9

Databases and Tomcat

IN THIS CHAPTER

- Using JDBC and Data Sources
- Direct JDBC Database Access
- Tomcat Data Sources
- Data Access Objects (DAOs)
- Security Considerations

Most Web applications need to store information on a temporary or permanent basis. The most common repository for data storage is the ubiquitous relational database. In this chapter you will be using databases from within Tomcat Web applications using both direct JDBC access and the preferred approach of JNDI data sources. The Data Access Object (DAO) design pattern is shown as a method of encapsulating database access into a reusable component.

Using JDBC and Data Sources

Tomcat servlets and JSPs use JDBC in the same manner as any other Java program. In many Java programs, it is normal to ask the user to provide a username and password for database access. With Web applications, you would typically use a single database account for all users and encode the account name and password within either the servlet or JSP, or encapsulate database access using a data source (see the section "Tomcat Data Sources") or a DAO (see the section "Data Access Objects (DAOs)"). You must then add the business logic of the application to enforce any user security authorization and ensure the data integrity of the database.

The disadvantage to using JDBC from within a servlet is that the JDBC driver class, database connection string, user account name, and password are all hard-coded into the program. The hard-coded database details link the servlet to a specific database, complicating the move from a development environment (with test data) to a production environment (potentially using a different database supplier). Hard-coded details always reduce portability of code and should be avoided if at all possible.

A common approach in reducing coupling of the servlet to the database is to provide the database connection parameters using servlet initialization parameters (see the `<init-params>` element discussion in Chapter 7, "The Web Application Environment"). If in the future you change database vendors, move the database to a new server, move the tables into a new database, change the username, or change the password, you can do so without modifying the Java servlet (or JSP). This is a good technique; but, as you will be shown, there are other performance problems that suggest you shouldn't use direct JDBC access from within a servlet.

A connection pooling data source provides a solution to both the hard coding of database access details and the performance issues inherent in using direct database access. This chapter describes how to use Tomcat data sources after briefly discussing, and rejecting, using direct JDBC access from within a servlet.

Before we describe direct JDBC access and data sources, you need a bit of background information about the database tables and sample data used for the examples.

The Sample Database

Continuing with the currency converter example application, the code examples in this chapter use a currency definition table called `Currency` and a database exchange rate table called `Exchange`. The `Currency` table has columns for the ISO 4217 currency `name` as well as the currency's ISO `country` name and ISO `language`. The `Exchange` table has columns that define the three-character ISO 4217 code for the currency to convert from (`src`) and to (`dst`). The `rate` column defines the appropriate exchange rate. The two tables can be created in any SQL database with the following SQL:

```
create table Currency(
    language varchar(2),
    country varchar(2),
    name varchar(3)
);
create table Exchange(
    src varchar(3),
    dst varchar(3),
    rate double
);
```

You will need to create this table in your database and add some sample data in order to use the examples presented in this chapter. The following SQL will populate the table with suitable data:

```
insert into Currency values ('en','CA', 'CAD');
insert into Currency values ('de','DE', 'EUR');
insert into Currency values ('en','GB', 'GBP');
insert into Currency values ('en','US', 'USD');
```

```
insert into Exchange values ('CAD','EUR', 0.6955);
insert into Exchange values ('CAD','USD', 0.6376);
insert into Exchange values ('CAD','GBP', 0.4344);
insert into Exchange values ('EUR','CAD', 1.4376);
insert into Exchange values ('EUR','GBP', 0.6246);
insert into Exchange values ('EUR','USD', 0.9166);
insert into Exchange values ('GBP','CAD', 2.3019);
insert into Exchange values ('GBP','EUR', 1.6011);
insert into Exchange values ('GBP','USD', 1.4676);
insert into Exchange values ('USD','CAD', 1.5685);
insert into Exchange values ('USD','EUR', 1.0909);
insert into Exchange values ('USD','GBP', 0.6813);
```

TIP

The `CreateDB.java` program available from the accompanying Web site for this book (browse to `http://www.samspublishing.com` and search for the ISBN 0-672-32439-3) can be used to create and populate the sample database tables.

