THE EXPERT'S VOICE® IN JAVA

SECOND EDITION

Hibernate Recipes

A Problem-Solution Approach

Joseph Ottinger, Srinivas Guruzu, and Gary Mak

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Starting with Hibernate

An *object model* uses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, polymorphism, and persistence. The object model enables you to create well-structured and complex systems. In an object model system, *objects* are the components of the system. Objects are instances of classes, and classes are related to other classes via inheritance relationships. An object has an identity, a state, and a behavior. An object model helps you create reusable application frameworks and systems that can evolve over time. In addition, object-oriented systems are usually smaller than non-object-oriented implementations.

A *relational model* defines the structure of data, data manipulation, and data integrity. Data is organized in the form of tables, and different tables are associated by means of referential integrity (a foreign key). Integrity constraints such as a primary key, unique check constraints, and not null are used to maintain an entity's integrity in the relational model.

A relational data model isn't focused on supporting entity-type inheritance: entity-based polymorphic association from an object model can't be translated into similar entities in a relational model. In an object model, you use the state of the model to define equality between objects. But in a relational model, you use an entity's primary key to define equality of entities. Object references are used to associate different objects in an object model, whereas a foreign key is used to establish associations in a relational model. Object references in the object model facilitate easier navigation through the object graph.

Because these two models are distinctly different, you need a way to persist object entities (Java objects) into a relational database. Figure 1-1 provides a simple representation of the object model and Figure 1-2 shows the relational model.

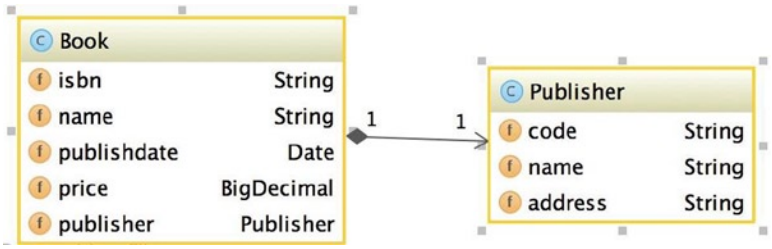

Figure 1-1. *The object model*

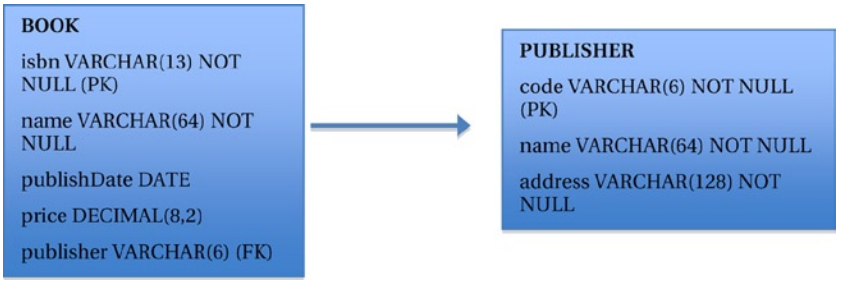

Figure 1-2. *The relational model*

Object/relational mapping (ORM) frameworks help you take advantage of the features present in an object model (such as Java) and a relational model (such as database management systems [DBMSs]). With the help of ORM frameworks, you can persist objects in Java to relational tables using metadata that describes the mapping between the objects and the database. The metadata shields the complexity of dealing directly with SQL and helps you develop solutions in terms of business objects.

An ORM solution can be implemented at various levels:

- *Pure relational*: An application is designed around the relational model.
- *Light object mapping*: Entities are represented as classes and are mapped manually to relational tables.
- *Medium object mapping*: An application is designed using an object model, and SQL is generated during build time using code-generation utilities.
- *Full object mapping*: Supports sophisticated object modeling including composition, inheritance, polymorphism, and persistence by reachability.

The following are the benefits of using an ORM framework:

- *Productivity*: Because you use metadata to persist and query data, development time decreases and productivity increases.
- *Prototyping*: Using an ORM framework is extremely useful for quick prototyping.
- *Maintainability*: Because much of the work is done through configuration, your code has fewer lines and requires less maintenance.
- *Vendor independence*: An ORM abstracts an application from the underlying SQL database and SQL dialect, which gives you the portability to support multiple databases. Java Specification Request 317 (JSR317) defines the Java Persistence API (JPA) specification. Using JPA means you can transparently switch between ORM frameworks such as Hibernate and TopLink.

ORM frameworks also have some disadvantages:

- *Learning curve*: You may experience a steep learning curve as you learn when and how to map and manage objects. You also have to learn a new query language.
- *Overhead*: For simple applications that use a single database and data without many business requirements for complex querying, an ORM framework can be extra overhead.
- *Slower performance*: For large batch updates, performance is slower.

Hibernate is one of the most widely used ORM frameworks in the industry. It provides all the benefits of an ORM solution and implements the JPA defined in the JPA 2.0 specification.

Its main components are as follows:

- *Hibernate Core*: The Core generates SQL and relieves you from manually handling Java Database Connectivity (JDBC) result sets and object conversions. Metadata is defined in simple XML files. The Core offers various options for writing queries: plain SQL; Hibernate Query Language (HQL), which is specific to Hibernate; programmatic criteria, and Query by Example (QBE). It can optimize object loading with various fetching and caching options.
- *Hibernate Annotations*: Hibernate provides the option of defining metadata using annotations. This feature reduces configuration using XML files and makes it easy to define the required metadata directly in the Java source code.
- *Hibernate EntityManager*: The JPA specification defines programming interfaces, life-cycle rules for persistent objects, and query features. The Hibernate implementation for this part of the JPA is available as Hibernate EntityManager.

Hibernate also has Hibernate Search, Hibernate Validator, and Hibernate OGM[1] (No SQL), which are not addressed in this book. This book provides solutions using Hibernate ORM Core and Annotations in a problem-solution approach. The Hibernate version used is 4.3.5.Final.

1-1. Setting Up a Project

Problem

How do you set up a project structure for Hibernate code so that all required libraries are available in the classpath? And how can you build the project and execute unit tests?

Solution

Following are the available build tools that help you build the project and manage the libraries:

- Maven
- Gradle
- SBT

■ **Note** Although we provide all details to create a project structure, manage libraries, build, and execute tests using Maven, the sample code, downloadable from `Apress.com` includes Gradle and SBT.

Maven is a software project-management and comprehension tool. Based on the concept of a Project Object Model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information. In Maven, the `POM.XML` file is the central place in which all the information is stored.

