**MODULE 8**

**Connecting databases with Tomcat applications**

Many of the web applications running on your server will process data, and most of that data will be stored in a database. The most popular databases, including MySQL, PostgreSQL, Oracle, SQL Server, Sybase, Interbase, and DB2, are based on relational concepts. You need to understand how Tomcat interacts with relational databases to better understand the requirements of your users.

In this module, you’ll see the many situations that will arise when configuring Tomcat to work with relational databases. More important, you’ll gain some hands-on experience configuring several examples. By the end of this chapter, you’ll be comfortable integrating databases with Tomcat.

**Introducing SQL**

The Structured Query Language (SQL) is a text-based query language used to perform operations with data stored in a relational database. These operations include selecting data for display, inserting data into the database, deleting data from the database, and manipulating database structure.

As an administrator, you won’t see much SQL in your Tomcat setup, because the server uses it behind the scenes to look up usernames and passwords. You will, however, have to be fairly familiar with simple SQL commands so that you can add and remove users from your Tomcat user databases. Because these are realm-related commands.

**Introducing JDBC**

JDBC is a Java programming interface for accessing databases, which makes it the obvious choice when using databases with Tomcat. JDBC submits SQL query statements to the remote SQL processing engine (part of the database that handles multiple, simultaneous connections via a connection manager), and the SQL processing engine returns the result of the query in a set of data called a result set. A result set is typically zero or more rows of data. You can think of result sets as temporary database tables.

Therefore, JDBC operations are designed to do the following:

* Take the JDBC API calls and transform them into a SQL query.
* Submit that query to the SQL processing engine on the database.
* Retrieve the result set that’s returned from the query and transform it into a Java- accessible data structure.

Not all statements return a result set; you may conduct a search that isn’t successful, so the returned result set will be empty (called a null result set). In addition, some SQL statements, such as those you use to create tables, update data, and delete rows, don’t return any result sets.

**Running Basic JDBC Operations**

In JDBC programming, a developer must follow these typical steps:

1. Obtain a connection to the remote database server.
2. Create and prepare a SQL statement for execution (or call a stored procedure in the database).
3. Execute the SQL statement.
4. Obtain the returned result set (if any) and work on it.
5. Disconnect from the remote database.

It’s usually the case that you’ll be concerned only with the first step of this process. Once you’ve connected the web server to a database, you hand the connection over to any web applications that need it.

**Establishing and Terminating Connections to Databases**

Other than providing a unified way of accessing, modifying, and manipulating data in databases, JDBC also provides a unified way of connecting to databases from different vendors. While normal native connections to Oracle will be different from connections to MySQL, which will be different yet from working with Microsoft’s SQL Server, connecting to any of these databases can be accomplished using the same JDBC API calls.

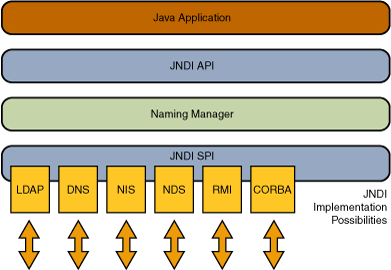
As you saw in Module 4, JDBC, like JNDI, is simply a layer of abstraction between the native interfaces and Java. In Figure 8-1, you can see how the underlying database can be changed without the application having to worry about changing any of its settings. You have to worry about only one interface with JDBC, and JDBC deals with talking to the database in its own language.

Figure 8-1. JDBC driver structure

JDBC uses JDBC drivers to communicate with the native database drivers, and these vary from database to database. However, application code doesn’t notice the differences among drivers, because they all follow the same standard.

**Which JDBC Version?**

Under the JDBC 1.0 standard, the code to establish a connection to a database, as well as the code to disconnect from the database, is written by the developer. In fact, even the code to select and activate a JDBC driver is coded by the developer.

Although simple and straightforward to code, this approach creates a problem. In some cases where the driver is written by the developer, the database access code works only with a specific database from the vendor. This makes it difficult to swap to a database from another vendor and removes many of the advantages of JDBC described in the previous section.

