JDBC Recipes

A Problem-Solution Approach

Mahmoud Parsian

Apress®

JDBC Recipes: A Problem-Solution Approach

ISBN: 1-59059-520-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Development Editor: Jim Sumser
Technical Reviewer: Sumit Pal
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beckie Stones
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreaders: Liz Welch and Lori Bring
Indexer: Tim Tate
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Making Database Connections Using DataSource

Relativity teaches us the connection between the different descriptions of one and the same reality.

—Albert Einstein

In this chapter, you will learn how to make database connections using JDBC's `DataSource` object. The purpose of this chapter is to provide snippets, reusable code samples, and methods that deal with the `Connection` objects using the `javax.sql.DataSource` interface. When writing this chapter, I relied on JDK 1.4 and the final release of the JDBC 3.0 specification.

This chapter's focus will be on answering the following question: how do you obtain a `java.sql.Connection` object from `javax.sql.DataSource` objects? I can present this question in another way: what are the connection options? This chapter will answer these questions, and for each case I will provide working code.

To select data from tables/views, update columns in tables, create a new table in a database, or do anything useful with any database, you need a database connection (in JDBC, this is called `java.sql.Connection`). Typically, database access in any environment starts with the connection.

4-1. How Do You Create Connection Using a DataSource Object?

`DataSource` (defined in the `javax.sql` package) is an abstraction layer for Java database applications. Database applications may use a JNDI context to find `DataSource` attributes that are configured on the deployment server. You can obtain a DataSource object in two ways:

- *Using JNDI*
- *Without using JNDI*

JNDI is an API for accessing different kinds of naming and directory services. JNDI is not specific to a particular naming or directory service; it can be used to access many different kinds of systems including file systems, Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Enterprise JavaBeans (EJB), as well as directory services such as Lightweight Directory Access Protocol (LDAP) and Network Information Service (NIS). Although you can use JDBC to access a set of relational databases, you can use JNDI to access a set of naming and directory services.

You will look at both of these options in this chapter. From a portability point of view, obtaining a `DataSource` interface using JNDI is more portable than not using JNDI, according to `http://java.sun.com/products/jdbc/articles/package2.html`:

> *The DataSource interface provides an alternative to the DriverManager class for making a connection to a data source. Using a DataSource implementation is better for two important reasons: it makes code more portable, and it makes code easier to maintain. A DataSource object represents a real-world data source. Depending on how it is implemented, the data source can be anything from a relational database to a spreadsheet or a file in tabular format. When a DataSource object has been registered with a JNDI naming service, an application can retrieve it from the naming service and use it to make a connection to the data source it represents.*