[1]OGM stands for "Object Graph Model." Hibernate OGM provides an object abstraction for NoSQL databases.

How It Works

The version used here is Java 7, which you can download from http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Once you download Java, set the PATH variable to the bin folder in the Java Runtime Environment (JRE). Set the JAVA_HOME variable to the folder in which you installed Java.

■ **Note** You should be able to execute all code in Java 8 as well.

Installing Eclipse

Eclipse is an integrated development environment (IDE) for developing Java applications. The latest version is Kepler (4.3.2). You can download the latest version from https://www.eclipse.org/downloads/.

The Kepler version specifically for 64bin Win can be downloaded from https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/kepler/SR2/eclipse-standard-kepler-SR2-win32-x86_64.zip.

Once you extract and start Eclipse, make sure that it is using Java 7. You can see what version of Java is being used by going to Windows ➤ Preferences ➤ Java ➤ JRE in Eclipse. If you have multiple JRE versions, make sure that jre7 is checked.

Installing Maven

Maven can be run both from a command prompt and from Eclipse:

- To run from a command prompt, download the Maven libraries from http://maven.apache.org/download.cgi. Once you unzip into a folder, add the bin folder to the PATH variable. You can then run and build from the command prompt using commands such as `mvn package`, `mvn install`, and so on.
- To run Maven from Eclipse, use the M2 plug-in in Eclipse to integrate Eclipse and Maven. This plug-in gives you tremendous flexibility to work with Maven POM files while developing in Eclipse. You can install the M2 plug-in from Eclipse: go to Help ➤ Install New Software and enter http://download.eclipse.org/technology/m2e/releases. Follow the instructions. Once this plug-in is installed, you can create a new Maven project from File ➤ New Project.

Setting Up a Maven Project

This book follows the parent-module project structure instead of a flat project structure. All the common dependencies, library versions, and configurations such as properties and so on can be declared in the parent. Any common code between modules can be placed in a separate module and can be used as a library in the rest of the modules, which enables a cleaner code structure.

The parent project contains only the parent POM and represents the complete book. Each chapter is an individual module under the parent. The dependencies that are common across all chapters are specified in the parent POM, and those that are specific to each chapter are mentioned in the corresponding chapter's POM.

■ **Note** For more information on how to create multiple module projects in Maven by using Maven commands, see http://maven.apache.org/plugins/maven-eclipse-plugin/reactor.html.

You can also use Eclipse to create a similar structure. Here are the steps you can use:

1. Create the parent project: go to File ➤ New ➤ Other and select a Maven project. You should see the New Maven Project window shown in Figure 1-3.

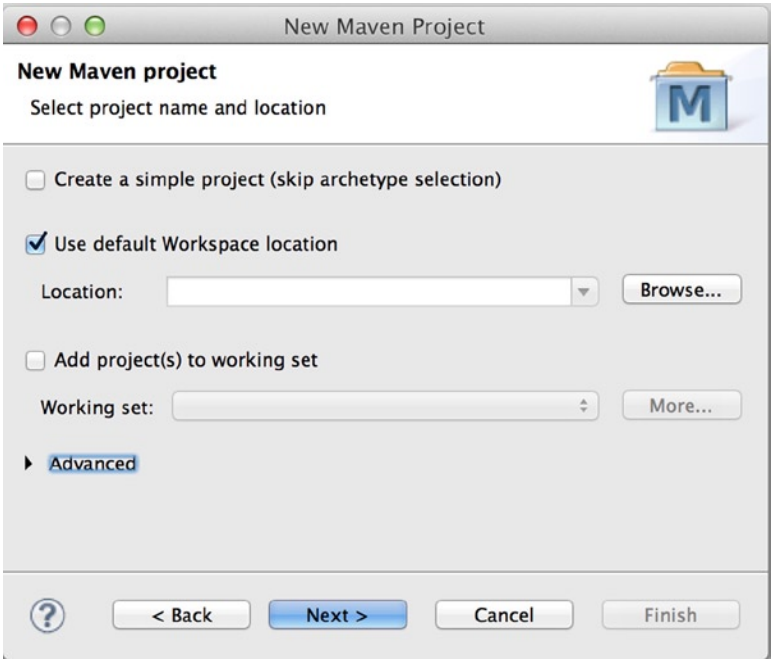

Figure 1-3. *New Maven Project window*

2. Select 'Create A Simple Project'. If you are using the default workspace location, click Next.
3. In the configuration window that displays, specify the Group ID and Artifact ID. The version has a default value. For packaging, choose `'pom'`. This book uses the following values (see Figure 1-4):
   - Group ID: `com.apress.hibernaterecipes`
   - Artifact ID: `ver2`
   - Version: `1.0.0`
   - Packaging: `pom`