NOTE

The examples in this chapter use the MySQL database available from `http://www.mysql.com`. MySQL is a popular, open source, SQL database available under the GNU General Public License (`http://www.gnu.org/`). Several JDBC drivers are available for MySQL. This chapter uses the MM.MySQL JDBC driver that is also provided under the GNU General Public License and is downloadable from the MySQL Web site. The MM.MySQL JDBC driver class name is `org.gjt.mm.mysql.Driver`. The examples use a JDBC connect string of `jdbc:mysql://localhost/test` to access the MySQL `test` database, which has been configured with a user account called `root`, password `secret`.

Now that you've finished the database configuration, let's take a closer look at using databases with Tomcat.

Direct JDBC Database Access

Any JDBC-compliant database can be used with Tomcat provided the necessary supporting classes are available. If the JDBC driver for a database is provided as a JAR file, this JAR file must be added to the `<CATALINA_HOME>/common/lib` directory; otherwise, the Tomcat 4.1 class loader will not be able to load the driver.

NOTE

The restriction of placing JDBC driver JAR files in `<CATALINA_HOME>/common/lib` applies to the Tomcat 4.1 beta release. Under Tomcat 4.0, you may also store the JAR files in the `WEB-INF/lib` or `<CATALINA_HOME>/lib` directory.

A Simple Database Servlet

The first example program shown in Listing 9.1 is a simple servlet that uses a database to display the exchange rate for converting UK pounds sterling (GBP) to US dollars (USD).

LISTING 9.1 The Simple Database Program `DatabaseRates.java`

```
import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class DatabaseRates extends HttpServlet
{
    public void doGet(HttpServletRequest request,
➥ HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Connection con = null;
        out.println ("<HTML><HEAD><TITLE>Conversion Rates");
        out.println ("</TITLE></HEAD><BODY>");
        out.println("<H1>Conversion Rates</H1>");

        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
➥ "jdbc:mysql://localhost/test","root","secret");
            PreparedStatement pstmt = con.prepareStatement(
➥ "SELECT rate FROM Exchange WHERE src = ? and dst = ?");
            pstmt.setString(1,"GBP");
            pstmt.setString(2,"USD");
            ResultSet results = pstmt.executeQuery();
            if (!results.next())
                throw new SQLException("Missing Exchange rate data row");
            double rate = results.getDouble(1);
            out.println("GBP to USD rate = "+rate+"<BR>");
            pstmt.setString(1,"EUR");
            pstmt.setString(2,"USD");
```

LISTING 9.1 Continued

```
            results = pstmt.executeQuery();
            if (!results.next())
                throw new SQLException("Missing Exchange rate data row");
            rate = results.getDouble(1);
            out.println("EUR to USD rate = "+rate+"<BR>");
        }
        catch (Exception ex)
        {
            out.println("<H2>Exception Occurred</H2>");
            out.println(ex);
            if (ex instanceof SQLException) {
                SQLException sqlex = (SQLException) ex;
                out.println("SQL state: "+sqlex.getSQLState()+"<BR>");
                out.println("Error code: "+sqlex.getErrorCode()+"<BR>");
            }
        }
        finally {
            try { con.close(); } catch (Exception ex) {}
        }
        out.println ("</BODY></HTML>");
    }
}
```

NOTE

The error handling in Listing 9.1 and all other examples in this chapter is designed to aid development and debugging. Chapter 12, "Error Handling," discusses techniques for incorporating user-friendly error handling for a live application.

Although the example in Listing 9.1 works and is thread-safe, there are a large number of problems:

- The JDBC driver, database, username, and password are hard-coded into the program.
- Every HTTP request must open a new connection to the database.
- The database access code and the HTML presentation code are inextricably intermixed within the servlet.

As previously discussed, hard-coded database details always reduce portability of code and should be avoided if at all possible.

Opening a database connection for every request will result in slower performance and may overload the database server because creating new client connections is a resource-intensive operation. In addition, as you have already been shown, the mixing of business logic and data presentation in a single class is a sign of poor system design and makes code maintenance a nightmare. Each of the problems just identified can be resolved using standard tools and design techniques as discussed later in this chapter.