JDBC 2.0 relaxes this restriction and introduces the concept of a data source, which maps a name to a set of values for obtaining a database connection. A developer can obtain a connection to a data source using its name, allowing the same JDBC code to work with drivers from any vendor. Meanwhile, you can switch database vendor support by configuring a different data source. The name remains the same, but the settings have changed, so the developer doesn’t need to change the code.

While data sources and connection pooling (covered in the “Database Connection Pooling” section) open new possibilities for database users, JDBC 2.0 doesn’t specify how these features should be used. As a result, many architectural issues are left for the JDBC driver writer to solve— and code can quickly become vendor specific again (this time depending on the JDBC driver vendor).

JDBC 3.0 is the first specification that clearly spells out the different architectures that JDBC can operate in, including two-tier and three-tier models. The three-tier model corresponds to the application-server model and the model of operation favored by Java Enterprise Edition (EE) 1.4 applications.

The specification also attempts to accommodate JDBC 1.0 and 2.0 drivers and models of operations, while formalizing JNDI as the preferred way for applications to obtain a data source. It also formalizes connection pooling as a value-added service of the application server or servlet container. Tomcat uses the Jakarta Commons Database Connection Pools (DBCP) component to implement JDBC 3.0, and all data sources configured in server.xml are JDBC 3.0 data sources (provided you’re using JDK 1.4 or newer—which you will be for Tomcat 6 anyway).

Regardless of the JDBC version, the JDBC driver still has to translate the JDBC commands into native commands to connect to the different databases. Most JDBC drivers are high- performance Type IV drivers (explained in the next section). However, some legacy systems will support only the older Type I to Type III drivers. It’s a good idea to gain some familiarity with the different types of JDBC driver.

**Examining JDBC Driver Types**

Four types of JDBC drivers exist; in general, higher driver type numbers represent an improve- ment on performance:

**Type I**: These are the most primitive JDBC drivers because they’re just data access adapters. They adapt another data access mechanism (such as ODBC) to JDBC. These drivers rely completely on the other data access mechanism and thus have double the administrative and maintenance problems. These drivers are also typically hardware/operating-system specific (because of the data access mechanism that they depend on), meaning they aren’t portable at all.

**Type II**: These are partially written in Java and partially written in native data-access languages

(typically C or C++). The non-Java portion of these drivers limits the portability of the final code and platform migration possibilities. The administrative and maintenance burdens of Type I still exist.

**Type III**: These are pure Java drivers on the client side, which gives them the portability benefit of Java. However, they rely on an external middleware engine to operate. The client code communicates with the middleware engine, and the engine talks to the different types of database. The administration and maintenance burden is somewhat reduced but is far from eliminated.

**Type IV**: These are 100 percent Java client drivers that talk directly to database network protocols. This results in the highest performance connection and the most portable application code. Administration and maintenance is greatly simplified (only the driver needs to be updated).

Fortunately, all the major databases have Type IV JDBC drivers available, either through the database vendors themselves or via a third-party driver vendor.

**Database Connection Pooling**

When a web application accesses a remote database, it may do so through a JDBC connection. Typically, a physical JDBC connection is established between the client application and the database server via a TCP/IP connection. Establishing such a connection is CPU and time inten- sive. It involves multiple layers of software and the transmission and receipt of network data. A typical physical database connection may take seconds to establish.

Some web applications consist of JSP pages and servlets that may need data from a data- base on every HTTP request. For example, an online library application will undoubtedly allow users to search the library catalog. On a heavily loaded server, the time it takes to establish, disconnect, and reestablish physical connections can substantially slow web application performance.

To create high-performing, scalable web applications, JDBC driver vendors and application servers are incorporating database connection pooling into their products. Connection pooling reduces expensive connection establishment time by creating a pool of physical connections when the system starts. When an application requires a connection, one of these physical connections is provided. Normally, when the application finishes using the connection, it would be disconnected. However, in the case of connection pooling, it’s merely returned to the pool where it awaits the next application request.