The following snippet shows how to retrieve the `DataSource` object associated with the logical name `jdbc/InventoryDB` and then use it to get a connection. The first two lines use the JNDI API to get the `DataSource` object; the third line uses JDBC API to get the connection.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/InventoryDB");
Connection con = ds.getConnection("myUserName", "myPassword");
```

The JDK 1.4 documentation defines `javax.sql.DataSource` as follows:

> *A factory for connections to the physical data source that this DataSource object represents. An alternative to the DriverManager facility, a DataSource object is the preferred means of getting a connection. An object that implements the DataSource interface will typically be registered with a naming service based on the JNDI API.*

The `DataSource` interface is implemented by a driver vendor. Three types of implementations exist:

- *Basic implementation*: Produces a standard `Connection` object.
- *Connection pooling implementation*: Produces a `Connection` object that will automatically participate in connection pooling. This implementation works with a middle-tier connection pooling manager.
- *Distributed transaction implementation*: Produces a `Connection` object that may be used for distributed transactions and almost always participates in connection pooling. This implementation works with a middle-tier transaction manager and almost always with a connection pooling manager.

A `DataSource` object has properties that you can modify when necessary. For example, if the data source moves to a different server, you can change the property for the server. The benefit is that because the data source's properties can be changed, any code accessing that data source does not need to be changed.

A driver that is accessed via a `DataSource` object does not register itself with the `DriverManager` facility. Rather, a `DataSource` object is retrieved through a lookup operation and then used to create a `Connection` object. With a basic implementation, the connection obtained through a `DataSource` object is identical to a connection obtained through the `DriverManager` facility.

This is according to Struts (`http://struts.apache.org/faqs/database.html`):

> *As a rule, you should always use a connection pool to access a database. The DataSource interface is the preferred way to implement a connection pool today. Many containers and database systems now bundle a DataSource implementation that you can use. Most often, the DataSource is made available through JNDI. The JNDI approach makes it easy for your business classes to access the DataSource without worrying about who set it up.*

When connecting to a data source (a relational database such as MySQL or Oracle) using a `DataSource` object registered with a JNDI naming service rather than using the `DriverManager` facility, you get three benefits:

- It makes code more portable.
- It makes code much easier to maintain.
- You get the benefit of connection pooling.

For details on these benefits, refer to *JDBC API Tutorial and Reference, Third Edition* (Addison-Wesley, 2003) by Maydene Fisher, Jon Ellis, and Jonathan Bruce.

JDBC 2.0 introduced a `DataSource` interface that eliminates connection URLs and driver names in your Java applications. `DataSource` enables you to register (using JNDI API) instances of `DataSource` with a unique name; then, other applications can retrieve the registered `DataSource` using the unique name. A `DataSource` object provides a new method for JDBC clients to obtain a DBMS connection (represented as `java.sql.Connection`). A `DataSource` object is usually created, deployed (that is, registered), located (lookup operation), and managed separately from the Java applications that use it.

Figure 4-1 shows the life cycle of a `DataSource` object.

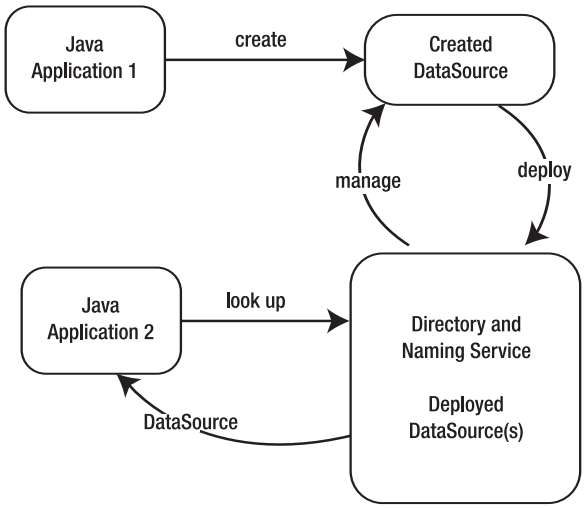

Figure 4-1. *Life cycle of a* `DataSource` *object*

In this figure, note that the Directory and Naming Service item is JNDI-enabled and can bind/register and hold any number of `DataSource` objects. Most databases can use CORBA-based naming and directory services, but in order not to tie yourself into a specific implementation of JNDI, in this chapter you will see how to use a file system–based reference implementation of a JNDI SPI driver from JavaSoft. In real production applications, you should select a commercially available naming and directory service product (such as Sun's Directory Server, Novell's Directory Server, and so on).

One Java application (Java Application 1) can create a data source (using some data source configuration from an XML file, a relational database, or a file-based system). Also, Java Application 1 can register (or bind) and manage the created data source. In registering a data source, you have to associate the data source with a unique key. Once the data source is registered, it is accessible to other applications (such as Java Application 2). Java Application 2 can get registered data sources

using a lookup method by providing the unique key. Some vendors (such as BEA's WebLogic and IBM's WebSphere) provide tools for deploying `DataSource` objects and then provide browsing/lookup operations to get the `DataSource` objects.

4-2. How Do You Create a DataSource Object?

To create a `DataSource` object, you define it with a vendor-specific Java class, which implements the `DataSource` interface. For the sake of this discussion, assume that `DBVendorDataSource` (which is a vendor-specific Java class) implements the `DataSource` interface. Then you can create a `DataSource` object by writing the following code:

```
//
// generic solution
//
DBVendorDataSource vendorDataSource = new DBVendorDataSource();
vendorDataSource.setServerName("saratoga");
vendorDataSource.setDatabaseName("payrollDatabase");
vendorDataSource.setDescription("the data source for payroll");
//
// you can set other attributes by using vendorDataSource.setXXX(...)
//

// now cast it to DataSource
DataSource payrollDS = (DataSource) vendorDataSource;
```

4-3. How Do You Create a DataSource Object Using Oracle?

To create a `DataSource` object using the Oracle database, you should use the (vendor-specific) `OracleDataSource` class (defined in the `oracle.jdbc.pool` package and available from Oracle). You can create a `DataSource` object by writing the following code:

```
import oracle.jdbc.pool.OracleDataSource;
import javax.sql.DataSource;
...
OracleDataSource oracleDataSource = new OracleDataSource();
oracleDataSource.setServerName("saratoga");
oracleDataSource.setDatabaseName("payrollDatabase");
oracleDataSource.setDescription("the data source for payroll");
//
// you can set other attributes by
// invoking oracleDataSource.setXXX(...)
//

// now cast it to DataSource
DataSource payrollDS = (DataSource) oracleDataSource;
```

4-4. How Do You Create a DataSource Object Using MySQL?

To create a `DataSource` object using the MySQL database, you should use the (vendor-specific) `MySQLDataSource` class. You can create a `DataSource` object by writing the following code:

```
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
import javax.sql.DataSource;
...
MysqlDataSource mysqlDataSource = new MysqlDataSource();
mysqlDataSource.setServerName("saratoga");
```

```
mysqlDataSource.setDatabaseName("payrollDatabase");
mysqlDataSource.setDescription("the data source for payroll");
//
// NOTE: you can set other attributes by
// invoking mysqlDataSource.setXXX(   )
//

// now cast it to DataSource
DataSource payrollDS = (DataSource) mysqlDataSource;
```

4-5. How Do You Create a DataSource Object Using a Relational Database (Oracle/MySQL)?

To create a `DataSource` object using an Oracle, MySQL, Sybase, or DB2 database, you introduce a `vendor` parameter. (The `vendor` parameter uniquely identifies a specific database such as Oracle, MySQL, or Sybase.) Depending on the `vendor` parameter, you apply a different implementation class for creating a `DataSource` object. For example, if `vendor` equals `oracle`, then you select the `OracleDataSource` class, and if `vendor` equals `mysql`, then you select the `MysqlDataSource` class; otherwise, you return null. (In addition, you can modify this to support more than two vendors.)

Invoking getDataSource() with a Specific User/Password

This shows how to invoke `getDataSource()` with a specific user/password:

```
/**
 * This method creates a DataSource object with a
 * specific username/password. If the vendor parameter
 * is not specified, then it returns null.
 *
 * @param vendor the vendor parameter: "oracle", "mysql",
 *
 */
 public static javax.sql.DataSource getDataSource
   (String vendor,
    String user,
    String password,
    String databaseName,
    String driverType,
    String networkProtocol,
    int portNumber,
    String serverName) throws SQLException {

   if (vendor.equals("oracle")) {
      // create Oracle's DataSource
      OracleDataSource ods = new OracleDataSource();      ods.setUser(user);
      ods.setPassword(password);
      ods.setDatabaseName(databaseName);
      ods.setDriverType(driverType);
      ods.setNetworkProtocol(networkProtocol);
      ods.setPortNumber(portNumber);
      ods.setServerName(serverName);
      return ods;
   }
   else if (vendor.equals("mysql")) {
      // create MySQL's DataSource
```

```
      MysqlDataSource mds = new MysqlDataSource();
      mds.setUser(user);
      mds.setPassword(password);
      mds.setDatabaseName(databaseName);
      //mds.setDriverType(driverType);
      //mds.setNetworkProtocol(networkProtocol);
      mds.setPortNumber(portNumber);
      mds.setServerName(serverName);
      return mds;
   }
   else {
      return null;
   }
}
```

Viewing the getDataSource() Results with a Specific User/Password

This code shows the result:

```
public static void main(String[] args)
   throws SQLException, javax.naming.NamingException {
   // create an Oracle DataSource with
   // specific username/password
   DataSource ods = getDataSource("oracle",
      "system", "gozal", "scorpian", "thin",
      "tcp", 1521, "localhost");
   Connection oraConn = ods.getConnection();
   System.out.println("oraConn="+oraConn);

   // create a MySQL DataSource with
   // specific username/password
   DataSource mds = getDataSource("mysql",
      "root", "root", "tiger",
      "", "", 3306, "localhost");
   Connection myConn = mds.getConnection();
   System.out.println("myConn="+myConn);
}
```

Invoking getDataSource() Without a Specific User/Password

This shows how to invoke `getDataSource()` without a specific user/password:

```
/**
 * This method creates a DataSource object without a
 * specific username/password. If the vendor parameter
 * is not specified, then it returns null.  When client
 * uses this DataSource, it has to pass username/password.
 *
 * @param vendor the vendor parameter: "oracle", "mysql",
 *
 */
 public static DataSource getDataSource
   (String vendor,
    String databaseName,
```

```
    String driverType,
    String networkProtocol,
    int portNumber,
    String serverName) throws SQLException {

   if (vendor.equals("oracle")) {
      // create Oracle's DataSource
      OracleDataSource ods = new OracleDataSource();
      ods.setDatabaseName(databaseName);
      ods.setDriverType(driverType);
      ods.setNetworkProtocol(networkProtocol);
      ods.setPortNumber(portNumber);
      ods.setServerName(serverName);
      return ods;
   }
   else if (vendor.equals("mysql")) {
      // create MySQL's DataSource
      MysqlDataSource mds = new MysqlDataSource();
      mds.setDatabaseName(databaseName);
      //mds.setDriverType(driverType);
      //mds.setNetworkProtocol(networkProtocol);
      mds.setPortNumber(portNumber);
      mds.setServerName(serverName);
      return mds;
   }
   else {
     return null;
   }
}
```

Viewing the getDataSource() Results Without a Specific User/Password

This code shows the result:

```
public static void main(String[] args)
   throws SQLException, javax.naming.NamingException {
   // create an Oracle DataSource with
   // specific username/password
   DataSource ods = getDataSource("oracle",
     "scorpian", "thin", "tcp", 1521, "localhost");
   String user = "system";
   String password = "gozal";
   Connection oraConn = ods.getConnection(user, password);
   System.out.println("oraConn="+oraConn);

   // create a MySQL DataSource with
   // specific username/password
   DataSource mds = getDataSource("mysql",
     "root", "root", "tiger", "", "", 3306, "localhost");
   String user2 = "root";
   String password2 = "root";
   Connection myConn = mds.getConnection(user2, password2);
   System.out.println("myConn="+myConn);
}
```

4-6. How Do You Create a DataSource Object Using a DataSource Factory?

You can create DataSource objects by using a data source factory (DSF) object.

First, you create a DSF object; second, you use the DSF object to create DataSource objects. To create a DSF object, you have at least two options: you can write your own custom code or use a third-party package. In this section, I will show how to use the package org.apache.torque.dsfactory from Apache. (For details about this package, please see http://db.apache.org/torque-32/.) This package has a DataSourceFactory interface implemented by several classes (JndiDataSourceFactory, PerUserPoolDataSourceFactory, and SharedPoolDataSourceFactory). You may provide your own implementation class as well for DataSourceFactory. DataSourceFactory is defined as follows:

```
package org.apache.torque.dsfactory;
import org.apache.commons.configuration.Configuration;