A Bad Example Servlet

Before studying the correct approach to database access using data sources, a common technique suggested by some online tutorials and textbooks will be examined and rejected due to the complications inherent in the approach.

In order to improve performance, you might think of moving the database connection code into the `HttpServlet.init()` method as follows:

```
public class DatabaseRates extends HttpServlet
    implements SingleThreadModel
{
    Connection con;
    PreparedStatement pstmt ;
    public void init() throws ServletException
    {
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
➥ "jdbc:mysql://localhost/test","root","secret");
            pstmt = con.prepareStatement(
➥ "SELECT rate FROM Exchange WHERE src = ? and dst = ?");
        }
        catch (SQLException ex) {
            throw new ServletException(
➥ "Cannot create database connection",ex);
        }
    }
...
```

This is a technique you should not adopt. By attempting to improve performance in this manner, you create a whole range of other potential problems:

- The `Connection` and `PreparedStatement` instance variables are not multi-thread–safe, and the servlet must now implement the `SingleThreadModel`, which will degrade servlet performance and scalability.
- The database connection is kept open for the lifetime of the servlet instance and this may affect database performance and scalability. If you adopt the same approach for many servlets, you will use up large numbers of database connections for servlets that are idle for a lot of the time. You may even have to reconfigure your database to support an unusually large numbers of connections.
- A database may time out a connection that is kept open for too long or one that remains idle for specified period of time. If this happens, you will have to add code to your servlet to deal with a closed connection, further complicating the logic of your servlet.

You can address these problems by using the servlet `init()` method to cache the exchange rate in an instance variable when the servlet is first accessed. All subsequent accesses will use the cached rate rather than access the database. But this approach cannot handle the real-world situation where the exchange rate varies over time. To solve this problem, the servlet must periodically update the cached exchange rate from the master value stored in the database. You then have the problem of deciding on a suitable algorithm for updating the cached exchange rate and adding additional database access code. This adds complications to the servlet that can be avoided simply by using connection pooling.

Connection Pooling

To improve database performance, you should use a technique known as *connection pooling*. This approach involves implementing a broker class that encapsulates access to the database. Typically the connection broker class has a `getConnection()` method that returns a proxy object that can partake in connection pooling. In other words, the connections do not connect directly to the database but will share a pool of available connections. New connections are added to the pool if demand exceeds current capacity, and excess connections can be closed down during idle periods.

As far as your written code is concerned, the connection behaves like a normal JDBC connection but the connection broker manages the pool of actual connections to the database.

Connection pooling is a well-understood technique, and many JDBC driver suppliers provide connection pooling implementations with their drivers. The J2EE specification has extended the JDBC support to include the `DataSource` class that may be used to support connection pooling in a portable manner.

Tomcat has adopted the J2EE specification and uses the `javax.sql.DataSource` class to provide connection pooling as discussed in the next section, "Tomcat Data Sources."

NOTE

If you want to use connection pooling outside of Tomcat, or do not want to use the Tomcat data source implementation, then there are several open-source connection pooling implementations available from the Internet. A popular, freely available broker that works with any JDBC driver is `DbConnectionBroker`, which is available from `http://www.javaexchange.com`.

Tomcat Data Sources

The recommended approach for accessing databases from Tomcat is to use the Database Connection Pool (DBCP) connection broker incorporated into Tomcat. DBCP is part of the Jakarta commons subproject that can be found at `http://jakarta.apache.org/commons`. DBCP has many advantages for Web application developers:

- It supports connection pooling.
- It manages the database driver and connection URL information.
- It is J2EE compliant.
- It is a standard component of Tomcat.

Not all beta releases of the Tomcat 4.1 archive included the DBCP package. If the DBCP package is not included in your Tomcat 4.1 archive, you will need to download and install DBCP yourself. If your Tomcat archive contains the file `<CATALINA_HOME>/common/lib/commons-dbcp.jar`, DBCP is included with Tomcat.