**Using Tomcat and JDBC**

Tomcat provides valuable services for hosted web applications that use JDBC connections. More specifically, Tomcat will enable running web applications to do the following:

* Access JDBC data sources using standard JNDI lookup
* Use a connection pooling service

**Providing JDBC Data Sources in Tomcat**

You configure JDBC drivers as JNDI resources in Tomcat. These resources are made available during web application run time via standard JNDI lookups. The steps are as follows:

1. A web application obtains a JNDI initial context from Tomcat; it then performs a lookup on the JDBC data source by name.
2. Tomcat handles the JNDI lookup by consulting the configuration files (the context XML file and web.xml) to determine the JDBC driver to use for obtaining a data source. Tomcat will also pool the physical connections made.

Even though no true JNDI-compatible directory services are involved, the Tomcat container emulates the action of a JNDI provider. This enables code that uses JNDI as the JDBC data source lookup mechanism to work within the Tomcat container.

**Configuring JNDI JDBC Resources**

Using JNDI resources in Tomcat to configure JDBC data sources is the recommended way to provide web applications with access to JDBC connections. While other methods are possible— you’ll see at least one alternative later—this approach will lead to portable code and easily maintainable Tomcat servers.

You must perform the following steps to configure JNDI resource for a JDBC data source:

1. Add *<Resource>* and *<ResourceParams>* tags in the *<Context>* element of the context XML file or in a *<DefaultContext>* subelement of the Tomcat 7.0.x *<Host>* element.
2. Ensure that the application developer has defined a <resource-ref> element, corresponding to the previous *<Resource>*, in the web.xml file of the web application using the JDBC resource.

**Using the Resource and ResourceParams Elements**

The *<Resource>* element specifies the JNDI resource that represents a JDBC data source, and the *<ResourceParams>* element configures the associated data source factory. Listing 8-1 shows you how to configure a data source.

**Listing 8-1.** *Defining a JDBC Data Source for Tomcat*

*<Context path="/tomcatBook"*

*docBase="tomcatBook" crossContext="false" debug="0" reloadable="true" >*

*<Resource name="jdbc/CatalogDB" auth="SERVLET" type="javax.sql.DataSource" driverClassName="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/catalog" username="kmittal"*

*password="pas44word" maxActive="30" maxIdle="20000" maxWait="120"/>*

*</Context>*

Both of these settings create a JNDI resource that the web application can access from the context *java:comp/env/jdbc/CatalogDB*. The web application can then use this context to look up the data source. The type of resource that will be returned during this lookup is a *javax.sql.DataSource*. It also specifies that the servlet should authenticate against the data- base on behalf of the web application.

The actual names and values of the parameters depend on the data source connection factory that’s used. The previous settings assume you’re configuring the default DBCP factory. The DBCP factory will work with JDBC drivers for any database and return a data source as appropriate.

**Transactions and Distributed Transactions Support**

Databases offer varying levels of support for transactions. A transaction is a unit of work composed of multiple operations; it can be committed only once all its operations complete successfully. If any of the constituent operations fail, the transaction is rolled back.

When a transaction involves work that crosses multiple physical databases, it’s called a distributed transaction. One standard that enables databases from different vendors to participate in the same distributed transaction is called XA. In the XA operation model, an external transaction manager coordinates a two-phase commit protocol between multiple resource managers (databases in this case). The two-phase commit protocol ensures that the pieces of work, scattered across multiple physical databases, either are all completed or are all rolled back.

JDBC 3.0 accommodates data sources that support XA operations. Administrators who work with XA data sources and data source factories should consult the vendor’s documentation to ensure they work with Tomcat.

**Testing JNDI Resource Configuration**

Here, you’ll work through an actual example and configure a DBCP data source with a Type IV JDBC driver. You’ll base your example on MySQL, as it’s easily available and widely used.

