/SQL and Java in the database. It can also be returned from a query or a stored procedure.

> **Note:**   REF CURSOR instances are not scrollable.

This section contains the following subsections:

- Oracle REF CURSOR Type Category
- Accessing REF CURSOR Data
- Using REF CURSOR in the Sample Application

## Oracle REF CURSOR Type Category

To create a cursor variable, begin by identifying a type that belongs to the REF CURSOR category. For example:

```
dept_cv DeptCursorTyp
...
```

Then, create the cursor variable by declaring it to be of the type DeptCursorTyp:

### Example 6–8    Declaring a REF CURSOR Type

```
DECLARE TYPE DeptCursorTyp IS REF CURSOR
```

REF CURSOR, then, is a category of data types, rather than a particular data type. Stored procedures can return cursor variables of the REF CURSOR category. This output is equivalent to a database cursor or a JDBC result set.

## Accessing REF CURSOR Data

In Java, a REF CURSOR is materialized as a ResultSet object and can be accessed as follows:

### Example 6–9    Accessing REF Cursor Data in Java

```
import oracle.jdbc.*;
...
CallableStatement cstmt;
ResultSet cursor;

// Use a PL/SQL block to open the cursor
cstmt = conn.prepareCall
        ("begin open ? for select ename from emp; end;");

cstmt.registerOutParameter(1, OracleTypes.CURSOR);
cstmt.execute();
cursor = ((OracleCallableStatement)cstmt).getCursor(1);

// Use the cursor like a normal ResultSet
while (cursor.next ())
    {System.out.println (cursor.getString(1));}
```

In the preceding example:

1. A CallableStatement object is created by using the prepareCall method of the connection class.

2. The callable statement implements a PL/SQL procedure that returns a REF CURSOR.

3. As always, the output parameter of the callable statement must be registered to define its type. Use the type code OracleTypes.CURSOR for a REF CURSOR.

4. The callable statement is run, returning the REF CURSOR.

5. The `CallableStatement` object is cast to `OracleCallableStatement` to use the `getCursor` method, which is an Oracle extension to the standard JDBC application programming interface (API), and returns the `REF CURSOR` into a `ResultSet` object.

## Using REF CURSOR in the Sample Application

In the following sections, you enhance the sample application to display a dynamically-generated list of job IDs and job titles in the Job field when they are inserting a new employee record.

- Creating a Package in the Database
- Creating a Database Function
- Calling the REF CURSOR from a Method
- Displaying a Dynamically Generated List

To do this, you create a database function, `GET_JOBS`, that uses a `REF CURSOR` to retrieve a result set of jobs from the `Jobs` table. A new Java method, `getJobs`, calls this database function to retrieve the result set.

### Creating a Package in the Database

The following steps create a new package in the database to hold a `REF CURSOR` declaration.

1. Select the **Connections** tab to view it in the Navigator.

2. Expand the **Database** node, the **DBConnection1** node, and the **HR** node, to view the list of database objects. Scroll down to Packages. Right-click **Packages** and select **New PL/SQL Package**.

3. In the Create PL/SQL Package dialog, enter `JOBSPKG` as the name. Click **OK**. The package definition is displayed in the Source Editor.

4. Position the cursor at the end of the first line and press Enter to create a new line. In the new line, declare a `REF CURSOR` as follows:

   ```
   TYPE ref_cursor IS REF CURSOR;
   ```

5. Save the package.

The code for the package is shown in Example 6–10:

***Example 6–10   Creating a Package in the Database***

```
PACKAGE "JOBSPKG" AS
    TYPE ref_cursor IS REF CURSOR;
END;
```

### Creating a Database Function

These steps create a database function `GET_JOBS` that uses a `REF CURSOR` to retrieve a result set of jobs from the `Jobs` table.

1. In the Connections Navigator, again expand the necessary nodes to view the objects in the `HR` database. Right-click **Functions** and select **New PL/SQL Function** from the shortcut menu.

2. In the Create PL/SQL Function dialog, enter `GET_JOBS` as the name. Click **OK**. The definition for the `GET_JOBS` function displays in the Source Editor

3. In the first line of the function definition, substitute `JobsPkg.ref_cursor` as the return value, in place of `VARCHAR2`.

4. After the `AS` keyword, enter the following:

```
jobs_cursor JobsPkg.ref_cursor;
```

5. In the `BEGIN` block enter the following code to replace the current content:

```
OPEN jobs_cursor FOR
SELECT job_id, job_title FROM jobs;
RETURN jobs_cursor;
```

6. Save the function

The code for the function is shown in Example 6–11.

**Example 6–11  Creating a Stored Function**

```
FUNCTION    "GET_JOBS"
RETURN JobsPkg.ref_cursor
AS jobs_cursor JobsPkg.ref_cursor;
BEGIN
  OPEN jobs_cursor FOR
  SELECT job_id, job_title FROM jobs;
  RETURN jobs_cursor;
END;
```

## Calling the REF CURSOR from a Method

These steps create a Java method, `getJobs`, in the `DataHandler` class that calls the `GET_JOBS` function to retrieve the result set.

1. Double-click `DataHandler.java` to open it in the Source Editor if it is not already open.

2. Enter the method declaration.

```
public ResultSet getJobs() throws SQLException {

}
```

3. Within the method body, connect to the database.

```
getDBConnection();
```

4. Following the connection, declare a new variable, `jobquery`:

```
String jobquery = "begin ? := get_jobs; end;";
```

5. Create a CallableStatement using the `prepareCall` method:

```
CallableStatement callStmt = conn.prepareCall(jobquery);
```

6. Register the type of the `OUT` parameter, using an Oracle-specific type.

```
callStmt.registerOutParameter(1, OracleTypes.CURSOR);
```

7. When you specify that you want to use an Oracle-specific type, JDeveloper displays a message asking you to use Alt+Enter to import `oracle.jdbc.OracleTypes`. Press Alt+Enter, and then select **OracleTypes** (`oracle.jdbc`) from the list that appears.

8. Run the statement and return the result set.

```
callStmt.execute();
rset = (ResultSet)callStmt.getObject(1);
```

**9.** Enclose the code entered so far in a `try` block.

**10.** Add a catch block to catch any exceptions, and call your logException method as well.

```
catch ( SQLException ex ) {
  logException( ex );
}
```

**11.** After the close of the `catch` block, return the result set.

```
return rset;
```

**12.** Make the file to check for syntax errors.

The code for the `getJobs` method is as follows:

```
public ResultSet getJobs() throws SQLException {
try {
  getDBConnection();
  String jobquery = "begin ? := get_jobs; end;";
  CallableStatement callStmt = conn.prepareCall(jobquery);
  callStmt.registerOutParameter(1, OracleTypes.CURSOR);
  callStmt.execute();
  rset = (ResultSet)callStmt.getObject(1);
} catch ( SQLException ex ) {
logException( ex );
}
return rset;
}
```

### Displaying a Dynamically Generated List

To create the drop down list displaying the list of job IDs and job titles in the Insert page, you hard-coded the job IDs and job titles. In the following steps, you replace this with a dynamically-generated list provided by the REF CURSOR created in the previous section.

**1.** Double-click `insert.jsp` in the Application Navigator to open it in the Visual Editor, if it is not already open.

**2.** Drag a **Page Directive** onto the page to the right of the useBean tag. In the Insert Page Directive dialog box, enter Java as the **Language**, and in the **Import** field, browse to select **java.sql.ResultSet**. Click **OK**.

**3.** Drag a scriptlet onto the page next to the Page Directive. In the Insert Scriptlet dialog box, add the following code to execute the getJobs method and return a result set containing a list of jobs.

```
ResultSet rset = empsbean.getJobs();
```

**4.** Select the **ListBox** component in the page, and click **Scriptlet** in the JSP Component Palette. (You need not drag and drop the scriptlet onto the page in this case.) The Insert Scriptlet dialog box appears.

**5.** Enter the following code into the Insert Scriptlet dialog box. Click **OK**.

```
while (rset.next ())
 {
  out.println("<option value=" + rset.getString("job_id") + ">" +
```

```
                  rset.getString("job_title") + "</option> "  );
              }
```

6.  Remove the hard-coded values as follows.

    With the **ListBox** component still selected, in the Structure window scroll to **Job** field. Examine the list of hard-coded options below the select keyword. Delete each of the options, ensuring that you retain the scriptlet.

*Figure 6–3   Structure View of Dropdown ListBox Options*
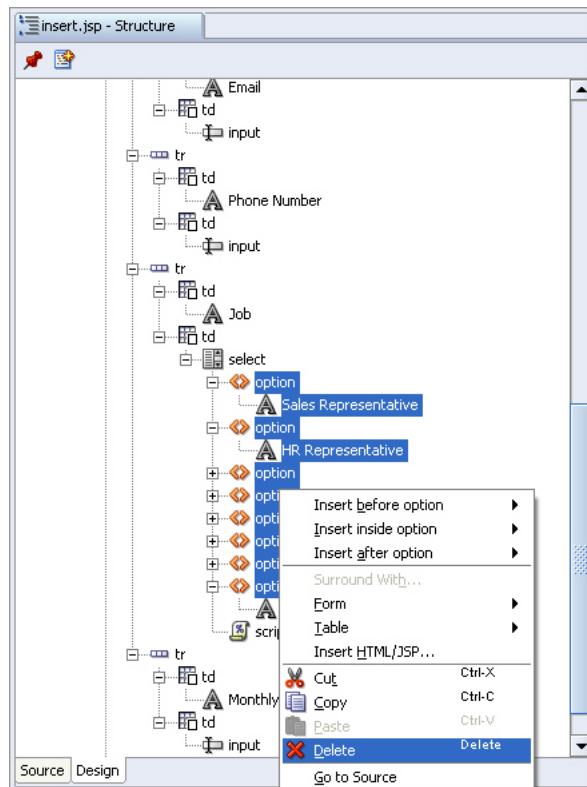


7.  Save the page.

Now run the application, click to insert a new employee and use the list to display a list of available jobs. Figure 6–4 shows the dynamic jobs list in the browser.