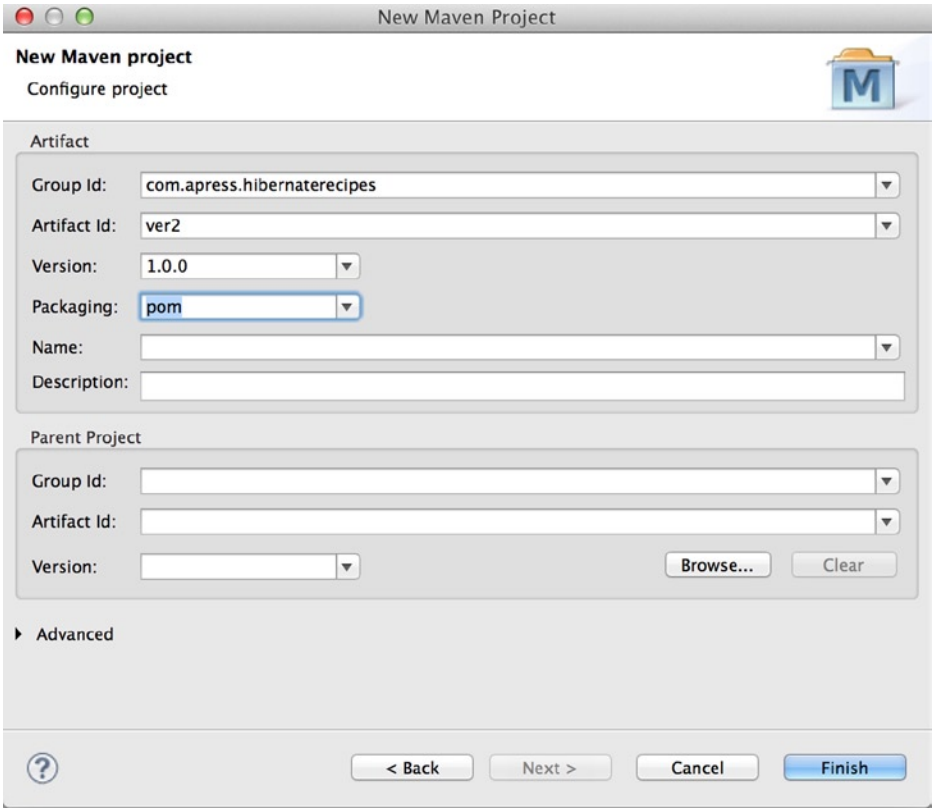

Figure 1-4. *Configuring Maven project values*

4. Click Finish.

A parent project called 'ver2' is created because `ver2` was specified as the Artifact ID. When you expand the 'ver2' project in Eclipse, you see only the `src` folder and a POM file.

Creating a Maven Project Using an Archetype

You can use archetypes to create Maven projects. There are many archetypes available for various kinds of projects such as Java-Spring, Java-Struts, and so on.

Here is how to create a simple Java project using the quickstart archetype:

1. Make sure that the Maven path is set and you can run the `mvn` command from the command line.
2. Execute the following command:

```
mvn archetype:generate -DgroupId="groupId of your project"
-DartifactId="artifact id of your application" -Dverion="version of your project"
-DarchetypeArtifactId=maven-archetype-quickstart
```

This will create a simple Java project with folders for Java source code and tests. This will not create a folder for resources. You can manually create this folder if it is needed.

Running Unit Tests

All unit tests are written in TestNG, and Eclipse has a plug-in for TestNG that can be installed. Go to Help ➤ Install New Software and enter `http://beust.com/eclipse`. Follow the instructions. Once this plug-in is installed, you should be able to create and execute unit testing with the TestNG application programming interface (API).

You can also execute unit tests for each chapter by accessing the chapter's home folder and running the Maven command `mvn test`.

Creating Modules

Now it's time to create modules for a cleaner code structure and to run individual chapters.

1. Right-click the parent project in Eclipse (ver2, in this case)," and choose New ➤ Other ➤ Maven Module (see Figure 1-5).

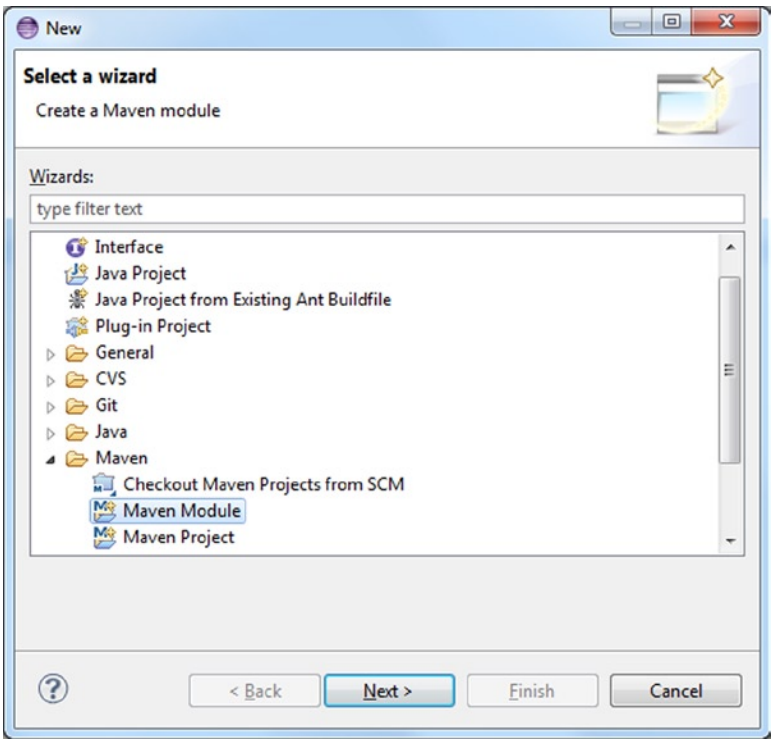

Figure 1-5. *Creating a Maven module*

2. Click Next.
3. In the next window, choose 'Create Simple Project' and a name for your module (we chose `chapter1`). Your parent project should be selected in the Parent Project field.
4. Click 'Next;' the Group ID, Artifact ID, Version, and Packaging fields should already be populated. In this case, the Group ID is `com.apress.hibernaterecipes`, which is the same as the Group ID of the parent project. The Artifact ID is the module name. Version is `0.0.1-SNAPSHOT` by default, and Packaging should be `jar`.

Now you have a parent project as well your first module. If you look into the POM of your parent project, you see that a `<module>` tag is added with the same name that you specified for your module.

The POM in your module has the `<parent>` tag with the `<groupId>`, `<artifactId>`, and `<version>` tags of the parent project:

```
<modelVersion>4.0.0</modelVersion>
 <parent>
   <groupId>com.apress.hibernaterecipes</groupId>
   <artifactId>ver2</artifactId>
   <version>1.0.0</version>
 </parent>
 <artifactId>chapter1</artifactId>
</project>
```

Now that the parent project and module are set up, add the Hibernate JAR files to the project. Because Hibernate JARs are used by all modules, these dependencies are added to the parent POM.

Add the following to the parent POM:

```
<dependencies>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-core</artifactId>
         <version>4.3.5.Final</version>
      </dependency>
</dependencies>
```

Once you save and refresh, 'Maven Dependencies' is added to your module folder structure. The Hibernate JARs and their dependencies are populated under Maven Dependencies.

If you see the screen shown in Figure 1-6, the dependency is added in the parent POM, and the dependent JARs are added under Maven Dependencies in the module.