If DBCP is not included with your Tomcat archive, download it from `http://jakarta.apache.org/builds/jakarta-commons`. At the time of writing this book (mid-2002), DBCP had not yet reached the release milestone build and must be downloaded from the latest nightly build at `http://jakarta.apache.org/builds/jakarta-commons/nightly/commons-dbcp/`. If a release version is available from `http://jakarta.apache.org/builds/jakarta-commons/release/commons-dbcp/`, you should use it in preference to a nightly build.

Download and extract the `commons-dbcp.tar` archive and copy the `commons-dbcp.jar` to `<CATALINA_HOME>/common/lib`.

DBCP uses the Jakarta Commons Pool package. At the time of this writing, the commons-pool package was at release 1.0, if a later release is available you should use it.

The commons-pool-1.0.tar archive must be downloaded from http://builds/jakarta-commons/release/commons-pool/. Move the commons-pool.jar from the extracted archive to <CATALINA_HOME>/common/lib.

After the commons-dbcp.jar and commons-pool.jar files are stored in <CATALINA_HOME>/common/lib, you can use the Tomcat data sources as described in the rest of this section. If the DBCP classes are not included with Tomcat, you will receive a "javax.naming.NamingException: Cannot create resource instance" error when accessing the data source.

NOTE

Tomcat 4.0 uses a third-party connection broker called Tyrex (see http://www.tyrex.com) which has been replaced by the DBCP connection broker. Tyrex has the same functionality and advantages as DBCP and is included with the Tomcat 4.0 download.

A minor complexity to using DBCP is that it uses the Java Naming Directory Interface (JNDI), and you have to configure the JNDI data source before you can use it. Here you will be shown how to define a JDBC data source without any unnecessary background information about JNDI.

To use a Tomcat data source you will have to

- Define a JNDI resource reference in your Web application deployment descriptor
- Map the JNDI resource reference onto a real resource (database connection) in the context of your application
- Look up the JNDI data source resource reference in your code to obtain a pooled database connection

Defining a Resource Reference

First, you will need a JNDI name for your database connection; conventionally it should begin with jdbc/. The example uses the name jdbc/conversion. Now add a <resource-ref> element to the web.xml file for your Web application to define the data source as follows:

```
<resource-ref>
  <res-ref-name>jdbc/conversion</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

The <res-ref-name> element identifies the resource reference name, the <res-type> entry defines the reference as a JDBC data source, and the <res-auth> specifies that resource authentication is applied by the Container. The <resource-ref> element must be added after any <servlet-mapping> elements in the deployment descriptor.

Defining the Resource

Second, you must add an entry for the database resource to the <CATALINA_HOME>/conf/server.xml file by adding a <ResourceParams> element to define the database connection information. The <ResourceParams> can be defined inside the <DefaultContext> tag to be available to all Web applications, or it can be defined inside the <Context> element for a specific application.

A suitable Tomcat 4.1 <ResourceParams> element for the example database is shown in the following listing (Tomcat 4.0 uses different attribute names as explained in the notes):

```
<Context path="/database" docBase="database" debug="0"
         reloadable="true" >
  <ResourceParams name="jdbc/conversion">
    <parameter>
      <name>username</name>
      <value>root</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>secret</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>org.gjt.mm.mysql.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost/test</value>
    </parameter>
  </ResourceParams>
</Context>
```

In the <ResourceParams> element example, the name attribute must exactly match the <res-ref-name> value you defined in the <resource-ref> element of the web.xml file.

NOTE

As an alternative to using resource references in an application's `web.xml` file, you can define the resource using a `<Resource>` element in the `server.xml` file. You can add a `<resource>` element to the `<Context>` element for an individual application or to the `<DefaultContext>` element to define the resource for all Web applications. The `<Resource>` definition for the example JDBC data source is