**Figure 6–4   Dynamically Generated List in Browser**

# 7
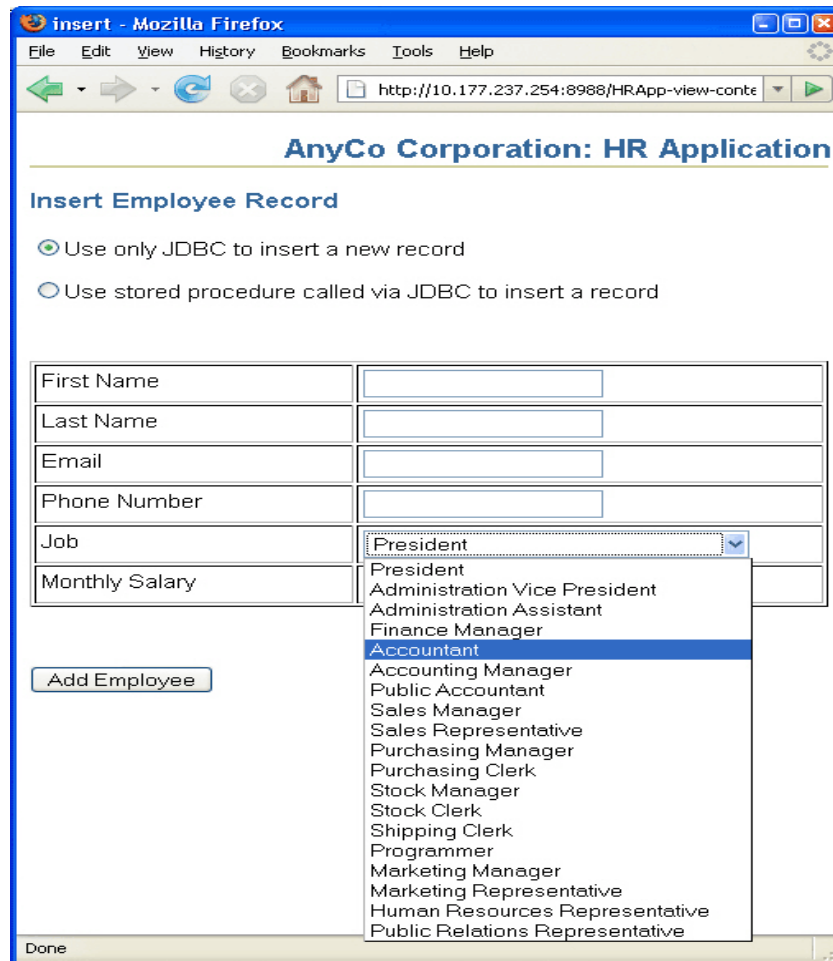
# Creating a Master-Detail Application Using Oracle ADF

This chapter describes how to create a master-detail application using Oracle Application Developer Framework (Oracle ADF) in the following sections:

- Overview of the Master-Detail Application
- Using Oracle ADF
- Creating an Application and Projects
- Creating Business Components in the model Project
- Displaying Master-Detail Data
- Navigation Between Application Pages: JSF Navigation Diagrams
- Editing Data
- Enabling COMMIT and ROLLBACK
- Running the Application

## Overview of the Master-Detail Application

A master-detail application allows you to view data from related tables at the same time. The records from a master table can be viewed along with related records from the detail table. If provisioning to edit the master-detail data is built into the application, you can also edit data from both the tables from a common interface.

The master-detail application created in this chapter consists of:

- An ADF middle-tier set of Java/XML objects to allow data in the table from the `HR` schema to be accessed and updated. This is in one project called `model`.

- A user interface (UI), or view, that consists of a set of JSPs that serve as the UI for the application. This will be in a project called `view`.

The `model` and `view` projects are based on the Java EE Model-View-Controller (MVC) design pattern, that is easily implemented using Oracle ADF.

Figure 7–1 shows the relationships among the items developed for this application.

*Figure 7–1   Master Detail Application Pages*
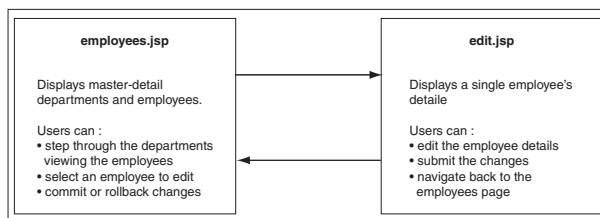


This application accesses the HR schema on Oracle Database. It uses the departments table as the master table to display detail data from the employees table. This chapter describes how you can use Oracle ADF with JDeveloper to create this application.

# Using Oracle ADF

Oracle ADF is an end-to-end application framework that builds on Java EE standards and open-source technologies to simplify and accelerate creating service-oriented applications. You can use Oracle ADF to develop enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces. Used in tandem, Oracle JDeveloper 10*g* and Oracle ADF give you an environment that covers the full development lifecycle from design to deployment, with drag-and-drop data binding, visual UI design, and team development features built-in.

The following subsections introduce some of the Oracle ADF features that you will use to create the master detail application:

- Oracle ADF Business Components

- Oracle ADF Faces

- ADF Data Controls

> **See Also:**
> - http://www.oracle.com/technology/products/jdev/collateral/papers/1013/adf_10.1.3_overview.pdf for more information on Oracle ADF architecture
>
> - http://www.oracle.com/technology/products/jdev/tips/muench/requiredreading/index.html for a compilation of resources on Oracle ADF

## Oracle ADF Business Components

Oracle ADF Business Components is Java EE-compliant technology for developing business services for database-centric, enterprise Java EE applications. Oracle ADF Business Components simplifies building business services for developers familiar with 4GL tools like Oracle Forms.

Oracle ADF Business Components technology:

- Automatically handles object-relational 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

- Provides a framework for implementing complex business logic

- Automatically implements many Java EE design patterns
- Has a powerful caching and data passivation system for increasing the performance and scalability of applications

All of this functionality is fully customizable. To create the ADF middle-tier for the application created in the book, you create ADF Business Components from database tables.

## Oracle ADF Faces

Oracle ADF Faces is based on the JavaServer Faces (JSF) JSR 127 specification. Oracle ADF Faces components are used in the user interfaces of the application. These components can be used in any IDE that supports JSF.

You can use Oracle ADF Faces to determine a consistent look and feel for your application. This allows you to focus on user interface interaction rather than look and feel compliance. ADF Faces components also support multi–language and translation implementation as well as accessibility features.

JDeveloper provides several design tools, wizards, special dialogs, and property editors that help you insert and use ADF Faces components in your pages. For example, the Visual Editor lets you design user interfaces by dragging and dropping components from the Component Palette. If you are familiar with XML or JSP/HTML coding, you can also edit the source of the page files to insert ADF Faces component tags.

### Facets Usage with Oracle ADF Faces

Facets are similar to named children components. You use facets inside parent components to control how some child components should be displayed on a page. Facets are placeholders for certain types of UI components.

The ADF Faces `af:panelPage` component, which is used in the sample application, lets you lay out an entire page. It supports facets for laying out page-level and application-level text, images, actions, and buttons in specific areas of the page.

When you drop an ADF Faces component onto a page, the JSP/HTML Visual Editor displays any facets as dotted line rectangular boxes.

## ADF Data Controls

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

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. In the sample application, it is not necessary to create data controls for the Oracle ADF Business Components you create, because they already implement the data control interfaces for you.

# Creating an Application and Projects

Before you proceed to developing the master detail application, you must create a `Connection` object that establishes a connection between the application and the database. For instructions to create a `Connection` object, refer to Chapter 3.

1. From the **File** menu, select **New** to display the New Gallery. From the **General** category, select **Application**.

2. In the Create Application dialog box, enter `AnyCo_ADF_MD` as the **Name** of the application, leave the **Application Package Prefix** blank, and for the **Application Template**, select **No template - All technologies**. Then click **OK**.

3. In the Create Project dialog, enter `model` as the **Project Name** and click **OK**.

You now have an application called `AnyCo_ADF_MD`, which contains a project called `model`.

## Creating Business Components in the model Project

In the model project, you will create the ADF Business Components to allow the `hr.Departments` and `hr.Employees` tables to be displayed and edited by the application.

1. In the JDeveloper Application Navigator, select the **model** project.

2. From the **File** menu, select **New** to display the New Gallery. Expand the **Business Tier** category, and select **ADF Business Components**. In the **Items** list, select **Business Components from Tables.**

3. In the Initialize Business Components Project screen, make sure the **hr** connection is selected, and click **OK**.

4. On the Create Business Components from Tables – Welcome screen, click **Next**.

5. On the Entity Objects screen, to see a list of the tables available, select the **Auto-Query** checkbox. From the **Available** list of tables, move the **DEPARTMENTS** and **EMPLOYEES** tables over to the **Selected** list as shown in Figure 7–2. Click **Next**.