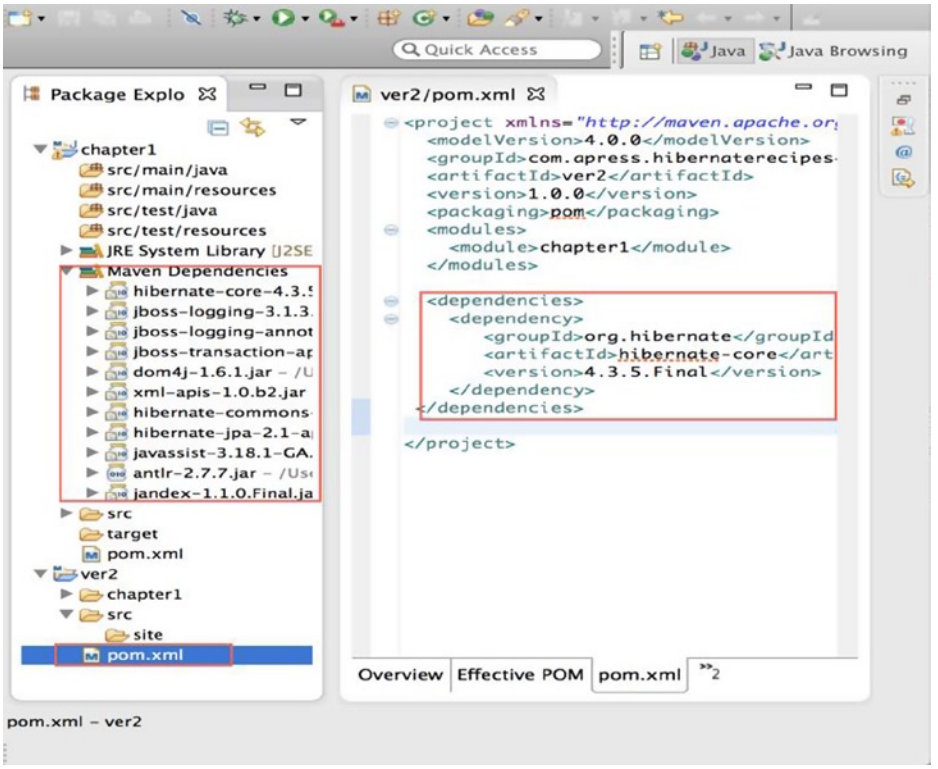

Figure 1-6. *Maven Dependencies*

■ **Note** Also included in the code repository is the configuration for a project set up using Gradle and SBT (you can download it from the code repository).

1-2. Setting Up a Database

Problem

How do you set up a database to work with Hibernate?

Solution

The H2 database in embedded mode is used for all testing in this book. In addition to instructions on how to set up H2, we provide documentation on how to set up Derby in server mode.

How It Works: Setting Up H2

The following dependency should be added to `pom.xml`:

```
<dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>1.4.178</version>
</dependency>
```

This dependency provides the required JARs to connect to an H2 database.

Configuring Hibernate

Add the required database properties either to `persistence.xml` or `hibernate.cgf.xml` depending on what you are using. Because we test with `hibernate.cfg.xml` and `persistence.xml`, these properties are added to both the files:

```
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:file:./chapter1</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
```

- The first property sets the driver class to be used to connect to the database. The `connection.url` property sets the URL to connect to the database. Because it is an embedded mode, we use the file URL.
- `jdbc:h2:file:./chapter1` creates the database file in the `chapter1` folder. For each chapter the URL to create the database file is the appropriate chapter home folder. The URL for `chapter2` is `jdbc:h2:file:./chapter2`, for example.
- The next two properties are for username and password. By default, H2 uses 'sa' as the username with no password.
- After this configuration is done, you should be able to run the tests that are bundled in the code. Each chapter has its own set of tests in the test folder. If you already installed the TestNG plug-in, go to the test folder, go to a specific test (for example, `Recipe1JPATest.java` in `chapter1`), right-click the test file, and click Run As ➤ TestNG Test. When you run the tests in the test folder, two database files are created in the chapter's home folder. For example, if you execute tests for `chapter1`, the `chapter1.mv.db` and `chapter1.trace.db` files are created in the `chapter1` folder.
- The `.mv.db` file is the actual database file that contains all scripts that were executed. Although it is not readable, if you want to view the schema and data, you can view it via the H2 console.

Setting Up the H2 Console to View Data

1. You can download the H2 console from http://www.h2database.com/html/main.html. There are installers for different environments.
2. After you install the console, you see 'H2 Console, which' is a web-based user interface (UI) that opens the connections page in the web browser. By default, the database selected is `'Generic H2(Embedded),'` and the driver selected is `org.h2.driver`.

3. The JDBC URL should be specified to the `.mv` file in the chapter folder (for example, if your Hibernate recipes code is in `C:\HR`). You find the `chapter1.mv` database file in`C:\HR\chapter1`, so the JDBC URL should be `jdbc:h2:file:C:\HR\chapter1\chapter1`.
4. Leave the default values for username and password.
5. Click Connect. You should see the schema on the left pane. When you click a table name, it adds the `SELECT` SQL statement in the code window. If you execute the statement, you should be able to view the data.

How It Works: Setting Up Derby

The following JAR files are required for the Derby setup:

- `Derby.jar`.
- `Derbyclient.jar`.
- `Derbynet.jar`.
- `Derbytools`.

Installing Derby

Derby is an open-source SQL relational database engine written in Java. You can go to `http://db.apache.org/derby/derby_downloads.html` and download the latest version. Derby also provides plug-ins for Eclipse, which provide the required JAR files for development and a command prompt (`ij`) in Eclipse to execute Data Definition Language (DDL) and Data Manipulation Language (DML) statements.

Creating a Derby Database Instance

To create a new Derby database called `BookShopDB` at the `ij` prompt, use the following command:

```
connect 'jdbc:derby://localhost:1527/BookShopDB;create=true;
user=book;password=book';
```

After the database is created, execute the SQL scripts in the next section to create the tables.

Creating the Tables (Relational Model)

These solutions use the example of a bookshop. Books are published by a publisher, and the contents of a book are defined by the chapters. The entities `Book` and `Publisher` are stored in the database; you can perform various operations such as reading, updating, and deleting.

Because an ORM is a mapping between an object model and a relational model, you first create the relational model by executing the DDL statements to create the tables/entities in the database. You later see the object model in Java, and finally you see the mapping between the relational and the object models.