```
<Resource name="jdbc/conversion"
          auth="Container"
          type="javax.sql.DataSource"/>
```

The various `<ResourceParams>` components define the database connection parameters as follows:

Parameter	Value
`username`	Username for the database (`root`); under Tomcat 4.0 this attribute is called `user`
`password`	Password for the user root (`secret`)
`driverClassName`	Driver class for the database connection (`org.gjt.mm.mysql.Driver`)
`url`	JDBC connection string for the database (`jdbc:mysql://localhost/test`); under Tomcat 4.0 this attribute is called `driverName`

CAUTION

The `server.xml` file contains the unencrypted password for accessing the database. This file should be secured so that unauthorized users cannot read it.

The `<ResourceParams>` element in the `server.xml` file encapsulates all the information required to access the database. When moving your application from a development environment to a live environment, you need only change the values for the parameters in the `server.xml` file. Neither the Java code nor the `web.xml` resource reference entry need to be modified.

Using Tomcat data sources decouples the database vendor connection details from the servlet code and gives you the performance benefits gained from using a connection pool. Put simply, always use Tomcat's data sources when accessing a database.

CAUTION

When using data sources with both Tomcat 4.1 and Tomcat 4.0, you must place the JDBC JAR file for your database driver in the `<CATALINA_HOME>/common/lib` directory; otherwise, Tomcat will not be able to load the JDBC driver.

Obtaining the Data Source Connection

When you are using resource references in your application, you must use JNDI to look up and access the resource. You will have to import the `javax.naming` package to use JNDI. Note that the JNDI methods shown below can throw a `javax.naming.NamingException` error.

The name you defined in the `<res-ref-name>` element is the JNDI name you must use in your code. In order to resolve the resource associated with this name, you must obtain the JNDI context for your Web application. Use the `javax.naming.Context.lookup()` method to look up the data source and obtain a database connection as shown in the following code:

```
Context init = new InitialContext();
Context ctx = (Context) init.lookup("java:comp/env");
DataSource ds = (DataSource) ctx.lookup("jdbc/conversion");
Connection con = ds.getConnection();
```

Use this `Connection` object just like a normal JDBC connection. Listing 9.2 shows the simple database example rewritten to use the JNDI data source shown in the previous examples.

LISTING 9.2 Using a `DataSource` Object in `DataSourceRates.java`

```
import java.io.*;
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class DataSourceRates extends HttpServlet
{
    private DataSource dataSource;
```

LISTING 9.2 Continued

```
public void init(ServletConfig config) throws ServletException {
    try {
        Context init = new InitialContext();
        Context ctx = (Context) init.lookup("java:comp/env");
        dataSource = (DataSource) ctx.lookup("jdbc/conversion");
    }
    catch (NamingException ex) {
        throw new ServletException(
            "Cannot retrieve java:comp/env/jdbc/conversion",ex);
    }
}

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Connection con = null;
    out.println ("<HTML><HEAD><TITLE>Conversion Rates</TITLE></HEAD><BODY>");
    out.println("<H1>Conversion Rates</H1>");

    try {
        synchronized (dataSource) {
            con = dataSource.getConnection();
        }
        PreparedStatement pstmt = con.prepareStatement(
            "SELECT rate FROM Exchange WHERE src = ? and dst = ?");
        pstmt.setString(1,"GBP");
        pstmt.setString(2,"USD");
        ResultSet results = pstmt.executeQuery();
        if (!results.next())
            throw new SQLException("Missing Exchange rate data row");
        double rate = results.getDouble(1);
        out.println("GBP to USD rate = "+rate+"<BR>");
        pstmt.setString(1,"EUR");
        pstmt.setString(2,"USD");
        results = pstmt.executeQuery();
        if (!results.next())
            throw new SQLException("Missing Exchange rate data row");
        rate = results.getDouble(1);
        out.println("EUR to USD rate = "+rate+"<BR>");
    }
```

LISTING 9.2 Continued