**Figure 7–2   Selecting Schema Objects to Create Entity Objects**
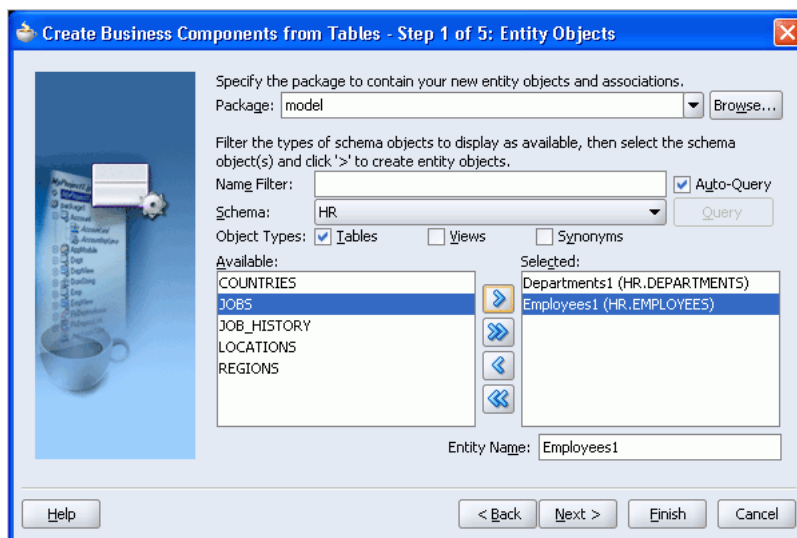


6. On the Updatable View Objects screen, move **Departments (HR.DEPARTMENTS)** and **Employees (HR.EMPLOYEES)** to the **Selected** list. Click **Next**.

7. On the Read-Only View Objects screen, click **Next**.

8. On the Application Module screen, you can name your ADF Application Module. Enter `AnyCoAppModule` as the **Name**. Click **Next**.

9. On the Diagram, you can request a business components diagram. However, this is not essential for this application, so click **Next**.

10. On the Finish dialog box, check the details and if they are correct, click **Finish**. The business components you defined are created in the `model` project.

11. Save all your work.

You have now generated an ADF middle-tier, for which you can build a user interface. At this point, you additional business component objects that handle the foreign key relationships from the database are also generated. These include: `DeptMgrFkAssoc`, `DeptMgrFkLink`, and so on.

Figure 7–3 shows the Application Navigator, where you can expand the contents of the `model` project, and see the items that are created for your application.

**Figure 7–3  Navigating Through an ADF Model**
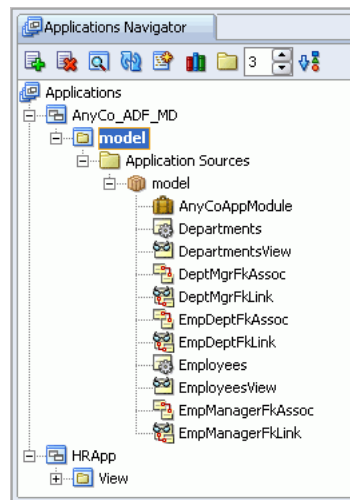


## Displaying Master-Detail Data

The application user interface consists of a set of JSP pages. For this application, the user interface (UI), referred to as the view, is defined in a separate project.

## Creating a Project for the Application UI

To create the application UI, you define a project called `view` as follows:

1. In the Application Navigator, select the **AnyCo_ADF_MD** application and from the **File** menu, select **New** to display the New Gallery. From the **General** category, select **Empty Project**.

2. In the Create Project dialog box, enter `view` as the **Name** of the new project, and click **OK**.

## Creating  a JSP to Display Employee Details

Using the following instructions, you create a page called `employees.jsp`, which will be used to allow users to browse through a coordinated set of master-detail pages of departments and employees.

1. In the Application Navigator, select the **view** project. Right-click the project and select **New** to display the New Gallery.

2. In the New Gallery, expand the **Web Tier** category and select **JSF**. In the **Item** list, select **JSF JSP** and click **OK**.

3. On the Create JSF JSP – Welcome screen, click **Next**.

4. On the Web Application screen, accept the default and click **Next**.

5. On the JSP File screen, enter `employees.jsp` as the **File Name**, and ensure that **JSP Page (*.jsp)** is selected for the page type. Click **Next**.

6. On the Component Binding screen, ensure that **Do Not Automatically Expose UI Components in a Managed Bean** is selected. Click **Next**.

7. On the Tag Libraries screen, if they are not already included, add **ADF Faces Components** and **ADF Faces HTML** to **Selected Libraries** on the right-hand side. The **JSF Core** and **JSF HTML** tag libraries should also be listed as Selected Libraries on the right-hand side. This is shown in Figure 7–4.

**Figure 7–4   Selecting Libraries for the JSF JSP**
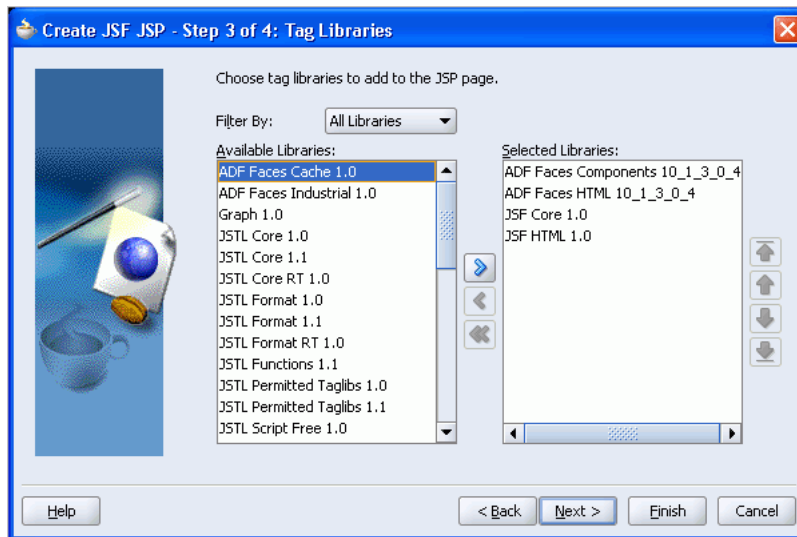


8. Click **Finish**.

The new empty `employees.jsp` is now displayed in the JSP/HTML Visual Editor, ready for you to start designing the page.

## Defining the Page Layout and Heading

In these steps, you add items to the page to define the page layout. As in earlier chapters, you use the Component Palette to drop tags onto your page. For this page, you use the ADF Faces PanelPage component.

1. If the Component Palette is not displayed, use the **View** menu to display it. Select the **ADF Faces Core** tab of components. Drag and drop the **PanelPage** component on the blank page. The ADF Faces Core page of the Component Palette is shown in Figure 7–5.

*Figure 7–5   ADF Faces Core Component Palette*
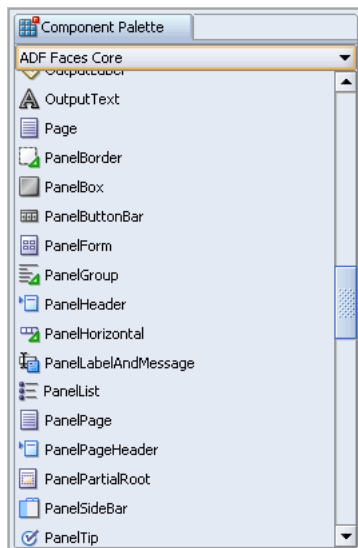


After dropping it, you should see a dark blue line with the text Title 1, as shown in Figure 7–6.

*Figure 7–6   ADF Faces PanelPage Component*



2.  On the page,  click on the PanelPage component and in the Property Inspector, change the value of **Title** field from `Title 1` to `Browse Employees`, then press Enter. The title change is now reflected in the page in the Visual Editor.

3.  In this step, you add another **ADF Faces** component onto the page to serve as the page title, by dragging a component into the branding section of the **PanelPage**. From ADF Faces Core page of the Component Palette, drag an **OutputText** component and drop it into the branding facet, that is, the dotted line square box with title branding in the upper left corner of the page.

4.  The new OutputText is now displayed with a default value of outputText1. Change this to read **AnyCo Corporation** by changing the value of OutputText in the Property Inspector.

5.  Now change the look and feel of the text by changing the **StyleClass** property of the OutputText to have the value **AFHeaderLevelOne**. This is a style which is defined by ADF Faces.

    Once changed, the text should appear larger and blue in the Visual Editor. Figure 7–7 shows the page in the Visual Editor after these steps.

*Figure 7–7   PanelPage Component with Added Text*
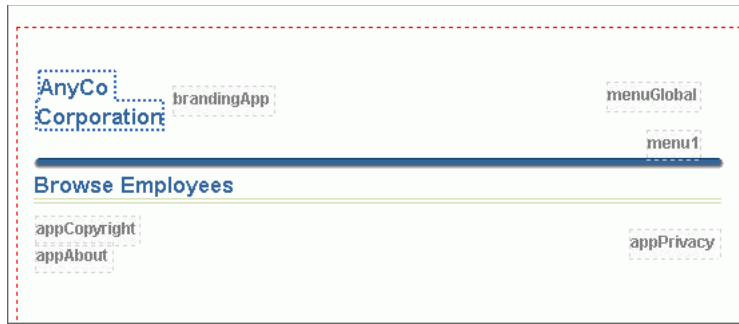


## Displaying Master Data on a JSP Page

In these steps you will add items to the page to allow users to browse departments and their employees. In these steps you use the Data Control Palette, which by default is on the right of the JDeveloper display, alongside the Component Palette. The Data Control Palette allows you to drop data objects called ADF Data Controls onto your pages.

The first stage is to add a read-only form which will display the departments along with navigation buttons to step through the different departments and their related employees. To do this, you drag onto the page one of the data controls you created in the model project.

1.  Select the **Data Control Palette**, and expand the **AnyCoAppModule** node. Figure 7–8 shows the Data Control Palette with the AnyCoAppModule expanded.