■Note Installing and configuring MySQL is beyond the scope of this course

This chapter will assume that you have MySQL already configured and tested and that you have an account with privileges to create tables and add records to create the test data- base. The latest version of MySQL is available for download from [www.mysql.com.](http://www.mysql.com/)

The Type IV JDBC driver you’ll use is the Connector/J driver from MySQL. This driver is open source and is widely used by the MySQL community. You can download the latest version of the driver from [http://dev.mysql.com/downloads/connector/j/5.0.html.](http://dev.mysql.com/downloads/connector/j/5.0.html)

You must unzip the driver JAR from the download and use the binary JAR file. Place this file under CATALINA\_HOME/common/lib so that the common class loader can make it available to Tomcat and all its web applications.

**Creating the MySQL Test Database**

First, you’ll need to create the database you’ll use. This is a database of products available to buy online and will correspond to the database defined in Listing 10-1 previously.

Listing 10-2 shows a SQL script that will create and populate the catalog database. It’s unlikely you’ll have to create tables such as this normally, but it’s a useful instructional exercise.

Listing 8-2. The createCatalogDB.sql script

*DROP TABLE IF EXISTS product;*

*CREATE TABLE product ( prodid int not null, prodname varchar(30), proddesc varchar(150), price double(7,2)*

*);*

*INSERT INTO product VALUES ( 1,'Yo-Yo','High-quality wooden yo-yo with your company name and logo imprinted on both sides.', 3.50);*

*INSERT INTO product VALUES ( 2,'Slinky',*

*'Plastic slinky in the color of your choice with your company logo imprinted on closed slinky.',*

*0.75);*

*INSERT INTO product VALUES ( 3,'Envelope Cutter','Small cutting tool for opening envelopes. Your company logo is imprinted on handle.', 1.25*

*);*

*INSERT INTO product VALUES ( 4,'Padfolio','Synthetic leather padfolio with company name and logo imprinted on cover.',9.50);*

*INSERT INTO product VALUES ( 5,'Fountain Pen','Attractive fountain pen sporting your company name on the cap.',1.20);*

*INSERT INTO product VALUES ( 6,'Keychain','Rubber keychain with your company name and logo imprinted in a variety of colors.', 0.50);*

*INSERT INTO product VALUES ( 7,*

*'Ruler',*

*'Wooden ruler with raised lettering containing your company name and logo.',*

*0.25);*

*INSERT INTO product VALUES ( 8,'Flashlight','Metal flashlight in a variety of colors. Your company name and logo is imprinted on the handle.', 5.0);*

*Use createCatalogDB.sql to create the database as follows:*

*> mysql < createCatalogDB.sql*

Now that you have the tables, you need to create a user that the developers will use to access the data in the database. Since your web application functionality requires only read access to the data, you’ll create a read-only user for developer access. This will ensure that data can’t be accidentally or maliciously modified or altered.

**Setting Up the Read-Only User**

If you don’t have privileges as the database system administrator, you’ll need to seek help from the database administrator. To give a user read-only privilege on the catalog database, use the following:

*mysql> GRANT SELECT ON catalog.\**

*-> TO 'tester'@'localhost'*

*-> IDENTIFIED BY 'pa$$word';*

The developer may now use this user to access the data in the table, since in this exam- ple, they won’t perform any modifications to the underlying data. This is the user you saw in Listing 10-1.

**Adding the JDBC JNDI Resource to the Server**

You saw the context XML file for this example in Listing 8-1, so now you have to configure the web application’s settings. This is usually the developer’s job, but you’re filling both roles for this example. Remember that DBCP connection pooling is automatically set up. Now edit the tomcatBook web application’s web.xml file, as shown in Listing 8-3.

**Listing 10-3.** *A* <resource-ref> *in the* tomcatBook *Web Application’s* web.xml

*<!-- Describe a DataSource -->*

*<resource-ref>*

*<description>*

*Resource reference to a factory for java.sql.Connection instances that may be used for talking to a*

*particular database that is configured in the tomcatBook.xml file.*

*</description>*

*<res-ref-name>*

*jdbc/CatalogDB*

*</res-ref-name>*

*<res-type> javax.sql.DataSource*

*</res-type>*

*<res-auth> SERVLET*

*</res-auth>*

*</resource-ref>*

This *<resource-ref>* makes the *jdbc/CatalogDB* context available to the web application via JNDI APIs.