//A factory that returns a DataSource.
public interface DataSourceFactory {
   // returns the DataSource configured by the factory.
   public javax.sql.DataSource getDataSource()
      throws TorqueException;
   // initialize the factory.
   public void initialize(Configuration configuration)
      throws TorqueException;
}
```

You can use one of the implementation classes to create a DataSourceFactory and then use the created DSF to create a DataSource object:

```
JndiDataSourceFactory jndiDSF = new JndiDataSourceFactory();
// create a DataSource configuration object
Configuration config = getConfiguration(<your-datasource-properties>);
// first you need to initialize
jndiDSF.initialize(config);
...
// get a DataSource object:
DataSource ds = jndiDSF.getDataSource();
...
// now use the DataSource object to create Connection objects:
Connection conn = ds.getConnection();
```

4-7. What Are the DataSource Properties?

A DataSource object has several properties that identify and describe the real-world data source (such as an Oracle database, a MySQL database, and so on) that the object represents. These properties include information such as the driver type, the URL of the database server, the name of the database, the network protocol to use to communicate with the database server, and so on.

DataSource properties follow the JavaBeans design pattern and are usually set when a DataSource object is created and deployed. The JDBC API specifies a standard set of properties and a standard name for each property. Table 4-1 describes the standard name, the data type, and a description for each of the standard properties. According to the JDBC specification, a DataSource implementation does not have to support all of these properties.

Table 4-1. *Standard* `DataSource` *Properties*

Property Name	Type	Description
`databaseName`	`String`	The name of a particular database on a server
`dataSourceName`	`String`	The logical name for the underlying `XADataSource` or `ConnectionPoolDataSource` object; used only when pooling of connections or distributed transactions are implemented
`description`	`String`	A description of this data source
`networkProtocol`	`String`	The network protocol used to communicate with the server
`password`	`String`	The user's database password
`portNumber`	`Int`	The port number where a server is listening for requests
`roleName`	`String`	The initial SQL role name
`serverName`	`String`	The database server name
`user`	`String`	The user's account name

If a `DataSource` object supports a property, it must supply getter (`get<PropertyName>`) and setter (`set<PropertyName>`) methods for it. The following code fragment illustrates the methods that a `DataSource` object, `ds`, would need to include if it supports, for example, the property `description`:

```
DataSource ds = <get-a-DataSource object>;
ds.setDescription("This db server is for payroll processing.");
String description = ds.getDescription();
```

4-8. How Do You Deploy/Register a DataSource?

Assume that you have created a `DataSource` object and you want to deploy/register it as `jdbc/PayrollDataSource`. Using JNDI naming services, you can bind/register a data source with a JNDI naming service.

The naming service provides distributed naming support and can be implemented in many different ways:

- File-based systems
  - `java.naming.factory.initial` (for example, `com.sun.jndi.fscontext.RefFSContextFactory`)
  - `java.naming.provider.url` (for example, `file:c:\\jdbcDataSource`)
- Directory-based systems (such as LDAP)
  - `java.naming.factory.initial` (for example, `com.ibm.websphere.naming.WsnInitialContextFactory`)
  - `java.naming.provider.url` (for example, `java:comp/env/jdbc/SampleDB`)
- CORBA-based systems
- NIS-based systems
- DNS-based systems
- RMI-based systems
  - `java.naming.factory.initial` (for example, `com.sun.jndi.rmi.registry.RegistryContextFactory`)
  - `java.naming.provider.url` (for example, `rmi://localhost:1099`)

4-9. How Do You Use a File-Based System for Registering a DataSource Object?

To use the file system–based JNDI, you need to do the following:

1. Download `fscontext1_*.zip` from `http://java.sun.com/products/jndi`. (The asterisk refers to a version number of the software bundle.)
2. Extract `providerutil.jar` and `fscontext.jar`.
3. Include in your `CLASSPATH` environment variable the `providerutil.jar` and `fscontext.jar` files extracted from the `fscontext1_*.zip` file. (Also, use a full pathname for your JAR files.)

In this way, you can deploy a `DataSource` object by using the following code listings. Specifically, this is the `getContext()` method:

```
private static Context getContext(String classFactory,
                 String providerURL)
   throws javax.naming.NamingException {
   //
   // Set up environment for creating initial context
   //
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, classFactory);
    env.put(Context.PROVIDER_URL, providerURL);
    Context context = new InitialContext(env);
    return context;
}

private static Context getFileSystemContext(String directoryName)
   throws javax.naming.NamingException {
   //
   // Set up environment for creating initial context
   //
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file:" + directoryName);
    Context context = new InitialContext(env);
    return context;
}
```

This is the `deployDataSource()` method:

```
private static void deployDataSource(Context context,
                String jndiName,
                DataSource ds)
   throws javax.naming.NamingException{
   //
   // register the data source under the jndiName using
   // the provided context
   //
    context.bind(jndiName, ds);
}
```

And this code shows how to deploy/register the `DataSource` object:

```
// dataSourceName is a unique name to identify the data source.
String jndiDataSourceName = "jdbc/PayrollDataSource";
```

```
// create an Oracle DataSource with
// specific username/password
DataSource ods = createDataSource("oracle",
   "system", "gozal", "scorpian", "thin",
   "tcp", 1521, "localhost");

// now bind it using JNDI
String directoryName = "c:\\jdbcDataSource";
Context context = getFileSystemContext(directoryName);

// register/bind DataSource
deployDataSource(context, jndiDataSourceName, ods);
```

What happens when you deploy a data source using the file system? Under the directory name (`c:\jdbcDataSource`), it creates a file called `.bindings`, which has the following content (the content of the `.bindings` file has been formatted to fit the page):

```
#This file is used by the JNDI FSContext.
#Sun Feb 02 00:20:35 PST 2003
jdbc/MyDataSource/RefAddr/3/Encoding=String
jdbc/MyDataSource/RefAddr/5/Type=databaseName
jdbc/MyDataSource/RefAddr/6/Type=networkProtocol
jdbc/MyDataSource/FactoryName=oracle.jdbc.pool.OracleDataSourceFactory
jdbc/MyDataSource/RefAddr/1/Encoding=String
jdbc/MyDataSource/RefAddr/6/Encoding=String
jdbc/MyDataSource/RefAddr/7/Type=portNumber
jdbc/MyDataSource/RefAddr/6/Content=tcp
jdbc/MyDataSource/RefAddr/0/Type=url
jdbc/MyDataSource/RefAddr/5/Content=scorpian
jdbc/MyDataSource/RefAddr/4/Encoding=String
jdbc/MyDataSource/RefAddr/3/Content=thin
jdbc/MyDataSource/RefAddr/1/Content=system
jdbc/MyDataSource/ClassName=oracle.jdbc.pool.OracleDataSource
jdbc/MyDataSource/RefAddr/1/Type=userName
jdbc/MyDataSource/RefAddr/2/Encoding=String
jdbc/MyDataSource/RefAddr/7/Encoding=String
jdbc/MyDataSource/RefAddr/2/Type=passWord
jdbc/MyDataSource/RefAddr/3/Type=driverType
jdbc/MyDataSource/RefAddr/0/Encoding=String
jdbc/MyDataSource/RefAddr/5/Encoding=String
jdbc/MyDataSource/RefAddr/7/Content=1521
jdbc/MyDataSource/RefAddr/4/Type=serverName
jdbc/MyDataSource/RefAddr/4/Content=localhost
jdbc/MyDataSource/RefAddr/2/Content=gozal
jdbc/MyDataSource/RefAddr/0/Content=jdbc\:oracle\:thin\:
   @(DESCRIPTION\=(ADDRESS\=(PROTOCOL\=tcp)(PORT\=1521)
   (HOST\=localhost))(CONNECT_DATA\=(SID\=scorpian)))
```

4-10. What Is the Problem with File-Based DataSource Objects?

The problem with file-based `DataSource` objects is the compromise on security. Since the password is not encrypted (in any way), this is a security hole; therefore, it is not a viable choice in production systems. In production systems, you should use production-ready "directory-based" products. (In these cases, the password can be protected by groups/roles privileges.) For example, the password in file-based systems can be viewed as clear text if you have access to the file system; see the following password line—no encryption is used at all:

```
#This file is used by the JNDI FSContext.
#Sun Feb 02 00:20:35 PST 2003
jdbc/MyDataSource/RefAddr/3/Encoding=String
jdbc/MyDataSource/RefAddr/5/Type=databaseName
...
jdbc/MyDataSource/RefAddr/2/Type=passWord
...
jdbc/MyDataSource/RefAddr/2/Content=gozal
```

4-11. How Do You Retrieve a Deployed/Registered DataSource?

Assume the registered name of a data source is `jdbc/PayrollDataSource`. Using JNDI , you can look up (search) a data source with a JNDI naming service. In this way, you can access a `DataSource` object by using code that looks like this:

```
String dataSourceName = "jdbc/PayrollDataSource";
String user = "dbUser":
String password = "dbPassword";
Context context = new InitialContext();
DataSource ds = (DataSource) context.lookup(dataSourceName);
Connection conn = ds.getConnection(user, password);
//
// or
//
// if a DataSource has been created (before registration)
// with user and password attributes, then you can get the
// connection without passing the user/password.
Connection con = ds.getConnection();
```

You can use a Java method to accomplish the task:

```
public static DataSource getDataSource(String dataSourceName)
   throws Exception {
   if (dataSourceName == null) {
      return null;
   }
   Context context = new InitialContext();
   DataSource ds = (DataSource) context.lookup(dataSourceName);
   return ds;
}
```

Then you can use the `getDataSource()` method for getting `Connection` objects:

```
String dataSourceName = "jdbc/PayrollDataSource";
String user = "dbUser":
String password = "dbPassword";
DataSource ds = null;
try {
   ds = getDataSource(dataSourceName);
}
catch(Exception e) {
   // handle the exception
   // DataSource is not available/registered
}

if (ds != null) {
   Connection conn = ds.getConnection(user, password);
   //
```

```
   // or
   //
   // if a DataSource has been created (before registration)
   // with user and password attributes, then you can get the
   // connection without passing the user/password.
   // Connection conn = ds.getConnection();
}
```

4-12. How Do You Obtain a Connection with the DataSource Without Using JNDI?

A `DataSource` object provides a portable way for JDBC clients to obtain a DBMS connection. To create a `DataSource` object, you can use a vendor-specific class (such as `OracleDataSource` from Oracle or `MysqlDataSource` from MySQL), and then you can cast it to a `DataSource` object.

Oracle Example

Using the `OracleDataSource` class, you can create an instance of a `DataSource` object like so:

```
String databaseName = "scorpian";
String driverType = "thin";
String networkProtocol = "tcp";
int portNumber = 1521;
String serverName = "localhost";
...
OracleDataSource ods = new OracleDataSource();
ods.setDatabaseName(databaseName);
   ods.setDriverType(driverType);
ods.setNetworkProtocol(networkProtocol);
ods.setPortNumber(portNumber);
ods.setServerName(serverName);
DataSource oracleDS = (DataSource) ods;
```

Once you have created an instance of a `DataSource` object, you can use the `getConnection()` method:

```
String username = "system";
String password = "gozal";
java.sql.Connection connection = null;
try {
   connection = oracleDS.getConnection(username, password);
}
catch(SQLException e) {
   // database access error occurred
   // handle the exception
   e.printStackTrace();
    ...
}
```

MySQL Example

Using the `MysqlDataSource` class, you can create an instance of a `DataSource` object:

```
String databaseName = "tiger";
String networkProtocol = "tcp";
int portNumber = 3306;
String serverName = "localhost";
```

```
...
MysqlDataSource mds = new MysqlDataSource();
mds.setDatabaseName(databaseName);
mds.setNetworkProtocol(networkProtocol);
mds.setPortNumber(portNumber);
mds.setServerName(serverName);
DataSource mysqlDS = (DataSource) mds;
```

Once you have created an instance of a `DataSource` object, you can use the `getConnection()` method:

```
String username = "root";
String password = "root";java.sql.Connection connection = null;
try {
   connection = mysqlDS.getConnection(username, password);
}
catch(SQLException e) {
   // database access error occurred
   // handle the exception
   e.printStackTrace();
    ...
}
```

4-13. How Do You Obtain a Connection with the DataSource Using JNDI?

The steps for obtaining and using a connection with the `javax.sql` package API differ slightly from those using the `java.sql` package core API. Using the `javax.sql` package, you access a relational database as follows:

1. Retrieve a `javax.sql.DataSource` object from the JNDI naming service.
2. Obtain a `Connection` object from the data source. (If a data source is created with a username and password, then you can get a connection without passing a username and password; otherwise, you must pass the username and password.)
3. Using the `Connection` object (`java.sql.Connection`), send SQL queries or updates to the database management system.
4. Process the results (returned as `ResultSet` objects).

Figure 4-2 shows `DataSource` and JNDI configuration.

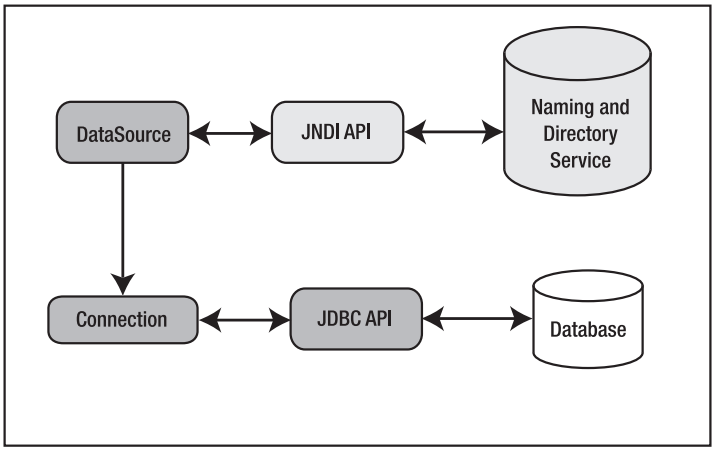

Figure 4-2. `DataSource` *and JNDI configuration*

`DataSource` provides two methods for obtaining a `java.sql.Connection` object:

- `Connection getConnection()`: Attempts to establish a connection with the data source that this `DataSource` object represents
- `Connection getConnection(String username, String password)`: Attempts to establish a connection with the data source that this `DataSource` object represents

The following example shows these steps (assuming that the name of the `DataSource` object is `java:comp/env/jdbc/employeeDB`):

```
import jcb.util.DatabaseUtil;
...
java.sql.Connection conn = null;
java.sql.Statement stmt = null;
java.sql.ResultSet rs = null;
try {
   // Retrieve a DataSource through the JNDI naming service
   java.util.Properties parms = new java.util.Properties();
   parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");

   // Create the Initial Naming Context
   javax.naming.Context context = new javax.naming.InitialContext(parms);

   // Look up through the naming service to retrieve a DataSource object
   javax.sql.DataSource ds = (javax.sql.DataSource)
      context.lookup("java:comp/env/jdbc/employeeDB");

   //Obtain a connection from the DataSource; here you
   // are assuming that the DataSource is created with
   // the required username and password.
   conn = ds.getConnection();

   // query the database
   stmt = conn.createStatement();
   rs = stmt.executeQuery("SELECT id, lastname FROM employees");
```

```
   // process the results
   while (rs.next()) {
      String id = rs.getString("id");
      String lastname = rs.getString("lastname");
      // process and work with results (id, lastname)
   }
}
catch (SQLException e) {
   // handle SQLException
   e.printStackTrace();
}
finally {
   DatabaseUtil.close(rs);
   DatabaseUtil.close(stmt);
   DatabaseUtil.close(conn);
}
```