*Figure 7–8   Data Control Palette*
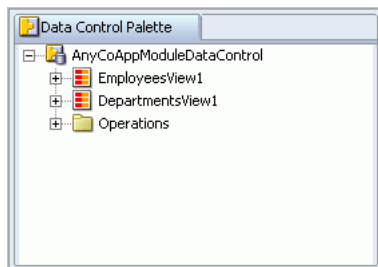


2.  In the Data Control Palette, select the **DepartmentsView1** node and drag it onto the middle of the page just beneath the Browse Employees text.

3.  As the drop occurs, a popup menu is displayed. Select **Create**, **Forms**, then **ADF read-only form**.

4.  In the Edit Form Fields dialog, select **Include Navigation Controls** and click **OK**. This is shown in Figure 7–9.

*Figure 7–9   Adding Form Fields*
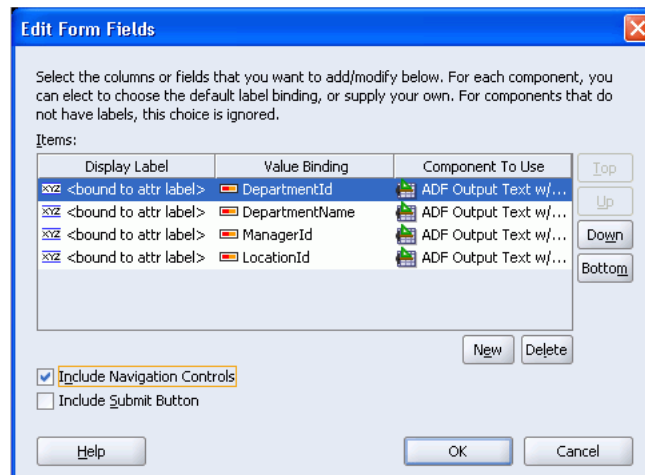


The Visual Editor now displays the read-only form with navigation buttons on the page. If the form does not appear, click the refresh button. The values of the form fields are provided by using expression language such as #{...}. This is how JSF identifies application data that is to be rendered on the page.

Figure 7–10 shows what the page looks like in the Visual Editor after these steps.

*Figure 7–10   Form Fields in Visual Editor*
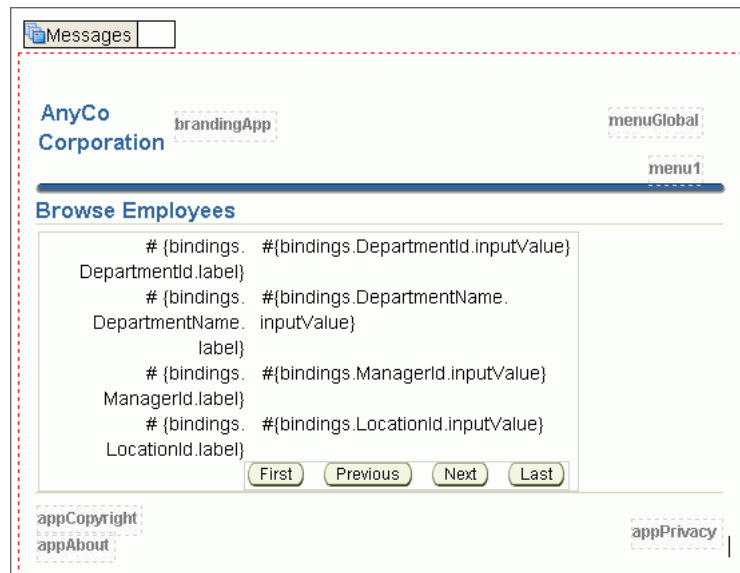


## Displaying Detail Data for Master Records

In these steps, you will first add a horizontal separator to the page, to separate the department data from the detail employee data. Then, to display the detail employee data for each master department, you use another of the data controls created from the Business Components you created in the model project. You choose the data control that links employee data to a particular master department. You display the employee data below the ObjectSeparator.

1. To add a separator to the page, from the ADF Faces Core page of the Component Palette drag an **ObjectSeparator** component and place it underneath the form in the Visual Editor. A dotted horizontal line is displayed beneath the form.

2. In the Data Control Palette, expand the **DepartmentsView1** node.

   In addition to the fields of the departments table, there is also a child EmployeesView3 node. This node represents the detailed or set of employees restricted by its foreign key to the departments table.

3. Drag and drop the **EmployeesView3** node onto the page beneath the dotted horizontal line (ObjectSeparator).

   As you hover the mouse before dropping, the Visual Editor and the Structure window on the lower left indicate where the drop will occur. Adjust the drop position in the Structure window so that it is after the af:objectSeparator as shown in Figure 7–11.

*Figure 7–11   Structure Window Showing Dropped Position*
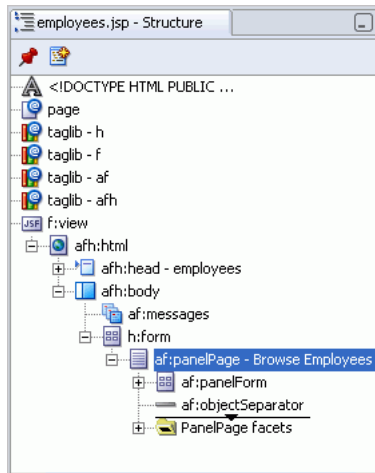


4. As the drop is finished, a shortcut menu is displayed. Select **Create**, then **Table**, and **ADF Read-only Table**.

5. In the Edit Table Columns dialog, select **Enable selection** and **Enable sorting**.

   Before clicking **OK**, delete the three columns: **DepartmentId**, **ManagerId**, and **CommissionPct**. For each of these rows in turn, select the row, then click **Delete**.

   The Edit Table Columns dialog box is shown in Figure 7–12.

*Figure 7–12   Editing Table Columns*
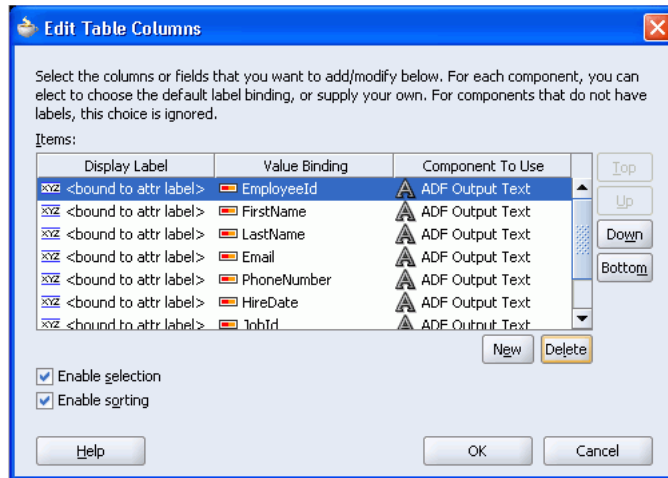


Then click **OK** to generate the table.

**6.** Save your work.

You should now see both the master form and detail table in the Visual Editor as in Figure 7–13.

*Figure 7–13   Master-Detail View in Visual Editor*
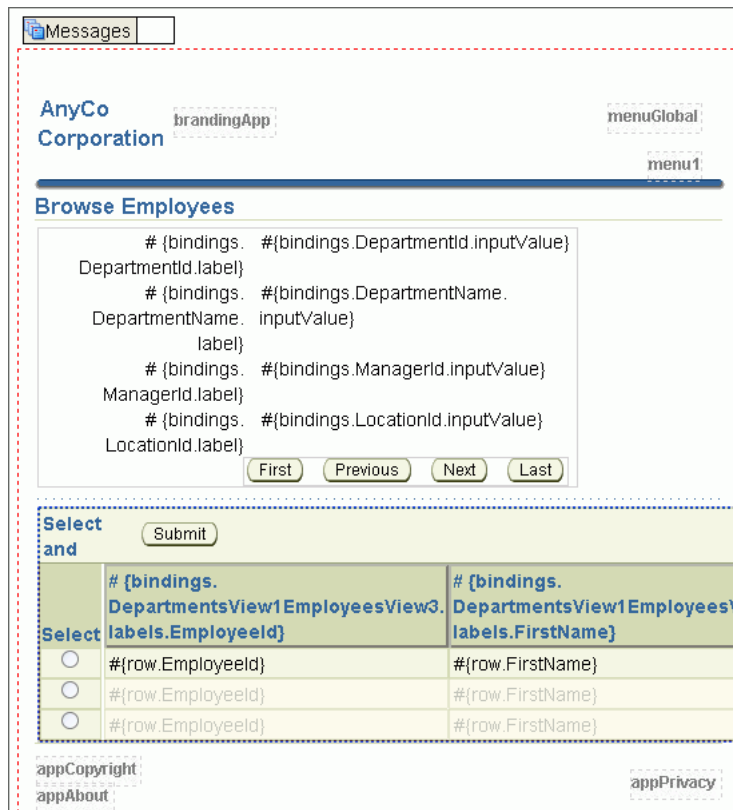
## Testing the Application

After adding the departments and employees to the page is a good stage to test the application so far. To test the application:

1. In the Application Navigator, right-click the page **employees.jsp** and select **Run**. This will start up the application locally by using the embedded OC4J server in JDeveloper. When the application has started, the master-detail employees page is displayed in a browser.

2. To test the master-detail coordination, click on the navigation buttons in the departments form (**Next**, **Previous**, **First**, **Last**). Notice as you click on the navigation buttons, that the related employees are displayed in the table below.

   The page when viewed in a browser is similar to that shown in Figure 7–14.

**Figure 7–14 Employee Data Viewed in a Browser**



3. To conclude the test, shutdown the server. To do this, in JDeveloper, right-click in the log window titled Running: Embedded OC4J Server – Log, and select **Terminate**.