**Using JNDI to Look Up a Data Source**

Finally, the developer will look up the data source and start querying the database. The JSP page in Listing 8-4, lookup.jsp, will do exactly that. Put it into the CATALINA\_HOME/webapps/ tomcatBook/ch10 directory. Pay special attention to the way JNDI is used to obtain the data source in the <sql:setDataSource> tag.

**Listing 10-4.** lookup.jsp *Uses a Data Source to Obtain Data*

[*<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core\_rt"*](http://java.sun.com/jstl/core_rt) *%>*

[*<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql\_rt"*](http://java.sun.com/jstl/sql_rt) *%>*

*<sql:setDataSource dataSource="jdbc/CatalogDB"/>*

*<sql:query var="products"> SELECT \* FROM product*

*</sql:query>*

*<html>*

*<head>*

*<title>Online Products</title>*

*</head>*

*<body>*

*<center>*

*<h1>Products</h1>*

*</center>*

*<table border="1" align="center">*

*<tr>*

*<th>Name</th><th>Description</th><th>Price</th>*

*</tr>*

*<c:forEach items="${products.rows}" var="row">*

*<tr>*

*<td><c:out value="${row.prodname}" /></td>*

*<td><c:out value="${row.proddesc}" /></td>*

*<td><c:out value="${row.price}" /></td>*

*</tr>*

*</c:forEach>*

*</table>*

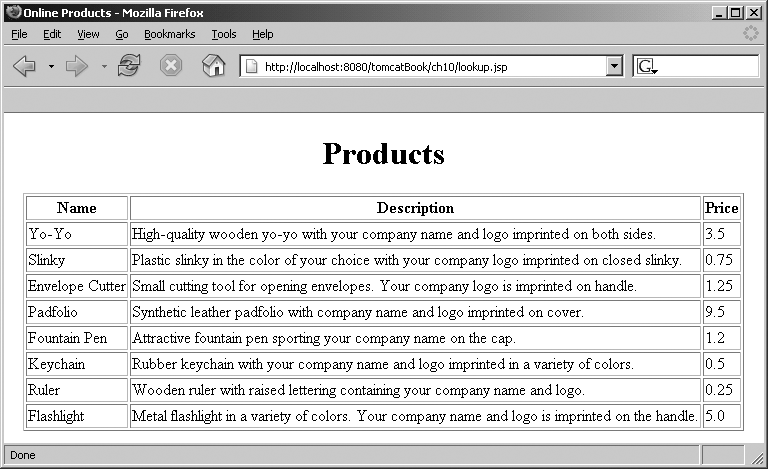
*</body>*

*</html>*

To run this example, you need to add the JSP 1.1 standard tag library (from http:// jakarta.apache.org/taglibs/) to the web application’s classpath, either by placing jstl.jar and standard.jar in tomcatBook/WEB-INF/lib or in the common or shared class loader path.

The *<sql:setDataSource>* tag uses the *jdbc/CatalogDB* context to look up the JNDI resource and makes it available to the page. Behind the scenes, it’s used to create a connection (actually pooled through DBCP). The JSP page then performs a SELECT \* query on the product table and creates an HTML table containing all the table rows.

Connect to http://localhost:8080/tomcatBook/ch8/lookup.jsp. This will compile and execute the JSP code. If everything is configured correctly and working, you should see the page as shown in Figure 8-2.



**Figure 8-2.** *A JSP page that uses a JDBC data source to obtain data*

**Summary**

In this MODULE, you saw JDBC connectivity in the context of Tomcat. The most obvious interaction is the need of web applications to connect to relational database sources.

We discussed Java’s support for accessing databases in the form of JDBC and covered the JDBC version evolution, and I talked briefly about the different types of JDBC drivers that are available. Next, you saw the recommended way of providing a JDBC data source to web applications, which involved the configuration of JNDI resources in the Tomcat configuration file. In addition, Tomcat also provides a database connection pooling service through the Jakarta Commons DBCP project.