2.3 Basic configuration

We've looked at an example application and examined Hibernate's core interfaces. To use Hibernate in an application, you need to know how to configure it. Hibernate can be configured to run in almost any Java application and development environment. Generally, Hibernate is used in two- and three-tiered client/server applications, with Hibernate deployed only on the server. The client application is usually a web browser, but Swing and SWT client applications aren't uncommon. Although we concentrate on multitiered web applications in this book, our explanations apply equally to other architectures, such as command-line applications. It's important to understand the difference in configuring Hibernate for managed and non-managed environments:

- *Managed environment*—Pools resources such as database connections and allows transaction boundaries and security to be specified declaratively (that

is, in metadata). A J2EE application server such as JBoss, BEA WebLogic, or IBM WebSphere implements the standard (J2EE-specific) managed environment for Java.

- *Non-managed environment*—Provides basic concurrency management via thread pooling. A servlet container like Jetty or Tomcat provides a non-managed server environment for Java web applications. A stand-alone desktop or command-line application is also considered non-managed. Non-managed environments don't provide automatic transaction or resource management or security infrastructure. The application itself manages database connections and demarcates transaction boundaries.

Hibernate attempts to abstract the environment in which it's deployed. In the case of a non-managed environment, Hibernate handles transactions and JDBC connections (or delegates to application code that handles these concerns). In managed environments, Hibernate integrates with container-managed transactions and datasources. Hibernate can be configured for deployment in both environments.

In both managed and non-managed environments, the first thing you must do is start Hibernate. In practice, doing so is very easy: You have to create a `SessionFactory` from a `Configuration`.

2.3.1 Creating a SessionFactory

In order to create a `SessionFactory`, you first create a single instance of `Configuration` during application initialization and use it to set the location of the mapping files. Once configured, the `Configuration` instance is used to create the `SessionFactory`. After the `SessionFactory` is created, you can discard the `Configuration` class.

The following code starts Hibernate:

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties( System.getProperties() );
SessionFactory sessions = cfg.buildSessionFactory();
```

The location of the mapping file, `Message.hbm.xml`, is relative to the root of the application classpath. For example, if the classpath is the current directory, the `Message.hbm.xml` file must be in the `hello` directory. XML mapping files *must* be placed in the classpath. In this example, we also use the system properties of the virtual machine to set all other configuration options (which might have been set before by application code or as startup options).

METHOD CHAINING

Method chaining is a programming style supported by many Hibernate interfaces. This style is more popular in Smalltalk than in Java and is considered by some people to be less readable and more difficult to debug than the more accepted Java style. However, it's very convenient in most cases.

Most Java developers declare setter or adder methods to be of type `void`, meaning they return no value. In Smalltalk, which has no `void` type, setter or adder methods usually return the receiving object. This would allow us to rewrite the previous code example as follows:

```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

Notice that we didn't need to declare a local variable for the `Configuration`. We use this style in some code examples; but if you don't like it, you don't need to use it yourself. If you *do* use this coding style, it's better to write each method invocation on a different line. Otherwise, it might be difficult to step through the code in your debugger.

By convention, Hibernate XML mapping files are named with the `.hbm.xml` extension. Another convention is to have one mapping file per class, rather than have all your mappings listed in one file (which is possible but considered bad style). Our "Hello World" example had only one persistent class, but let's assume we have multiple persistent classes, with an XML mapping file for each. Where should we put these mapping files?