# Navigation Between Application Pages: JSF Navigation Diagrams

You can use a JSF navigation diagram to plan out your application. The diagram shows the application pages and the navigation cases between them.

You can create new pages directly from the diagram and then define the navigation between them, using elements dropped from the Component Palette.

You can then edit your JSF pages with tools such as the Visual Editor, and can edit the navigation cases directly from the diagram.

## Creating a Page Using a JSF Navigation Diagram

To allow users to edit employee data, you will create a new page, `edit.jsp`. Onto this page you will drop an ADF input form from Data Control Palette.

In these steps, instead of creating a new page directly using the JSF JSP wizard, you will create it from the JSF navigation diagram. In later steps, you will also define the navigation between the pages visually on the diagram.

1. To open the JSF navigation diagram for the application, in the Application Navigator, right-click the **view** project and select **Open JSF Navigation**.

A blank navigation diagram is displayed.

2. To start defining the navigation rules, drag the existing **employees.jsp** page from the Application Navigator onto the blank diagram.

   A page icon is displayed on the navigation diagram representing the `employees.jsp` page.

3. To create a new page, with the JSF navigation diagram still displayed in the editing window, from the JSF Navigation Diagram page of the Component Palette, drag **JSF Page** onto the diagram.

   As the new page icon appears on the diagram, enter the text `edit.jsp` to replace the name `/untitled1.jsp` and press **Enter**. (You do not need to add the forward slash in front of the name, because it will be added automatically.)

   At this point, a new page has been added to the diagram, but the page file itself does not exist. To indicate that, the page icon renders with a yellow caution sign.

4. To create the `edit.jsp` page, double-click the page icon for the page `edit.jsp` in the diagram.

5. In the Create JSF JSP wizard, if the Welcome page is displayed, click **Next**. The name of the page is already entered for you, and the remaining steps should be exactly as for the `employees.jsp` page created earlier, so you can click **Finish**.

   The new page is displayed in the Visual Editor, ready for you to design the page.

## Navigating Between Pages

Navigation by a user through a JSF application is defined using navigation rules that determine which page is displayed next when the user clicks a link. The different cases, such as the different links on the page, are defined as navigation cases.

You can use a diagram to draw the navigation between the pages of the application representing the flow between the pages.

When you draw navigation on a JSF navigation diagram, two things happen: the navigation cases are added to the necessary configuration files automatically for you, and the navigation cases are shown on the diagram as arrows connecting pages, the direction of the arrow indicating the pages from which and to which the user will navigate.

A label is associated with each navigation case, to distinguish different navigation cases from a single page.

## Defining Navigation Between Pages

In these steps, to allow for navigation between the two pages of the application, you will add navigation details to the application.

1. Return to the JSF navigation diagram to define the navigation between the two pages. To do this, among the items open in the editing window, select the tab for **faces-config.xml**. If the diagram is not still open, open it as before.

2. In the diagram, click on the **employees.jsp** page icon to establish focus on this page.

3. On the JSF Navigation Diagram page of the Component Palette, click (not drag) **JSF Navigation Case**.

4. On the diagram, click the **employee.jsp** icon again. After you have clicked on the page, as you start to move the mouse away from the page, a line appears,

connected to the **employees.jsp** page. Connect this line to the new page by clicking on the **edit.jsp** page.

A line now connects the two pages, representing a navigation case navigating from the `employees.jsp` page to the `edit.jsp` page. It has a label with a default value of success.

5. To change the value, click **success**, type **edit**, and press the **Enter** key. The text edit is now displayed on the diagram as the label for the line.

6. Add another navigation case navigating from the **edit.jsp** page to the **employees.jsp** page, and give the navigation case the name **return**. This navigation case represents navigating from the `edit.jsp` page back to the `employees.jsp` page.

7. Save your work.

The navigation diagram now shows both navigation cases: one for a user to navigate from the `employees.jsp` page to the `edit.jsp` page so they can edit employee details, and another for the user to return to the `employees.jsp`. Figure 7–15 shows the navigation diagram for the `employees.jsp` and `edit.jsp` pages.

*Figure 7–15   Navigation Diagram*
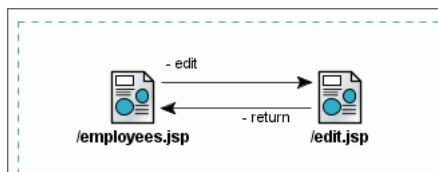


# Editing Data

To create an edit page in which users can edit employee data, you will use an ADF Faces component to lay out the page, in exactly the same way as for the employees page.

For the employee data, a data control is used. To display the correct employee data for a particular employee in a particular department, there is a data control available giving employee data as a child data control of the department data control you used on the employees page. This data control gives the correct set of employee data.

So that users can edit the data, you display it in an ADF Form, rather than a table.

## Creating an Edit Form

In these steps, you create the `edit.jsp` page:

1. Open the `edit.jsp` page in the Visual Editor.

2. On the `edit.jsp` page, from the ADF Faces Core page of the Component Palette, add a **PanelPage** component to the page. Change the title of the PanelPage to **Edit Employee**.

3. Also, add an **OutputText** component to the branding facet at the top of the page. Set the **Value** property to **AnyCo Corporation** and the **StyleClass** property to **AFHeaderLevelOne**.

4. In the Data Control Palette, expand the **DepartmentsView1** node (if it is closed) and locate the same **EmployeesView3** node beneath it. This must be the child of DepartmentsView1, as illustrated below.

**5.** Drag the **EmployeesView3** node onto the center of the panel beneath the Edit Employee title. In the dialog box that is displayed, select **Create**, **Forms**, and **ADF Form**.

> **Note:** It is important to select the child `EmployeesView3` node under parent `DepartmentsView1` from the Data Control Palette. The child `EmployeesView3` will show the detail Employee records for a particular parent `Deparment` as designated by the foreign key relationship.

**6.** In the Edit Form Fields dialog, select **Include Submit Button** and click **OK**.

The edit form is almost complete. The final task is to add a button that will allow the user to navigate back to the `employees.jsp` page. To do this:

**1.** From the ADF Faces Core page of the Component Palette, drag a **CommandButton** to the page and place it next to the **Submit** button at the bottom of the page.

**2.** Use the Property Inspector to change the **Text** property of the button from **commandButton** 1 to **Return**.

**3.** Again using the Property Inspector, change the **Action** property of the CommandButton to **Return**. You can use the arrow in the Action field to select the value return. This is the value you entered for the navigation case for a user navigating from the `edit.jsp` page back to the `employees.jsp` page.

**4.** Save the page.

The edit page is now complete, and should resemble the page shown in Figure 7–16.

*Figure 7–16   Editing the Employees detail form in the Visual Editor*



## Navigating to an Edit Page

One remaining task is to allow users viewing the employees page to navigate to the edit page to edit employee details. To do this, you make use of the navigation case you defined on the navigation diagram, which had the name edit. The Employees table on the employees page already has a button, which can be used for the user to navigate to the edit page.

1.   Open the `employees.jsp` page in the Visual Editor.

2.   Select the **Submit** button to establish focus.

3.   In the Property Inspector, change the **Text** property to **Edit**. Set the Action property to edit:  use the arrow in the **Action** field to select the value **edit**. This is the value you specified on the navigation diagram as the navigation case for a user to navigate from the `employees.jsp` page to the `edit.jsp` page.

Now when a user clicks on this button, the application will navigate to the edit page with the selected record displayed.

## Enabling COMMIT and ROLLBACK

To enable commit and rollback functionality, so a user can commit the edits they have made to employee details, or abandon them by rolling back, you will add a Commit button and a Rollback button to the base of the `employees.jsp` page.

This step again uses facets. In this case, a footer facet is added to the table, and the Commit and Rollback buttons are added inside of this footer facet.

In these steps, to add the footer facet, you use the Structure window.

1. With `employees.jsp` displayed in the Visual Editor, in the Structure window, expand the table component (**af:table**). Within the structure, there is a Table Facets folder.

2. Expand the **Table Facets** folder. It contains several subfolders for different facets supported by the ADF Faces table component.

   The footer facet is grey, because it is currently empty, as illustrated in Figure 7–17.

*Figure 7–17   Empty Footer Facet*
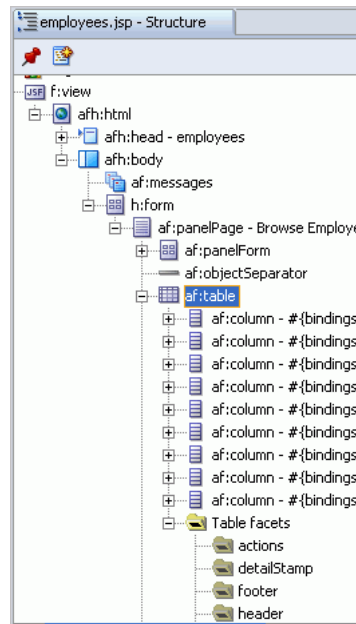


You will insert a footer facet into this placeholder, ready to hold the Commit and Rollback buttons.

3. In the Structure window, right-click the footer facet node and select **Facets -Table**, and then **Footer**.

   The footer facet is added to the page at the base of the table as shown in Figure 7–18.

*Figure 7–18   Inserting the Footer Facet*
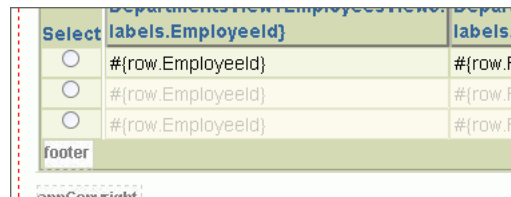


You are now able to place JSF components into the footer facet.

