Revised Edition of *HIBERNATE IN ACTION*

JAVA PERSISTENCE WITH HIBERNATE

Christian Bauer
Gavin King

foreword by Linda DeMichiel

Sample Chapter

MANNING

2 Starting a project

This chapter covers

- "Hello World" with Hibernate and Java Persistence
- The toolset for forward and reverse engineering
- Hibernate configuration and integration

You want to start using Hibernate and Java Persistence, and you want to learn it with a step-by-step example. You want to see both persistence APIs and how you can benefit from native Hibernate or standardized JPA. This is what you'll find in this chapter: a tour through a straightforward "Hello World" application.

However, a good and complete tutorial is already publicly available in the Hibernate reference documentation, so instead of repeating it here, we show you more detailed instructions about Hibernate integration and configuration along the way. If you want to start with a less elaborate tutorial that you can complete in one hour, our advice is to consider the Hibernate reference documentation. It takes you from a simple stand-alone Java application with Hibernate through the most essential mapping concepts and finally demonstrates a Hibernate web application deployed on Tomcat.

In this chapter, you'll learn how to set up a project infrastructure for a plain Java application that integrates Hibernate, and you'll see many more details about how Hibernate can be configured in such an environment. We also discuss configuration and integration of Hibernate in a managed environment—that is, an environment that provides Java EE services.

As a build tool for the "Hello World" project, we introduce Ant and create build scripts that can not only compile and run the project, but also utilize the Hibernate Tools. Depending on your development process, you'll use the Hibernate toolset to export database schemas automatically or even to reverse-engineer a complete application from an existing (legacy) database schema.

Like every good engineer, before you start your first real Hibernate project you should prepare your tools and decide what your development process is going to look like. And, depending on the process you choose, you may naturally prefer different tools. Let's look at this preparation phase and what your options are, and then start a Hibernate project.

2.1 Starting a Hibernate project

In some projects, the development of an application is driven by developers analyzing the business domain in object-oriented terms. In others, it's heavily influenced by an existing relational data model: either a legacy database or a brand-new schema designed by a professional data modeler. There are many choices to be made, and the following questions need to be answered before you can start:

- Can you start from scratch with a clean design of a new business requirement, or is legacy data and/or legacy application code present?

- Can some of the necessary pieces be automatically generated from an existing artifact (for example, Java source from an existing database schema)? Can the database schema be generated from Java code and Hibernate mapping metadata?
- What kind of tool is available to support this work? What about other tools to support the full development cycle?

We'll discuss these questions in the following sections as we set up a basic Hibernate project. This is your road map:

1. Select a development process
2. Set up the project infrastructure
3. Write application code and mappings
4. Configure and start Hibernate
5. Run the application.

After reading the next sections, you'll be prepared for the correct approach in your own project, and you'll also have the background information for more complex scenarios we'll touch on later in this chapter.

2.1.1 Selecting a development process

Let's first get an overview of the available tools, the artifacts they use as source input, and the output that is produced. Figure 2.1 shows various import and