```
        catch (Exception ex)
        {
            out.println("<H2>Exception Occurred</H2>");
            out.println(ex);
            if (ex instanceof SQLException) {
                SQLException sqlex = (SQLException) ex;
                out.println("SQL state: "+sqlex.getSQLState()+"<BR>");
                out.println("Error code: "+sqlex.getErrorCode()+"<BR>");
            }
        }
        finally {
            try { con.close(); } catch (Exception ex) {}
        }
        out.println ("</BODY></HTML>");
    }
}
```

NOTE

To recap, to run the example in Listing 9.2, you must have defined the `<resource-ref>` element in `WEB-INF/web.xml` and the `<ResourceParams>` element in `<CATALINA_HOME>/conf/server.xml`. Furthermore, you must place your JDBC JAR file in the directory `<CATALINA_HOME>/common/lib` so the Tomcat class loader can load the driver correctly. You will need to stop and restart Tomcat in order for your changes to the `server.xml` file to be recognized.

In Listing 9.2, all the data source lookup code is added to the `init()` method to avoid the costly JNDI operations for every HTTP request. Because the `dataSource` instance variable is potentially shared across multiple threads, access to the variable must be from within a synchronized block.

The example in Listing 9.2 is an improvement over Listing 9.1, but it still has problems. The intermingling of HTML presentation code and database access code is less than ideal. You will now be shown how to refactor the example by using the Java design pattern (or idiom) called a Data Access Object.

Data Access Objects (DAOs)

A Data Access Object (DAO) encapsulates access to a database so that data manipulation code can be separated out from other business logic and data presentation code.

Good use of DAOs will simplify the design and development of Web applications and reduce the cost of ongoing maintenance and upgrades.

DAOs are a Java design pattern where all the database access code is encapsulated in supporting Java classes. Changes to the database details such as the table schema (column names and types) will usually only affect the DAO and not the Java code using the DAO. Combining the DAO with a Tomcat data source provides a good solution to using a database with Tomcat.

Listing 9.3 shows a JSP that uses a Java Bean DAO to display a table of currency exchange rates.

LISTING 9.3 The `rates.jsp` Page Using a DAO

```
<jsp:useBean id="dao" class="converters.ConversionDAO" scope="page" >
    <jsp:setProperty name="dao" property='*'/>
</jsp:useBean>
<% dao.updateRate(); %>
<HTML>
  <HEAD><TITLE>Currency Conversion Rates</TITLE></HEAD>
  <BODY>
    <H1>Conversion Rates</H1>
    GBP to USD rate = <%= dao.selectRate("GBP","USD") %><BR>
    EUR to USD rate = <%= dao.selectRate("EUR","USD") %><BR>
    EUR to GBP rate = <%= dao.selectRate("EUR","GBP") %><BR>
    <H1>Update Rate</H1>
    <FORM>
        <TABLE>
            <TR>
                <TD>Convert from:</TD>
                <TD><SELECT NAME='src'>
                    <OPTION>EUR<OPTION>GBP<OPTION>USD
                    </SELECT></TD>
                <TD>to:</TD>
                <TD><SELECT NAME='dst'>
                    <OPTION>EUR<OPTION>GBP<OPTION>USD
                    </SELECT></TD>
            </TR>
            <TR>
                <TD>Rate:</TD>
                <TD COLSPAN='3'><INPUT TYPE='text' NAME='rate'></TD>
            </TR>
        </TABLE>
        <P> <INPUT TYPE='submit' VALUE='Set new Rate'> </P>
```

LISTING 9.3 Continued

```
    </FORM>
  </BODY>
</HTML>
```

The first thing that should strike you about Listing 9.3 is the absence of complex Java code and any database access code. The beginning of the page creates the DAO as a Java Bean and sets the properties of the DAO that match the HTTP request parameters by using the element

```
<jsp:setProperty name="dao" property='*'/>
```

The DAO has been designed so that the properties reflect the three columns of the underlying `Exchange` table (`src`, `dst`, and `rate`). If the HTTP request parameters include these three properties, the DAO's `updateRate()` method will update the underlying `Exchange` table with a new exchange rate. The line

```
<% dao.updateRate(); %>
```

will update an existing exchange rate with parameters supplied with the HTTP request.

The main part of the JSP uses the DAO bean to display the exchange rates for three currencies (EUR, GBP, and USD) and displays a simple form for updating a conversion rate. Filling in a new conversion rate and submitting the form will invoke this JSP to update the database (using the `<% dao.updateRate(); %>` action coded at the top of the page).

The DAO is not complicated, as shown in Listing 9.4.

LISTING 9.4 The `ConversionDAO.Java` Data Access Object