The Commit and Rollback buttons you need are available in the Data Control Palette in the Operations node which is an immediate child of the parent AnyCoAppModuleDataControl.

To add the buttons to your page:

1. In the Data Control Palette, expand the **AnyCoAppModuleDataControl** folder, then the **Operations** folder. The Operations folder contains the Commit and Rollback nodes.

2. Drag each of the **Commit** and **Rollback** nodes onto the footer facet at the base of the table.

   As you drop each of the operations from the Data Control Palette onto the footer facet, a dialog box is displayed. Select **Create ADF Command Button**.

3. Save all your work.

The application is now complete. The final `employees.jsp` page is as shown in Figure 7–19.

*Figure 7–19   Finished Master-Detail Application in Visual Editor*
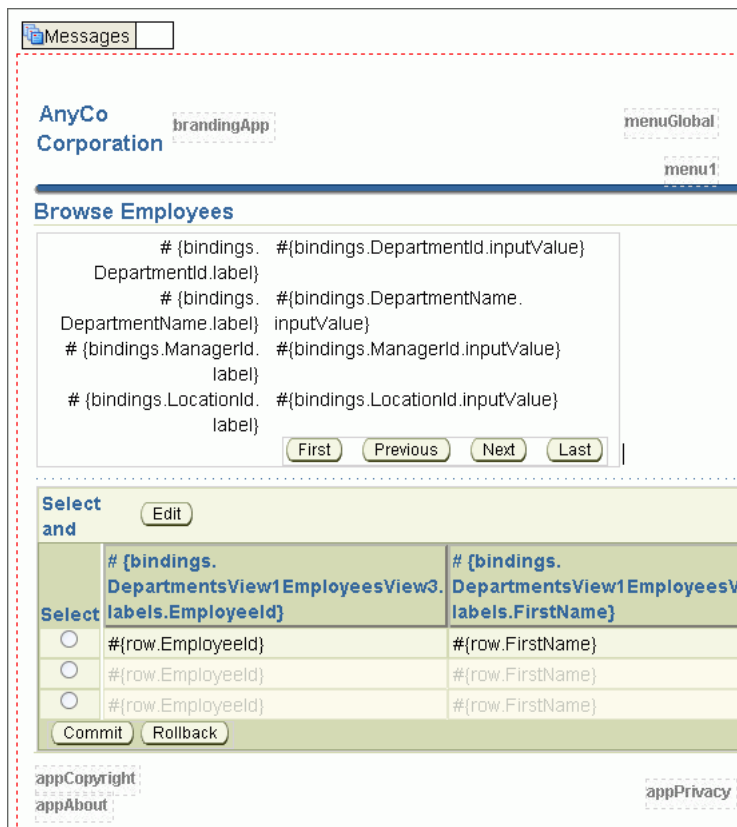


## Running the Application

You may now run the application as follows:

1. In the Application Navigator, right-click `employees.jsp` and select **Run** from the shortcut menu.

2.  As you run the application, you will be able to navigate through the different Departments and then select individual Employees for editing. Experiment with updating either the salary or hiredate of an employee.

The Employees page displayed in a browser is shown in Figure 7–20.

*Figure 7–20   Master-Detail Application Viewed in a Browser*



The edit page displayed in a browser is similar to that shown in Figure 7–21.

*Figure 7–21   Editing the Master Detail Application Content*

# 8

# Getting Unconnected from Oracle Database

While unconnecting from the database in JDeveloper is a simple task, it is not a process by itself in a Java application. In the application, you must explicitly close all `ResultSet`, `Statement`, and `Connection` objects after you are through using them. When you close the `Connection` object, you are unconnected from the database. The `close` methods clean up memory and release database cursors. Therefore, if you do not explicitly close `ResultSet` and `Statement` objects, serious memory leaks may occur, and you may run out of cursors in the database. You must then close the connection.

This chapter includes the following sections:

- Creating a Method to Close All Open Objects
- Closing Open Objects in the Application

## Creating a Method to Close All Open Objects

The following steps add a `closeAll` method to the `DataHandler` class:

1. Open `DataHandler.java` in the Java Source Editor by double-clicking it in the Application Navigator.

2. Declare the `closeAll` method at the end of the `DataHandler` class as follows:

   ```
   public void closeAll() {

   }
   ```

3. Within the method body, check whether the `ResultSet` object is open as follows:

   ```
   if ( rset != null ) {
   ```

4. If it is open, close it and handle any exceptions as follows:

   ```
     try { rset.close(); } catch ( Exception ex ) {}
     rset = null;
   }
   ```

5. Repeat the same actions with the `Statement` object.

   ```
   if ( stmt != null ) {
     try { stmt.close(); } catch ( Exception ex ) {}
     stmt = null;
   }
   ```

6. Finally, close the `Connection` object.

   ```
   if ( conn != null ) {
   ```

```
try { conn.close(); } catch ( Exception ex ) {}
conn = null;
}
```

## Closing Open Objects in the Application

You must close the `ResultSet`, `Statement`, and `Connection` objects only after you have finished using them. In the `DataHandler` class, the insert, update, and delete methods must close these objects before returning. Note that the query methods cannot close these objects until the `employees.jsp` page has finished processing the rows returned by the query.

In the following steps, you add the appropriate calls to the `closeAll` method in the `DataHandler.java` file:

1. Open `DataHandler.java` in the Java Source Editor.

2. At the end of the `addEmployee` method, after the closing brace of the `catch` block, add the following call to the `closeAll` method in a `finally` block:

   ```
   finally {
     closeAll();
   }
   ```

3. Add the same call to the `addEmployeeSP`, `deleteEmployeeById`, `findEmployeeById`, `updateEmployee`, and `authenticateUser` methods.

4. Open the `employees.jsp` file in the Visual Editor. Find the scriptlet inside the `Employees` table, and double-click to open the Insert Scriptlet dialog box.

5. Add the following statement after the `while` loop:

   ```
   empsbean.closeAll();
   ```

6. Save your work, and compile and run the application to ensure that everything still works correctly.

# 9

# Building Global Applications

Building a global Internet application that supports different locales requires good development practices. A locale refers to a national language and the region in which the language is spoken. The application itself must be aware of user locale preferences and present content following the cultural convention expected by the user. It is important to present data with appropriate locale characteristics, such as using the correct date and number formats. Oracle Database is fully internationalized to provide a global platform for developing and deploying global applications.

This chapter discusses global application development in a Java and Oracle Database environment. It addresses the basic tasks associated with developing and deploying global Internet applications, including developing locale awareness, constructing HTML content in the user-preferred language, and presenting data following the cultural conventions of the user locale.

This chapter has the following topics:

- Developing Locale Awareness
- Determining User Locales
- Encoding HTML Pages
- Organizing the Content of HTML Pages for Translation
- Presenting Data by User Locale Convention
- Localizing Text on JSP Pages in JDeveloper

## Developing Locale Awareness

Global Internet applications must be aware of the user locale. Locale-sensitive functions, such as date, time, and monetary formatting, are built into programming environments such as Java and SQL. Applications can use locale-sensitive functions to format the HTML pages according to the cultural conventions of the user locale.

Different programming environments represent locales in different ways. For example, the French (Canadian) locale is represented as follows:

| Environment | Representation | Locale | Explanation |
|---|---|---|---|
| Java | Java locale object | `fr_CA` | Java uses the ISO language and country code. |
| | | | `fr` is the language code defined in the ISO 639 standard. `CA` is the country code defined in the ISO 3166 standard. |

| Environment | Representation | Locale | Explanation |
|---|---|---|---|
| SQL and PL/SQL | `NLS_LANGUAGE` and `NLS_TERRITORY` parameters | `NLS_LANGUAGE ="CANADIAN FRENCH"`<br><br>`NLS_TERRITORY ="CANADA"` | See also: Chapter 8 "Working in a Global Environment" in the *Oracle Database Express Edition 2 Day Developer Guide*. |

Table 9–1 shows how some of the commonly used locales are defined in Java and Oracle environments.

*Table 9–1    Locale Representation in Java, SQL, and PL/SQL Programming Environments*

| Locale | Java | NLS_LANGUAGE, NLS_TERRITORY |
|---|---|---|
| Chinese (P.R.C) | `zh_CN` | `SIMPLIFIED CHINESE, CHINA` |
| Chinese (Taiwan) | `zh_TW` | `TRADITIONAL CHINESE, TAIWAN` |
| English (U.S.A) | `en_US` | `AMERICAN, AMERICA` |
| English (United Kingdom) | `en_GB` | `ENGLISH, UNITED KINGDOM` |
| French (Canada) | `fr_CA` | `CANADIAN FRENCH, CANADA` |
| French (France) | `fr_FR` | `FRENCH, FRANCE` |
| German (Germany) | `de_DE` | `GERMAN, GERMANY` |
| Italian (Italy) | `it_IT` | `ITALIAN, ITALY` |
| Japanese (Japan) | `ja_JP` | `JAPANESE, JAPAN` |
| Korean (Korea) | `ko_KR` | `KOREAN, KOREA` |
| Portuguese (Brazil) | `pt_BR` | `BRAZILIAN PORTUGUESE, BRAZIL` |
| Portuguese (Portugal) | `pt_PT` | `PORTUGUESE, PORTUGAL` |
| Spanish (Spain) | `es_ES` | `SPANISH, SPAIN` |

When writing global applications across different programming environments, the user locale settings must be synchronized between environments. For example, Java applications that call PL/SQL procedures should map the Java locales to the corresponding `NLS_LANGUAGE` and `NLS_TERRITORY` values and change the parameter values to match the user locale before calling the PL/SQL procedures.