Create the tables for the online bookshop using the following SQL statements (Figure 1-7 gives details of the object model and Figure 1-8 shows the relational model):

```
CREATE TABLE publisher (
  code VARCHAR(6) PRIMARY KEY,
  name VARCHAR(64) NOT NULL,
  address VARCHAR(128) NOT NULL,
  UNIQUE (name)
);

CREATE TABLE book (
  isbn VARCHAR(13) PRIMARY KEY,
  name VARCHAR(64) NOT NULL,
  publishDate DATE,
  price DECIMAL(8, 2),
  publisher VARCHAR(6),
  FOREIGN KEY (publisher) REFERENCES publisher (code),
  UNIQUE (name)
);
```

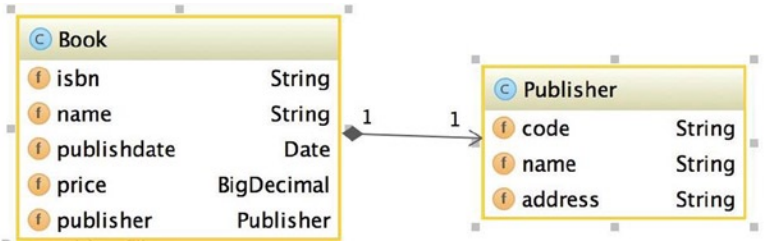

Figure 1-7. *The object model for the bookshop*

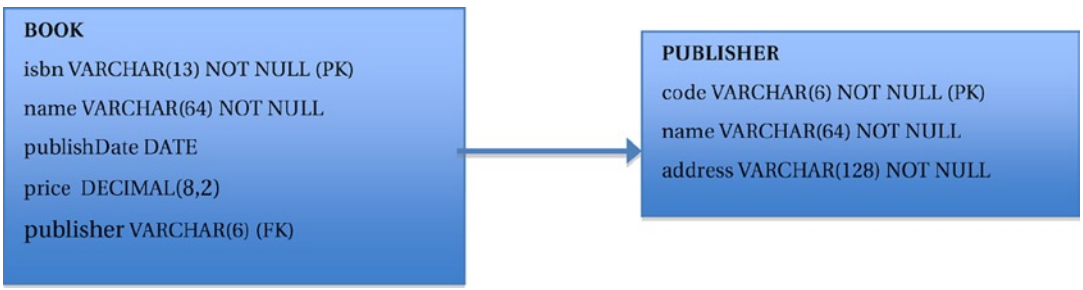

Figure 1-8. *The relational model for the bookshop*

Next, input some data for these tables using the following SQL statements:

```
insert into PUBLISHER(code, name, address)
values ('001', 'Apress', 'New York ,New York');
insert into PUBLISHER(code, name, address)
values ('002', 'Manning', 'San Francisco, CA');
```

```
insert into book(isbn, name, publisher, publishDate, price)
values ('PBN123', 'Spring Recipes', '001', DATE('2008-02-02'), 30);
insert into book(isbn, name, publisher, publishDate, price)
values ('PBN456', 'Hibernate Recipes', '002', DATE('2008-11-02'), 40);
```

1-3. Configuring Hibernate

Problem

How do you configure a Java project that uses an object/relational framework such as Hibernate as a persistence framework? How do you configure Hibernate programmatically?

Solution

Import the required JAR files into your project's classpath and create mapping files that map the state of a Java entity to the columns of its corresponding table. From the Java application, execute Create, Read, Update, Delete (CRUD) operations on the object entities. Hibernate takes care of translating the object state from the object model to the relational model.

How It Works

To configure a Java project to use the Hibernate framework, start by downloading the required JARs and configuring them in the build path.

If you add the following dependency to the Maven `pom.xml` file, all dependent libraries are downloaded under Maven Dependencies:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>4.3.5.Final</version>
</dependency>
```

Although we use both annotations and XML configurations for the Hibernate setup wherever possible, the preference is to use annotations instead of XML.

Configuration

Before Hibernate can retrieve and persist objects for you, you have to tell it your application's settings. For example, which kind of objects are persistent objects? What kind of database are you using? How do you connect to the database? You can configure Hibernate in three ways:

- *Programmatic configuration*: Use the API to load the `hbm` file, load the database driver, and specify the database connection details.
- *XML configuration*: Specify the database connection details in an XML file that's loaded along with the `hbm` file. The default file name is `hibernate.cfg.xml`. You can use another name by specifying the name explicitly.
- *Properties file configuration*: Similar to the XML configuration, but uses a `.properties` file. The default name is `hibernate.properties`.

This solution introduces only the first two approaches (programmatic and XML configuration). The properties file configuration is much like XML configuration.

Programmatic Configuration

The following code loads the configuration programmatically. If you have a very specific use case to configure programmatically, you can use this method; otherwise, the preferred way is to use annotations.

The `Configuration` class provides the API to load the `hbm` files, to specify the driver to be used for the database connection, and to provide other connection details:

```
Configuration configuration = new Configuration()
.addResource("com/metaarchit/bookshop/Book.hbm.xml")
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyTenSevenDialect")
.setProperty("hibernate.connection.driver_class", "org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url", "jdbc:derby://localhost:1527/BookShopDB")
.setProperty("hibernate.connection.username", "book")
.setProperty("hibernate.connection.password", "book");

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Instead of using `addResource()` to add the mapping files, you can also use `addClass()` to add a persistent class and let Hibernate load the mapping definition for this class:

```
Configuration configuration = new Configuration()
.addClass(com.metaarchit.bookshop.Book.class)
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyDialect")
.setProperty("hibernate.connection.driver_class", "org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url", "jdbc:derby://localhost:1527/BookShopDB")
.setProperty("hibernate.connection.username", "book")
.setProperty("hibernate.connection.password", "book");

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

If your application has hundreds of mapping definitions, you can pack it in a JAR file and add it to the Hibernate configuration. This JAR file must be found in your application's classpath:

```
Configuration configuration = new Configuration()
.addJar(new File("mapping.jar"))
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyDialect")
.setProperty("hibernate.connection.driver_class", "org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url", "jdbc:derby://localhost:1527/BookShopDB")
.setProperty("hibernate.connection.username", "book")
.setProperty("hibernate.connection.password", "book");
ServiceRegistry  serviceRegistry = new StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

SESSIONFACTORY

The following statement creates a Hibernate `SessionFactory` to use in the preceding code:

```
SessionFactory factory = configuration.buildSessionFactory(serviceRegistry);
```

A *session factory* is a global object for maintaining `org.hibernate.Session` objects. It i's instantiated once and i's thread-safe. You can look up the `SessionFactory` from a Java Naming and Directory Interface (JNDI) context in an `ApplicationServer` or any other location.