The Hibernate documentation recommends that the mapping file for each persistent class be placed in the same directory as that class. For instance, the mapping file for the `Message` class would be placed in the `hello` directory in a file named `Message.hbm.xml`. If we had another persistent class, it would be defined in its own mapping file. We suggest that you follow this practice. The monolithic metadata files encouraged by some frameworks, such as the `struts-config.xml` found in Struts, are a major contributor to "metadata hell." You load multiple mapping files by calling `addResource()` as often as you have to. Alternatively, if you follow the convention just described, you can use the method `addClass()`, passing a persistent class as the parameter:

```
SessionFactory sessions = new Configuration()
    .addClass(org.hibernate.auction.model.Item.class)
    .addClass(org.hibernate.auction.model.Category.class)
    .addClass(org.hibernate.auction.model.Bid.class)
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

The `addClass()` method assumes that the name of the mapping file ends with the `.hbm.xml` extension and is deployed along with the mapped class file.

We've demonstrated the creation of a single `SessionFactory`, which is all that most applications need. If another `SessionFactory` is needed—if there are multiple databases, for example—you repeat the process. Each `SessionFactory` is then available for one database and ready to produce `Session`s to work with that particular database and a set of class mappings.

Of course, there is more to configuring Hibernate than just pointing to mapping documents. You also need to specify how database connections are to be obtained, along with various other settings that affect the behavior of Hibernate at runtime. The multitude of configuration properties may appear overwhelming (a complete list appears in the Hibernate documentation), but don't worry; most define reasonable default values, and only a handful are commonly required.

To specify configuration options, you may use any of the following techniques:

- Pass an instance of `java.util.Properties` to `Configuration.setProperties()`.
- Set system properties using `java -Dproperty=value`.
- Place a file called `hibernate.properties` in the classpath.
- Include `<property>` elements in `hibernate.cfg.xml` in the classpath.

The first and second options are rarely used except for quick testing and prototypes, but most applications need a fixed configuration file. Both the `hibernate.properties` and the `hibernate.cfg.xml` files provide the same function: to configure Hibernate. Which file you choose to use depends on your syntax preference. It's even possible to mix both options and have different settings for development and deployment, as you'll see later in this chapter.

A rarely used alternative option is to allow the application to provide a JDBC `Connection` when it opens a Hibernate `Session` from the `SessionFactory` (for example, by calling `sessions.openSession(myConnection)`). Using this option means that you don't have to specify any database connection properties. We don't recommend this approach for new applications that can be configured to use the environment's database connection infrastructure (for example, a JDBC connection pool or an application server datasource).

Of all the configuration options, database connection settings are the most important. They differ in managed and non-managed environments, so we deal with the two cases separately. Let's start with non-managed.

2.3.2 Configuration in non-managed environments

In a non-managed environment, such as a servlet container, the application is responsible for obtaining JDBC connections. Hibernate is part of the application, so it's responsible for getting these connections. You tell Hibernate how to get (or create new) JDBC connections. Generally, it isn't advisable to create a connection each time you want to interact with the database. Instead, Java applications should use a pool of JDBC connections. There are three reasons for using a pool:

- Acquiring a new connection is expensive.
- Maintaining many idle connections is expensive.
- Creating prepared statements is also expensive for some drivers.

Figure 2.2 shows the role of a JDBC connection pool in a web application runtime environment. Since this non-managed environment doesn't implement connection pooling, the application must implement its own pooling algorithm or rely upon a third-party library such as the open source *C3P0* connection pool. Without Hibernate, the application code usually calls the connection pool to obtain JDBC connections and execute SQL statements.

With Hibernate, the picture changes: It acts as a client of the JDBC connection pool, as shown in figure 2.3. The application code uses the Hibernate `Session` and `Query` APIs for persistence operations and only has to manage database transactions, ideally using the Hibernate `Transaction` API.

Using a connection pool

Hibernate defines a plugin architecture that allows integration with any connection pool. However, support for C3P0 is built in, so we'll use that. Hibernate will set up the configuration pool for you with the given properties. An example of a `hibernate.properties` file using C3P0 is shown in listing 2.4.

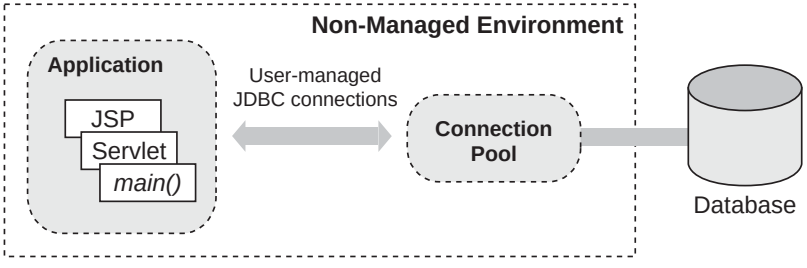

Figure 2.2 JDBC `connection` pooling in a non-managed environment

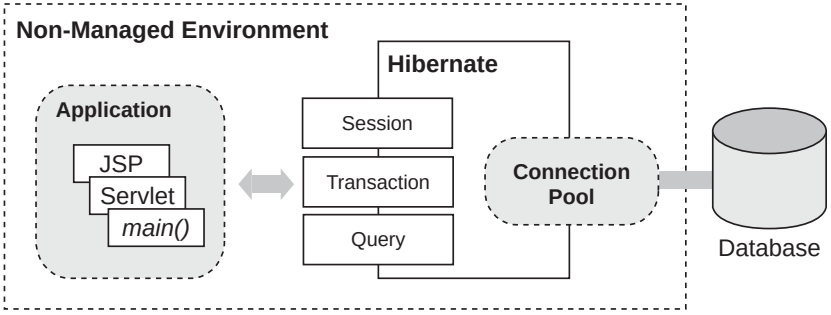

Figure 2.3 Hibernate with a connection pool in a non-managed environment

Listing 2.4 Using `hibernate.properties` for C3P0 connection pool settings

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
```