## Mapping Between Oracle and Java Locales

The Oracle Globalization Development Kit (GDK) provides the `LocaleMapper` class. It maps equivalent locales and character sets between Java, IANA, ISO, and Oracle. A Java application may receive locale information from the client that is specified in the Oracle locale name. The Java application must be able to map to an equivalent Java locale before it can process the information correctly.

Example 9–1 shows how to use the `LocaleMapper` class.

*Example 9–1    Mapping from a Java Locale to an Oracle Language and Territory*

```
Locale locale = new Locale("fr", "CA");
String oraLang = LocaleMapper.getOraLanguage(locale);
String oraTerr = LocaleMapper.getOraTerritory(locale);
```

The GDK is a set of Java application programming interfaces (APIs) that provide Oracle application developers with the framework to develop globalized Internet applications. The GDK complements the existing globalization features in Java. It

provides the synchronization of locale behaviors between a middle-tier Java application and the Oracle database server.

> **See Also:** For more information about the GDK, refer to
>
> http://www.oracle.com/technology/tech/globalization/gdk/inde x.html

# Determining User Locales

In a global environment, your application may have to accept users with different locale preferences. Determine the preferred locale of the user. Once that is known, the application should construct HTML content in the language of the locale, and follow the cultural conventions implied by the locale.

One of the most common methods in determining the user locale, is based on the default ISO locale setting of the browser of the user. Usually a browser sends locale preference settings to the HTTP server with the Accept-Language HTTP header. If this header is set to NULL, then there is no locale preference information available and the application should ideally fall back to a predefined application default locale.

Both JSP pages and Java Servlets can use calls to the Servlet API to retrieve the Accept-Language HTTP header as shown in Example 9–2.

**Example 9–2   Determining User Locale in Java Using the Accept-Language Header**

```
String lang = request.getHeader("Accept-Language")
StringTokenizer st = new StringTokenizer(lang, ",")
if (st.hasMoreTokens()) userLocale = st.nextToken();
```

This code gets the Accept-Language header from the HTTP request, extracts the first ISO locale, and uses it as the user-desired locale.

## Locale Awareness in Java Applications

A Java locale object represents the locale of the corresponding user in Java. The Java encoding used for the locale is required to properly convert Java strings to and from byte data. You must consider the Java encoding for the locale if you make the Java code aware of a user locale. There are two ways to make a Java method sensitive to the Java locale and encoding:

- Using the default Java locale and default Java encoding for the method
- Explicitly specifying the Java locale and Java encoding for the method

When developing a global application, it is recommended to take the second approach and explicitly specify the Java locale and Java encoding that correspond to the current user locale. You can specify the Java locale object that corresponds to the user locale, identified by user_locale, in the getDateTimeInstance method as in Example 9–3.

**Example 9–3   Explicitly Specifying User Locale in Java**

```
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL,
user_locale);
dateString = df.format(date); /* Format a date */
```

# Encoding HTML Pages

The encoding of an HTML page is important information for a browser and an Internet application. You can think of the page encoding as the character set used for the locale that an Internet application is serving. The browser needs to know about the page encoding so that it can use the correct fonts and character set mapping tables to display the HTML pages. Internet applications need to know about the HTML page encoding so they can process input data from an HTML form.

Instead of using different native encodings for the different locales, it is recommended that UTF-8 (Unicode encoding) is used for all page encodings. Using the UTF-8 encoding not only simplifies the coding for global applications, but it allows for multilingual content on a single page.

This section includes the following topics:

- Specifying the Page Encoding for HTML Pages
- Specifying the Page Encoding in Java Servlets and JSP Pages

## Specifying the Page Encoding for HTML Pages

There are two ways to specify the encoding of an HTML page, one is in the HTTP header, and the other is in the HTML page header.

### Specifying the Encoding in the HTTP Header

Include the `Content-Type` HTTP header in the HTTP specification. It specifies the content type and character set as shown in Example 9–4.

**Example 9–4   Specifying Page Encoding in the HTTP Specification**

```
Content-Type: text/html; charset=utf-8
```

The `charset` parameter specifies the encoding for the HTML page. The possible values for the `charset` parameter are the IANA names for the character encodings that the browser supports.

### Specifying the Encoding in the HTML Page Header

Use this method primarily for static HTML pages. Specify the character encoding in the HTML header as shown in Example 9–5.

**Example 9–5   Specifying Page Encoding on an HTML Page**

```
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
```

The `charset` parameter specifies the encoding for the HTML page. As with the `Content-Type` HTTP Header, the possible values for the charset parameter are the IANA names for the character encodings that the browser supports.

## Specifying the Page Encoding in Java Servlets and JSP Pages

You can specify the encoding of an HTML page in the `Content-Type` HTTP header in a JavaServer Pages (JSP) file using the `contentType` page directive. For example:

```
<%@ page contentType="text/html; charset=utf-8" %>
```

This is the `MIME` type and character encoding that the JSP file uses for the response it sends to the client. You can use any `MIME` type or IANA character set name that is valid for the JSP container. The default `MIME` type is `text/html`, and the default character set

is ISO-8859-1. In the above example, the character set is set to UTF-8. The character set of the contentType page directive directs the JSP engine to encode the dynamic HTML page and set the HTTP Content-Type header with the specified character set.

For Java Servlets, you can call the setContentType method of the Servlet API to specify a page encoding in the HTTP header. The doGet function in Example 9–6 shows how you can call this method.

***Example 9–6    Specifying Page Encoding in Servlets Using setContentType***

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{

// generate the MIME type and character set header
response.setContentType("text/html; charset=utf-8");

...

// generate the HTML page
Printwriter out = response.getWriter();
out.println("<HTML>");

...

out.println("</HTML>");
}
```

You should call the setContentType method before the getWriter method because the getWriter method initializes an output stream writer that uses the character set specified by the setContentType method call. Any HTML content written to the writer and eventually to a browser is encoded in the encoding specified by the setContentType call.

# Organizing the Content of HTML Pages for Translation

Making the user interface available in the local language of the user is one of the fundamental tasks related to globalizing an application. Translatable sources for the content of an HTML page belong to the following categories:

- Text strings hard-coded in the application code
- Static HTML files, images files, and template files such as CSS
- Dynamic data stored in the database

This section discusses externalizing translatable content in the following:

- Strings in Java Servlets and JSP Pages
- Static Files
- Data from the Database

## Strings in Java Servlets and JSP Pages

You should externalize translatable strings within Java Servlets and JSP pages into Java resource bundles so that these resource bundles can be translated independent of the Java code. After translation, the resource bundles carry the same base class names as the English bundles, but with the Java locale name as the suffix. You should place

the bundles in the same directory as the English resource bundles for the Java resource bundle look-up mechanism to function properly.

> **See Also:** Sun Microsystems documentation about Java resource bundles at
>
> http://java.sun.com/j2se/1.4.2/docs/api/java/util/ResourceBundle.html

Because the user locale is not fixed in multilingual applications, they should call the `getBundle` method by explicitly specifying a Java locale object that corresponds to the user locale. The Java locale object is called `user_locale` in the following example:

```
ResourceBundle rb = ResourceBundle.getBundle("resource", user_locale);
String helloStr = rb.getString("hello");
```

The above code will retrieve the localized version of the text string, `hello`, from the resource bundle corresponding to the desired locale of the user.

> **See Also:** For more information on creating resource bundles in Java, refer to Localizing Text on JSP Pages in JDeveloper on page 9-9.

## Static Files

Static files such as HTMLs and GIFs are readily translatable. When these files are translated, they should be translated into the corresponding language with UTF-8 as the file encoding. To differentiate between the languages of the translated files, the static files of different languages can be staged in different directories or with different file names.

## Data from the Database

Dynamic information such as product names and product descriptions are most likely stored in the database regardless of whether you use JSP pages or Java Servlets. In order to differentiate between various translations, the database schema holding this information should include a column to indicate the language of the information. To select the translated information, you must include the `WHERE` clause in your query to select the information in the desired language of the query.

# Presenting Data by User Locale Convention

Data in the application needs to be presented in a way that conforms to user expectation, if not, the meaning of the data can sometimes be misinterpreted. For example, '12/11/05' implies '11th December 2005' in the United States, whereas in the United Kingdom it means '12th November 2005'. Similar confusion exists for number and monetary formats, for example, the period (.) is a decimal separator in the United States, whereas in Germany, it is used as a thousand separator.

Different languages have their own sorting rules, some languages are collated according to the letter sequence in the alphabet, some according to stroke count in the letter, and there are some languages which are ordered by the pronunciation of the words. Presenting data that is not sorted according to the linguistic sequence that your users are accustomed to can make searching for information difficult and time-consuming.

Depending on the application logic and the volume of data retrieved from the database, it may be more appropriate to format the data at the database level rather than at the application level. Oracle Database offers many features that help you to

refine the presentation of data when the user locale preference is known. The following sections include examples of locale-sensitive operations in SQL:

- Oracle Date Formats
- Oracle Number Formats
- Oracle Linguistic Sorts
- Oracle Error Messages

## Oracle Date Formats

There are three different date presentation formats in Oracle Database. These are standard, short, and long dates. Example 9–7 illustrates the difference between the short data and long date formats for both United States and Germany.

**Example 9–7   Difference Between Date Formats by Locale (United States and Germany)**

```
SQL> ALTER SESSION SET NLS_TERRITORY=america NLS_LANGUAGE=american;

Session altered.