XML Configuration

Another way to configure Hibernate is to use an XML file. You create the file `hibernate.cfg.xml` in the source directory, so Eclipse copies it to the root of your classpath:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>

    <session-factory>
       <!-- H2 Configuration -->

        <property name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:file:./chapter1</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">create</property>
        <mapping resource="com/apress/hibernaterecipes/chapter1/model/Book.hbm.xml"/>
        <mapping resource="com/apress/hibernaterecipes/chapter1/model/Publisher.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Now the code fragment to build up a session factory can be simplified. The configuration loads the `hibernate.cfg.xml` file from the root of the classpath:

```
Configuration configuration = new Configuration().configure();
```

This method loads the default `hibernate.cfg.xml` file from the root classpath. The new `Configuration()` loads the `hibernate.properties` file, and the `configure()` method loads the `hibernate.cfg.xml` file if `hibernate.properties` isn't found. If you need to load another configuration file located elsewhere (not in the root classpath), you can use the following code:

```
new Configuration().configure("/config/recipes.cfg.xml")
```

This code looks for `recipes.cfg.xml` in the `config` subdirectory of the classpath.

Opening and Closing Sessions

A Hibernate `Session` object represents a unit of work and is bound to the current thread. It also represents a transaction in a database. A session begins when `getCurrentSession()` is first called on the current thread. The `Session` object is then bound to the current thread. When the transaction ends with a commit or rollback, Hibernate unbinds the session from the thread and closes it.

Just as with JDBC, you need to do some initial cleanup for Hibernate. First, ask the session factory to open a new session for you. After you finish your work, you must remember to close the session[2]:

```
Session session = factory.openSession();
try {
       // Using the session to retrieve objects
}catch(Exception e)
{
       e.printStackTrace();
} finally {
       session.close();
}
```

Creating Mapping Definitions

First ask Hibernate to retrieve and persist the book objects for you. For simplicity, 'ignore the publisher right now. Create a `Book.hbm.xml` XML file in the same package as the `Book` class. This file is called the *mapping definition* for the `Book` class. The `Book` objects are called *persistent objects* or *entities* because they can be persisted in a database and represent the real-world entities:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>

    <class name="com.apress.hibernaterecipes.chapter1.model.Book" table="BOOK" lazy="false">
        <id name="isbn">
            <column name="ISBN" sql-type="varchar(13)" not-null="true"/>
        </id>
        <property name="name">
            <column name="NAME" sql-type="varchar(64)" not-null="true" unique="true"/>
        </property>
        <property name="publishdate">
            <column name="PUBLISHDATE" sql-type="date"/>
        </property>
        <property name="price">
            <column name="PRICE" sql-type="decimal" precision="8" scale="2"/>
        </property>
        <many-to-one name="publisher" column="PUBLISHERCODE" cascade="all"/>
    </class>
</hibernate-mapping>
```

[2]The sample code rarely includes the exception handling for Hibernate, because we want to focus on how Hibernate is used, rather than on Java's exception mechanism.

Each persistent object must have an identifier, which' is used by Hibernate to uniquely identify that object. Here, you use the ISBN as the identifier for a Book object.

Retrieving and Persisting Objects

Given the ID of a book (an ISBN in this case), you can retrieve the unique Book object from the database. There are multiple ways to do it - session.load() and session.get() are merely two common ones you'll see:

```
Book book = (Book) session.load(Book.class, isbn);
```

and

```
Book book = (Book) session.get(Book.class, isbn);
```

What's the difference between a load() method and a get() method? First, when the given ID can't be found, the load() method throws an org.hibernate.ObjectNotFoundException exception, whereas the get() method returns a null object. Second, the load() method just returns a proxy by default; the database isn't hit until the proxy is first invoked. The get() method hits the database immediately. The load() method is useful when you need only a proxy, not' a database call. You need only a proxy in a given session when you have to associate an entity before persisting.

Just as you can use SQL to query a database, you can use Hibernate to query objects using HQL. For example, note the following code queries for all the Book objects:

```
Query query = session.createQuery("from Book");
List books = query.list();
```

If you're sure that only one object matches, you can use the uniqueResult() method to retrieve the unique result object:

```
Query query = session.createQuery("from Book where isbn = ?");
query.setString(0, isbn);
Book book = (Book) query.uniqueResult();
```

The create() method inserts a new row into the BOOK table. It also loads the object from the database to validate that it exists and is the correct object.

```
@Test
    public void testCreate() {
        Session session = SessionManager.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();
        Publisher publisher = new Publisher();
        publisher.setCode("apress");
        publisher.setName("Apress");
        publisher.setAddress("233 Spring Street, New York, NY 10013");
        session.persist(publisher);
        tx.commit();
        session.close();

        session = SessionManager.getSessionFactory().openSession();
        tx = session.beginTransaction();
        Publisher publisher1 = (Publisher) session.load(Publisher.class, "apress");
        assertEquals(publisher.getName(), publisher1.getName());
        tx.commit();
        session.close();
    }
```

The Hibernate output is as follows:

```
Hibernate: insert into PUBLISHER (NAME, ADDRESS, CODE) values (?, ?, ?)
Hibernate: select publisher0_.CODE as CODE1_1_0_, publisher0_.NAME as NAME2_1_0_,
publisher0_.ADDRESS as ADDRESS3_1_0_ from PUBLISHER publisher0_ where publisher0_.CODE=?
```

1-4. Using the JPA EntityManager

Problem

Is there a generalized mechanism to configure ORM with less dependency on individual providers such as Hibernate, TopLink, and so on?

Solution

A *persistence context* is defined by the JPA specification as a set of managed entity instances in which the entity instances and their life cycles are managed by an entity manager. Each ORM vendor provides its own entity manager, which is a wrapper around the core API and thus supports the JPA programming interfaces, JPA entity instance life cycles, and query language, providing a generalized mechanism for object/relational development and configuration.

How It Works

You obtain the Hibernate EntityManager from an entity manager factory. When container-managed entity managers are used, the application doesn't interact directly with the entity manager factory. Such entity managers are obtained mostly through JNDI lookup. In the case of application-managed entity managers, the application must use the entity manager factory to manage the entity manager and the persistence context life cycle. This example uses the application-managed entity manager.

Add the following dependency to the `pom.xml` file:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.3.5.Final</version>
</dependency>
```

`EntityManagerFactory` has the same role as `SessionFactory` in Hibernate. It acts a factory class that provides the `EntityManager` class to the application. It can be configured either programmatically or by using XML. When you use XML to configure it, the file must be named `persistence.xml` and must be located in the classpath.