```
package converters;

import java.io.*;
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
import java.util.*;

public class ConversionDAO
{
    private Connection con;
    private PreparedStatement select;
```

LISTING 9.4 Continued

```
    private PreparedStatement update;
    private String src;
    private String dst;
    private double rate;

    public ConversionDAO() throws SQLException, NamingException
    {
        Context init = new InitialContext();
        Context ctx = (Context) init.lookup("java:comp/env");
        DataSource ds = (DataSource) ctx.lookup("jdbc/conversion");
        con = ds.getConnection();
        select = con.prepareStatement(
➥ "SELECT rate FROM Exchange WHERE src = ? AND dst = ?");
        update = con.prepareStatement(
➥ "UPDATE Exchange SET rate = ? WHERE src = ? AND dst = ?");
    }

    public String getSrc() { return src;}
    public String getDst() { return dst;}
    public double getRate() { return rate;}
    public void setSrc(String src) { this.src = src;}
    public void setDst(String dst) { this.dst = dst;}
    public void setRate(double rate) { this.rate = rate;}

    public double selectRate(String src, String dst) throws SQLException
    {
        ResultSet results = null;
        try {
            select.setString(1,src);
            select.setString(2,dst);
            results = select.executeQuery();
            if (!results.next())
                throw new SQLException("Missing Exchange rate data row");
            return results.getDouble(1);
        }
        finally {
            try { results.close(); } catch (Exception ex) {}
        }
    }

    public int updateRate() throws SQLException
    {
```

LISTING 9.4 Continued

```
        if (src==null || dst==null)
            return 0;
        update.setDouble(1,rate);
        update.setString(2,src);
        update.setString(3,dst);
        return update.executeUpdate();
    }

    public void close()
    {
        try { select.close(); } catch (Exception ex) {}
        try { update.close(); } catch (Exception ex) {}
        try { con.close(); } catch (Exception ex) {}
    }
}
```

For simplicity, the examples in Listings 9.3 and 9.4 exclude any error-handling code. In Chapter 12, you will be shown how to use Web application error pages and other error-handling techniques that will enable you to enhance the JSP to deal gracefully with any underlying database or JNDI problems.

NOTE

You will need to deploy any DAO classes to the `WEB-INF/classes` directory together with the JSPs in your Web application.

An alternative technique for accessing a database from within a JSP is to use a custom tag to encapsulate the database access (see Chapter 11, "JSP Expression Language and JSTL").

That's it. You now know how to work with DAOs and data sources to access a JDBC database.

Security Considerations

Your main security problem concerns the coding of plain-text passwords in the `server.xml` file. You must ensure that only authorized users (ideally only the Tomcat administrator) can read the `server.xml` file.

One minor headache occurs if you configured Tomcat to use a security manager (see Chapter 14, "Access Control"). If you are accessing a database with a security manager, you will need to add appropriate permission lines to the `<CATALINA_HOME>/conf/catalina.policy` file to allow the Web application to connect to the database. You will need to add the following lines to the `catalina.policy` file (assuming the MySQL database is running on `localhost` using the default 3306 port):

```
grant codeBase "file:${catalina.home}/webapps/database/WEB-INF/classes/- {
    permission java.net.SocketPermission "localhost:3306", "connect";
};
```

If you use a different database, you will need to amend the database hostname and port number specified for the `java.net.SocketPermission` permission to match your database server configuration.

Tomcat does not use a security manager by default, so you will normally not have permission problems with database access.

Summary

Using a database with Tomcat is straightforward so long as you apply the following simple, well-known design criteria:

- Do not access the database directly from the servlet or JSP but encapsulate the database access in a Data Access Object (DAO) helper class. By putting all the database access code into a single class that is separate from the business and presentation logic of your application, you will simplify application maintenance and future feature enhancements.
- To obtain best performance, you should use a connection broker class that implements database connection pooling. Tomcat includes the DBCP connection broker that is compatible with any JDBC driver.

DBCP uses a JNDI resource reference to separate database connection details from database data access. This separation of database administration information is an invaluable aid in managing the deployment of a Web application in different environments. Moving an application from development, to test, to production environments is a simple administrative issue that will not require changes to the Java code of the application.