SQL> SELECT employee_id EmpID,
  2  SUBSTR(first_name,1,1)||'.'||last_name "EmpName",
  3  TO_CHAR(hire_date,'DS') "Hiredate",
  4  TO_CHAR(hire_date,'DL') "Long HireDate"
  5  FROM employees
  6* WHERE employee_id <105;

     EMPID EmpName                    Hiredate   Long HireDate
---------- -------------------------- ---------- -----------------------------
       100 S.King                     06/17/1987 Wednesday, June 17, 1987
       101 N.Kochhar                  09/21/1989 Thursday, September 21, 1989
       102 L.De Haan                  01/13/1993 Wednesday, January 13, 1993
       103 A.Hunold                   01/03/1990 Wednesday, January 3, 1990
       104 B.Ernst                    05/21/1991 Tuesday, May 21, 1991


SQL> ALTER SESSION SET SET NLS_TERRITORY=germany NLS_LANGUAGE=german;

Session altered.

SQL> SELECT employee_id EmpID,
  2  SUBSTR(first_name,1,1)||'.'||last_name "EmpName",
  3  TO_CHAR(hire_date,'DS') "Hiredate",
  4  TO_CHAR(hire_date,'DL') "Long HireDate"
  5  FROM employees
  6* WHERE employee_id <105;


     EMPID EmpName                    Hiredate Long HireDate
---------- -------------------------- -------- -----------------------------
       100 S.King                     17.06.87 Mittwoch, 17. Juni 1987
       101 N.Kochhar                  21.09.89 Donnerstag, 21. September 1989
       102 L.De Haan                  13.01.93 Mittwoch, 13. Januar 1993
       103 A.Hunold                   03.01.90 Mittwoch, 3. Januar 1990
       104 B.Ernst                    21.05.91 Dienstag, 21. Mai 1991
```

## Oracle Number Formats

Example 9–8 illustrates the differences in the decimal character and group separator between the United States and Germany.

***Example 9–8   Difference Between Number Formats by Locale (United States and Germany)***

```
SQL> ALTER SESSION SET SET NLS_TERRITORY=america;

Session altered.

SQL> SELECT employee_id EmpID,
  2  SUBSTR(first_name,1,1)||'.'||last_name "EmpName",
  3  TO_CHAR(salary, '99G999D99') "Salary"
  4  FROM employees
  5* WHERE employee_id <105

    EMPID EmpName                   Salary
---------- -------------------------- ----------
      100 S.King                     24,000.00
      101 N.Kochhar                  17,000.00
      102 L.De Haan                  17,000.00
      103 A.Hunold                    9,000.00
      104 B.Ernst                     6,000.00

SQL> ALTER SESSION SET SET NLS_TERRITORY=germany;

Session altered.

SQL> SELECT employee_id EmpID,
  2  SUBSTR(first_name,1,1)||'.'||last_name "EmpName",
  3  TO_CHAR(salary, '99G999D99') "Salary"
  4  FROM employees
  5* WHERE employee_id <105

    EMPID EmpName                   Salary
---------- -------------------------- ----------
      100 S.King                     24.000,00
      101 N.Kochhar                  17.000,00
      102 L.De Haan                  17.000,00
      103 A.Hunold                    9.000,00
      104 B.Ernst                     6.000,00
```

## Oracle Linguistic Sorts

Spain traditionally treats 'ch', 'll' as well as 'ñ' as letters of their own, ordered after c, l and n respectively. Example 9–9 illustrates the effect of using a Spanish sort against the employee names Chen and Chung.

***Example 9–9   Variations in Linguistic Sorting (Binary and Spanish)***

```
SQL> ALTER SESSION SET NLS_SORT=binary;

Session altered.

SQL> SELECT employee_id EmpID,
  2         last_name "Last Name"
  3  FROM employees
  4  WHERE last_name LIKE 'C%'
```

```
  5* ORDER BY last_name

      EMPID Last Name
---------- -------------------------
        187 Cabrio
        148 Cambrault
        154 Cambrault
        110 Chen
        188 Chung
        119 Colmenares

6 rows selected.

SQL> ALTER SESSION SET NLS_SORT=spanish_m;

Session altered.

SQL> SELECT employee_id EmpID,
  2         last_name "Last Name"
  3  FROM employees
  4  WHERE last_name LIKE 'C%'
  5* ORDER BY last_name

      EMPID Last Name
---------- -------------------------
        187 Cabrio
        148 Cambrault
        154 Cambrault
        119 Colmenares
        110 Chen
        188 Chung

6 rows selected.
```

### Oracle Error Messages

The `NLS_LANGUAGE` parameter also controls the language of the database error messages that are returned from the database. Setting this parameter prior to submitting your SQL statement will ensure that local language-specific database error messages will be returned to the application.

Consider the following server message:

```
ORA-00942: table or view does not exist
```

When the `NLS_LANGUAGE` parameter is set to French, the server message appears as follows:

```
ORA-00942: table ou vue inexistante
```

> **See Also:** "Working in a Global Environment" chapter in the *Oracle Database Express Edition 2 Day DBA* for a discussion of globalization support features within Oracle Database.

## Localizing Text on JSP Pages in JDeveloper

Your Java application can make use of resource bundles, to provide different localized versions of the text used on your JSP pages.

Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, such as some text to display on a page, your program can load it from the resource bundle that is appropriate for the current user locale. In this way, you can write program code that is largely independent of the user locale isolating the actual text in resource bundles.

In outline, the resource bundle technology has the following features:

- Resource bundles belong to families whose members share a common base name, but whose names also have additional components that identify their locales. For example, the base name of a family of resource bundles might be `MyResources`. A locale-specific version for German, for example, would be called `MyResources_de`.

- Each resource bundle in a family contains the same items, but the items have been translated for the locale represented by that resource bundle. For example, a `String` used on a button might in `MyResources` be defined as `Cancel`, but in `MyResources_de` as `Abbrechen`.

- You can make specializations for different resources for different countries, for example, for the German language (`de`) in Switzerland (`CH`).

To use resource bundles in your application, you must do the following:

1. Create the resource bundles.

2. In pages that have visual components, identify the resource bundles you will be using on the page.

3. For each item of text you want to display on your pages, retrieve the text from the resource bundle instead of using hard-coded text.

> **See Also:** Sun Microsystems documentation on resource bundles at
>
> http://java.sun.com/j2se/1.4.2/docs/api/java/util/ResourceBundle.html

In the sample application, resource bundles can be used in the following places:

- Headings and labels on JSP pages. In this case, rather than entering text directly on the pages, you can use a scriptlet to find the text.

- Values for buttons and other controls. In this case, set the `value` property of the button to an expression that retrieves the text from the resource bundle

This section covers the following tasks:

- Creating a Resource Bundle

- Using Resource Bundle Text on JSP Pages

## Creating a Resource Bundle

To create a default resource bundle:

1. Create a new Java class called `MyResources.java`, that extends class `java.util.ListResourceBundle`.

2. Define the resource bundle class and methods to return contents as follows:

```
public class MyResources extends ListResourceBundle
{
  public Object[][] getContents() {
    return contents;
  }
```

```
static final Object[][] contents = {
};
}
```

**3.** Add an entry for each item of text you need on your pages, giving a key and the text for that key. For example, in the following example, the comments indicate the strings that must be translated into other languages:

```
static final Object[][] contents = {
  // LOCALIZE THIS
  {"CompanyName", "AnyCo Corporation"},
  {"SiteName", "HR Application"},
  {"FilterButton", "Filter"},
  {"UpdateButton", "Update"},
  // END OF MATERIAL TO LOCALIZE
};
```

The complete resource bundle class should look similar to that shown in Example 9–10.

***Example 9–10    Creating a Resource Bundle Class***

```
public class MyResources extends ListResourceBundle
{
  public Object[][] getContents() {
    return contents;
  }
  static final Object[][] contents = {
    // LOCALIZE THIS
    {"CompanyName", "AnyCo Corporation"},
    {"SiteName", "HR Application"},
    {"FilterButton", "Filter"},
    {"UpdateButton", "Update"},
    // END OF MATERIAL TO LOCALIZE
  };
}
```

To globalize your application, you must create the locale-specific versions of the resource bundle for the different locales you are supporting, containing text for the items in each language.

## Using Resource Bundle Text on JSP Pages

To use the text defined in a resource bundle on your JSP pages:

**1.** Open the JSP page you want to work on in the Visual Editor, such as edit.jsp.

**2.** Create a new line at the top of the page before the first heading and set the **Style** of the line to None. Add a **jsp:usebean** tag to the new line. Enter myResources as the **ID**, and hr.MyResources as the **Class**. Set the **Scope** to session, and click **OK**.

**3.** Drag a **jsp:scriptlet** to the page, where you want the resource bundle text to be displayed, for example in the heading for the page.

In the Insert Scriptlet dialog, enter the script for retrieving text from the resource bundle:

```
out.println(myResources.getString("CompanyName") + ": " +
myResources.getString("SiteName"));
```

**4.** If there was text already displayed in the heading, you can remove it now.

5. If you select the **Source** tab below the Visual Editor, you should see code for the page similar to the following:

```
<h2 align="center">
  <% = myResources.getString("CompanyName") + ": " +
             myResources.getString("SiteName"));
  %>
</h2>
```

6. To use resource bundle text as the label for a button, double-click the button in the Visual Editor. In the button properties dialog, for the **Value** parameter of the button, enter a script similar to the following:

```
<% out.println(myResources.getString("UpdateButton"));%>
```

7. If you view the Source code for the page, you will see code similar to the following:

```
<input type="submit"
  value=<% out.println(myResources.getString("UpdateButton"));%> />
```

If you now run your application, you will see the text you defined in your resource bundle displayed on the page.

# Index

## V

view project, 7-5

## W

Web server, 2-3
   Apache Tomcat, 2-3
   JDeveloper OC4J server, 2-3
   OC4J server, 2-3
   servlet container, 2-3
web.xml, 5-21

## X

X/Open SQL Call Level Interface, 1-1