The `persistence.xml` files should provide a unique name for each persistence unit; this name is the way applications reference the configuration while obtaining a `javax.persistence.EntityManagerFactory` reference.

Here's the persistence.xml file for the Book and Publisher examples:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
             xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="chapter1" transaction-type="RESOURCE_LOCAL">
        <mapping-file>com/apress/hibernaterecipes/chapter1/model/Publisher.hbm.xml</mapping-file>
        <mapping-file>com/apress/hibernaterecipes/chapter1/model/Book.hbm.xml</mapping-file>

        <class>com.apress.hibernaterecipes.chapter1.model.Publisher</class>
        <class>com.apress.hibernaterecipes.chapter1.model.Book</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:file:~/chapter1jpa"/>

            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

The RESOURCE_LOCAL transaction type is used here. Two transaction types define transactional behavior: JTA and RESOURCE_LOCAL. JTA is used in J2EE-managed applications in which the container is responsible for transaction propagation. For application-managed transactions, you can use RESOURCE_LOCAL.

The <provider> tag specifies the third-party ORM implementation you use. In this case, it i's configured to use the Hibernate persistence provider.

The entity instances are configured with the <class> tag.

The rest of the properties are similar to the configuration in the hibernate.cfg.xml file, including the driver class of the database you're connecting to, the connection URL, a username, a password, and the dialect.

Here's the code to create the EntityManagerFactory (EMF) from the configuration and to obtain the EntityManager from the EMF:

```
package com.hibernaterecipes.annotations.dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class SessionManager {

           public static EntityManager getEntityManager() {
              EntityManagerFactory managerFactory =
Persistence.createEntityManagerFactory("chapter1");
              EntityManager manager = managerFactory.createEntityManager();

              return manager;
           }
}
```

The `Persistence.createEntityManagerFactory()` method creates the EMF. The parameter that it takes is the name of the persistence unit (in this case, `"chapter1"`). It should be the same as the name specified in the `persistence.xml` file's `<persistence-unit>` tag:

```
<persistence-unit name="chapter1"  transaction-type="RESOURCE_LOCAL">
```

The `Book` entity instance remains the same as defined in the XML `config` file:

```
public class Book {
    private String isbn;
    private String name;
    private Date publishdate;
    private BigDecimal price;
    private Publisher publisher;

    //Getters and Setters

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Book book = (Book) o;

        if (isbn != null ? !isbn.equals(book.isbn) : book.isbn != null) return false;
        if (name != null ? !name.equals(book.name) : book.name != null) return false;
        if (price != null ? !price.equals(book.price) : book.price != null) return false;
        if (publishdate != null ? !publishdate.equals(book.publishdate) : book.publishdate
        != null) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = isbn != null ? isbn.hashCode() : 0;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        result = 31 * result + (publishdate != null ? publishdate.hashCode() : 0);
        result = 31 * result + (price != null ? price.hashCode() : 0);
        return result;
    }
}
```

Here's the `Publisher` class:

```
public class Publisher {
    private String code;
    private String name;
    private String address;
```

```
    //Getters and setters

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Publisher publisher = (Publisher) o;

        if (address != null ? !address.equals(publisher.address) : publisher.address != null)
        return false;
        if (code != null ? !code.equals(publisher.code) : publisher.code != null)
        return false;
        if (name != null ? !name.equals(publisher.name) : publisher.name != null)
        return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = code != null ? code.hashCode() : 0;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        result = 31 * result + (address != null ? address.hashCode() : 0);
        return result;
    }
}
```

Here's the test code to create and retrieve the object graph of a `Publisher` and a `Book`:

```
@Test
    public void testCreateObjectGraph() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Publisher publisher = new Publisher();
        publisher.setCode("apress");
        publisher.setName("Apress");
        publisher.setAddress("233 Spring Street, New York, NY 10013");

        Book book = new Book();
        book.setIsbn("9781484201282");
        book.setName("Hibernate Recipes");
        book.setPrice(new BigDecimal("44.00"));
        book.setPublishdate(Date.valueOf("2014-10-10"));
        book.setPublisher(publisher);

        em.persist(book);

        em.getTransaction().commit();
        em.close();
```

```
        em = emf.createEntityManager();
        em.getTransaction().begin();
        Book book1 = em.find(Book.class, "9781484201282");
        assertEquals(book.getName(), book1.getName());
        assertNotNull(book.getPublisher());
        assertEquals(book.getPublisher().getName(), publisher.getName());
        em.getTransaction().commit();
        em.close();

        em = emf.createEntityManager();
        em.getTransaction().begin();

        // this changes the publisher back to managed state by
        // returning the managed version of publisher
        publisher = em.merge(publisher);

        book = new Book();
        book.setIsbn("9781430265177");
        book.setName("Beginning Hibernate");
        book.setPrice(new BigDecimal("44.00"));
        book.setPublishdate(Date.valueOf("2014-04-04"));
        book.setPublisher(publisher);

        em.persist(book);
        em.getTransaction().commit();
        em.close();

        em = emf.createEntityManager();
        em.getTransaction().begin();
        book1 = em.find(Book.class, "9781430265177");
        assertEquals(book.getName(), book1.getName());
        assertNotNull(book.getPublisher());
        assertEquals(book.getPublisher().getName(), publisher.getName());
        em.getTransaction().commit();
        em.close();
    }
```

Create the publisher object and, when creating the book object with the title *'Hibernate Recipes*, set the publisher on the book to the created publisher object. `em.persist(book)` persists the complete object graph of the publisher and the book. To retrieve the book details from the database, use `em.find(Book.class, isbn)`.

We persist another book with the title '*Beginning Hibernate*'. Because the same publisher object is used for this book as the first book, now only the new book is inserted. The SQL statements in the Hibernate ouput are shown here.