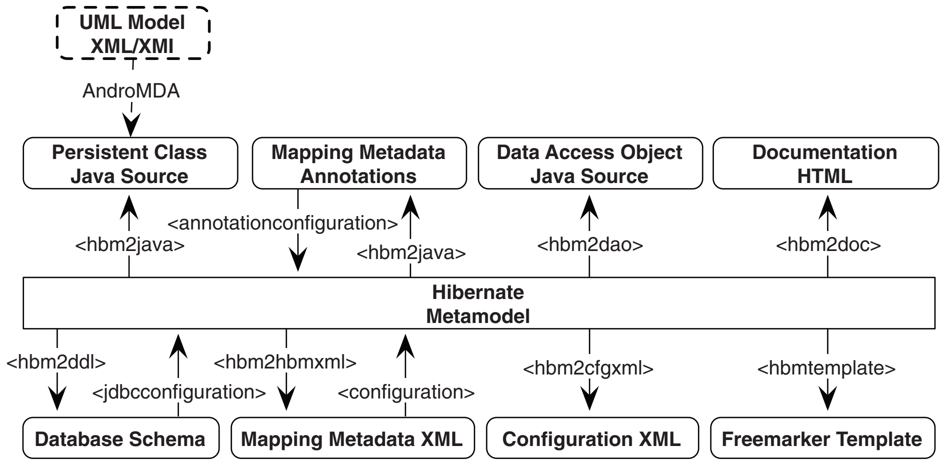


Figure 2.1 Input and output of the tools used for Hibernate development

export tasks for Ant; all the functionality is also available with the Hibernate Tools plug-ins for Eclipse. Refer to this diagram while reading this chapter.[1]

NOTE *Hibernate Tools for Eclipse IDE*—The Hibernate Tools are plug-ins for the *Eclipse IDE* (part of the *JBoss IDE for Eclipse*—a set of wizards, editors, and extra views in Eclipse that help you develop EJB3, Hibernate, JBoss Seam, and other Java applications based on JBoss middleware). The features for forward and reverse engineering are equivalent to the Ant-based tools. The additional *Hibernate Console* view allows you to execute ad hoc Hibernate queries (HQL and `Criteria`) against your database and to browse the result graphically. The Hibernate Tools XML editor supports automatic completion of mapping files, including class, property, and even table and column names. The graphical tools were still in development and available as a beta release during the writing of this book, however, so any screenshots would be obsolete with future releases of the software. The documentation of the Hibernate Tools contains many screenshots and detailed project setup instructions that you can easily adapt to create your first "Hello World" program with the Eclipse IDE.

The following development scenarios are common:

- *Top down*—In *top-down* development, you start with an existing domain model, its implementation in Java, and (ideally) complete freedom with respect to the database schema. You must create mapping metadata—either with XML files or by annotating the Java source—and then optionally let Hibernate's `hbm2ddl` tool generate the database schema. In the absence of an existing database schema, this is the most comfortable development style for most Java developers. You may even use the Hibernate Tools to automatically refresh the database schema on every application restart in development.
- *Bottom up*—Conversely, *bottom-up* development begins with an existing database schema and data model. In this case, the easiest way to proceed is to use the reverse-engineering tools to extract metadata from the database. This metadata can be used to generate XML mapping files, with `hbm2hbmxml` for example. With `hbm2java`, the Hibernate mapping metadata is used to generate Java persistent classes, and even data access objects—in other words, a skeleton for a Java persistence layer. Or, instead of writing to XML

[1] Note that AndroMDA, a tool that generates POJO source code from UML diagram files, isn't strictly considered part of the common Hibernate toolset, so it isn't discussed in this chapter. See the community area on the Hibernate website for more information about the Hibernate module for AndroMDA.

mapping files, annotated Java source code (EJB 3.0 entity classes) can be produced directly by the tools. However, not all class association details and Java-specific metainformation can be automatically generated from an SQL database schema with this strategy, so expect some manual work.

- *Middle out*—The Hibernate XML mapping metadata provides sufficient information to completely deduce the database schema and to generate the Java source code for the persistence layer of the application. Furthermore, the XML mapping document isn't too verbose. Hence, some architects and developers prefer *middle-out* development, where they begin with handwritten Hibernate XML mapping files, and then generate the database schema using `hbm2ddl` and Java classes using `hbm2java`. The Hibernate XML mapping files are constantly updated during development, and other artifacts are generated from this master definition. Additional business logic or database objects are added through subclassing and auxiliary DDL. This development style can be recommended only for the seasoned Hibernate expert.
- *Meet in the middle*—The most difficult scenario is combining existing Java classes and an existing database schema. In this case, there is little that the Hibernate toolset can do to help. It is, of course, not possible to map arbitrary Java domain models to a given schema, so this scenario usually requires at least some refactoring of the Java classes, database schema, or both. The mapping metadata will almost certainly need to be written by hand and in XML files (though it might be possible to use annotations if there is a close match). This can be an incredibly painful scenario, and it is, fortunately, exceedingly rare.