This code's lines specify the following information, beginning with the first line:

- The name of the Java class implementing the JDBC `Driver` (the driver JAR file must be placed in the application's classpath).
- A JDBC URL that specifies the host and database name for JDBC connections.
- The database user name.
- The database password for the specified user.
- A `Dialect` for the database. Despite the ANSI standardization effort, SQL is implemented differently by various databases vendors. So, you must specify a `Dialect`. Hibernate includes built-in support for all popular SQL databases, and new dialects may be defined easily.
- The minimum number of JDBC connections that C3P0 will keep ready.

- The maximum number of connections in the pool. An exception will be thrown at runtime if this number is exhausted.
- The timeout period (in this case, 5 minutes or 300 seconds) after which an idle connection will be removed from the pool.
- The maximum number of prepared statements that will be cached. Caching of prepared statements is essential for best performance with Hibernate.
- The idle time in seconds before a connection is automatically validated.

Specifying properties of the form `hibernate.c3p0.*` selects C3P0 as Hibernate's connection pool (you don't need any other switch to enable C3P0 support). C3P0 has even more features than we've shown in the previous example, so we refer you to the Hibernate API documentation. The Javadoc for the class `net.sf.hibernate.cfg.Environment` documents every Hibernate configuration property, including all C3P0-related settings and settings for other third-party connection pools directly supported by Hibernate.

The other supported connection pools are Apache DBCP and Proxool. You should try each pool in your own environment before deciding between them. The Hibernate community tends to prefer C3P0 and Proxool.

Hibernate also ships with a default connection pooling mechanism. This connection pool is only suitable for testing and experimenting with Hibernate: You should *not* use this built-in pool in production systems. It isn't designed to scale to an environment with many concurrent requests, and it lacks the fault tolerance features found in specialized connection pools.

Starting Hibernate

How do you start Hibernate with these properties? You declared the properties in a file named `hibernate.properties`, so you need only place this file in the application classpath. It will be automatically detected and read when Hibernate is first initialized when you create a `Configuration` object.

Let's summarize the configuration steps you've learned so far (this is a good time to download and install Hibernate, if you'd like to continue in a non-managed environment):

1. Download and unpack the JDBC driver for your database, which is usually available from the database vendor web site. Place the JAR files in the application classpath; do the same with `hibernate2.jar`.
2. Add Hibernate's dependencies to the classpath; they're distributed along with Hibernate in the `lib/` directory. See also the text file `lib/README.txt` for a list of required and optional libraries.

3. Choose a JDBC connection pool supported by Hibernate and configure it with a properties file. Don't forget to specify the SQL dialect.
4. Let the `Configuration` know about these properties by placing them in a `hibernate.properties` file in the classpath.
5. Create an instance of `Configuration` in your application and load the XML mapping files using either `addResource()` or `addClass()`. Build a `SessionFactory` from the `Configuration` by calling `buildSessionFactory()`.

Unfortunately, you don't have any mapping files yet. If you like, you can run the "Hello World" example or skip the rest of this chapter and start learning about persistent classes and mappings in chapter 3. Or, if you want to know more about using Hibernate in a managed environment, read on.