For book1, *Hibernate Recipes*:

```
Hibernate: insert into PUBLISHER (NAME, ADDRESS, CODE) values (?, ?, ?)

Hibernate: insert into BOOK (NAME, PUBLISHDATE, PRICE, PUBLISHERCODE, ISBN)
values (?, ?, ?, ?, ?)

Hibernate: select book0_.ISBN as ISBN1_0_0_, book0_.NAME as NAME2_0_0_, book0_.
PUBLISHDATE as PUBLISHD3_0_0_, book0_.PRICE as PRICE4_0_0_, book0_.PUBLISHERCODE as
PUBLISHE5_0_0_, publisher1_.CODE as CODE1_1_1_, publisher1_.NAME as NAME2_1_1_, publisher1_.
ADDRESS as ADDRESS3_1_1_ from BOOK book0_ left outer join PUBLISHER publisher1_ on book0_.
PUBLISHERCODE=publisher1_.CODE where book0_.ISBN=?

Hibernate: select publisher0_.CODE as CODE1_1_0_, publisher0_.NAME as NAME2_1_0_,
publisher0_.ADDRESS as ADDRESS3_1_0_ from PUBLISHER publisher0_ where publisher0_.CODE=?
```

You see that there is an insert into the publisher as well.
For book2, *Beginning Hibernate*:

```
Hibernate: insert into BOOK (NAME, PUBLISHDATE, PRICE, PUBLISHERCODE, ISBN)
values (?, ?, ?, ?, ?)

Hibernate: select book0_.ISBN as ISBN1_0_0_, book0_.NAME as NAME2_0_0_, book0_.PUBLISHDATE
as PUBLISHD3_0_0_, book0_.PRICE as PRICE4_0_0_, book0_.PUBLISHERCODE as PUBLISHE5_0_0_,
publisher1_.CODE as CODE1_1_1_, publisher1_.NAME as NAME2_1_1_, publisher1_.ADDRESS
as ADDRESS3_1_1_ from BOOK book0_ left outer join PUBLISHER publisher1_ on book0_.
PUBLISHERCODE=publisher1_.CODE where book0_.ISBN=?
```

1-5. Enabling Logging in Hibernate

Problem

How do you determine what SQL query is being executed by Hibernate? How can you see the Hibernate' internal workings? How do you enable logging to troubleshoot complex issues related to Hibernate?

Solution

You have to enable Hibernate logging in the Hibernate configuration. Hibernate uses Simple Logging Facade for Java (SLF4J) to log various system events. SLF4J, which is distributed as a free software license, abstracts the actual logging framework that an application uses. SLF4J can direct your logging output to several logging frameworks:

- *NOP*: Null logger implementation
- *Simple*: A logging antiframework that is very simple to use and attempts to solve every logging problem in one package
- *Log4j version 1.2*: A widely used open-source logging framework
- *JDK 1.4 logging*: A logging API provided by Java

- *JCL*: An open-source Commons logging framework that provides an interface with thin wrapper implementations for other logging tools
- *Logback*: A serializable logger that logs after its deserialization, depending on the chosen binding

To set up logging, you need the `slf4j-api.jar` file in your classpath, together with the JAR file for your preferred binding: `slf4j-log4j12.jar` in the case of log4j. You can also enable a property called `showsql` to see the exact query being executed. You can configure a logging layer such as Apache log4j to enable Hibernate class- or package-level logging. And you can use the `Statistics` interface provided by Hibernate to obtain detailed information.

How It Works

You have to configure the Hibernate `show_sql` property to enable logging.

Inspecting the SQL Statements Issued by Hibernate

Hibernate generates SQL statements that enable you to access the database behind the scene. You can set the `show_sql` property to `true` in the `hibernate.cfg.xml` XML configuration file to print the SQL statements to `stdout`:

```
<property name="show_sql">true</property>
```

Enabling Live Statistics

You can enable live statistics by setting the `hibernate.generate_statistics` property in the configuration file:

```
<property name="hibernate.generate_statistics">true</property>
```

You can also access statistics programmatically by using the `Statistics` interfaces. Hibernate provides `SessionStatistics` and `Statistics` interfaces in the `org.hibernate.stat` package. The following code shows the use of some utility methods:

```
SessionFactory sessionFactory = SessionManager.getSessionFactory();
            session = sessionFactory.openSession();
            SessionStatistics sessionStats = session.getStatistics();
            Statistics stats = sessionFactory.getStatistics();
            tx = session.beginTransaction();
            Publisher publisher = new Publisher();
            publisher.setCode("apress");
            publisher.setName("Apress");
            publisher.setAddress("233 Spring Street, New York, NY 10013");
            session.persist(publisher);
            tx.commit();
            logger.info("getEntityCount- "+sessionStats.getEntityCount());
            logger.info("openCount- "+stats.getSessionOpenCount());
            logger.info("getEntityInsertCount- "+stats.getEntityInsertCount());
            stats.logSummary();
            session.close();
```

The output of this code sample is shown here (the complete log is given for clarity):

```
HCANN000001: Hibernate Commons Annotations {4.0.4.Final}
HHH000412: Hibernate Core {4.3.5.Final}
HHH000206: hibernate.properties not found
HHH000021: Bytecode provider name : javassist
HHH000043: Configuring from resource: /hibernate.cfg.xml
HHH000040: Configuration resource: /hibernate.cfg.xml
HHH000221: Reading mappings from resource: com/apress/hibernaterecipes/chapter1/model/Book.
hbm.xml
HHH000221: Reading mappings from resource: com/apress/hibernaterecipes/chapter1/model/
Publisher.hbm.xml
HHH000041: Configured SessionFactory: null
HHH000402: Using Hibernate built-in connection pool (not for production use!)
HHH000401: using driver [org.hsqldb.jdbcDriver] at URL [jdbc:hsqldb:file:./chapter1;write_
delay=false]
HHH000046: Connection properties: {}
HHH000006: Autocommit mode: false
HHH000115: Hibernate connection pool size: 20 (min=1)
HHH000400: Using dialect: org.hibernate.dialect.HSQLDialect
HHH000399: Using default transaction strategy (direct JDBC transactions)
HHH000397: Using ASTQueryTranslatorFactory
HHH000227: Running hbm2ddl schema export
HHH000230: Schema export complete
Session Metrics {
    14000 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    2445000 nanoseconds spent preparing 2 JDBC statements;
    1636000 nanoseconds spent executing 2 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0
collections);
    59000 nanoseconds spent executing 2 partial-flushes (flushing a total of 0 entities
and 0 collections)
}
getEntityCount- 1
openCount- 2
getEntityInsertCount- 1
HHH000161: Logging statistics....
HHH000251: Start time: 1408349026472
HHH000242: Sessions opened: 2
HHH000241: Sessions closed: 1
HHH000266: Transactions: 2
HHH000258: Successful transactions: 2
HHH000187: Optimistic lock failures: 0
HHH000105: Flushes: 1
HHH000048: Connections obtained: 2
HHH000253: Statements prepared: 3
```