We now explore the tools and their configuration options in more detail and set up a work environment for typical Hibernate application development. You can follow our instructions step by step and create the same environment, or you can take only the bits and pieces you need, such as the Ant build scripts.

The development process we assume first is top down, and we'll walk through a Hibernate project that doesn't involve any legacy data schemas or Java code. After that, you'll migrate the code to JPA and EJB 3.0, and then you'll start a project bottom up by reverse-engineering from an existing database schema.

2.1.2 Setting up the project

We assume that you've downloaded the latest production release of Hibernate from the Hibernate website at http://www.hibernate.org/ and that you unpacked the archive. You also need *Apache Ant* installed on your development machine.

You should also download a current version of HSQLDB from http://hsqldb.org/ and extract the package; you'll use this database management system for your tests. If you have another database management system already installed, you only need to obtain a JDBC driver for it.

Instead of the sophisticated application you'll develop later in the book, you'll get started with a "Hello World" example. That way, you can focus on the development process without getting distracted by Hibernate details. Let's set up the project directory first.

Creating the work directory

Create a new directory on your system, in any location you like; C:\helloworld is a good choice if you work on Microsoft Windows. We'll refer to this directory as WORKDIR in future examples. Create lib and src subdirectories, and copy all required libraries:

```
WORKDIR
    +lib
      antlr.jar
      asm.jar
      asm-attrs.jars
      c3p0.jar
      cglib.jar
      commons-collections.jar
      commons-logging.jar
      dom4j.jar
      hibernate3.jar
      hsqldb.jar
      jta.jar
    +src
```

The libraries you see in the library directory are from the Hibernate distribution, most of them required for a typical Hibernate project. The hsqldb.jar file is from the HSQLDB distribution; replace it with a different driver JAR if you want to use a different database management system. Keep in mind that some of the libraries you're seeing here may not be required for the particular version of Hibernate you're working with, which is likely a newer release than we used when writing this book. To make sure you have the right set of libraries, always check the lib/README.txt file in the Hibernate distribution package. This file contains an up-to-date list of all required and optional third-party libraries for Hibernate—you only need the libraries listed as required for *runtime*.

In the "Hello World" application, you want to store messages in the database and load them from the database. You need to create the domain model for this business case.

Creating the domain model

Hibernate applications define *persistent classes* that are mapped to database tables. You define these classes based on your analysis of the business domain; hence, they're a model of the domain. The "Hello World" example consists of one class and its mapping. Let's see what a simple persistent class looks like, how the mapping is created, and some of the things you can do with instances of the persistent class in Hibernate.

The objective of this example is to store messages in a database and retrieve them for display. Your application has a simple persistent class, `Message`, which represents these printable messages. The `Message` class is shown in listing 2.1.

Listing 2.1 `Message.java`: a simple persistent class

```
package hello;

public class Message {                   Identifier
    private Long id;                     attribute
    private String text;                           Message text
    private Message nextMessage;                  Reference to another
                                                  Message instance
    Message() {}

    public Message(String text) {
        this.text = text;

    }

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

    public Message getNextMessage() {
        return nextMessage;
    }
    public void setNextMessage(Message nextMessage) {
        this.nextMessage = nextMessage;
    }
}
```

The Message class has three attributes: the identifier attribute, the text of the message, and a reference to another Message object. The identifier attribute allows the application to access the database identity—the primary key value—of a persistent object. If two instances of Message have the same identifier value, they represent the same row in the database.

This example uses Long for the type of the identifier attribute, but this isn't a requirement. Hibernate allows virtually anything for the identifier type, as you'll see later.

You may have noticed that all attributes of the Message class have JavaBeans-style property accessor methods. The class also has a constructor with no parameters. The persistent classes we show in the examples will almost always look something like this. The no-argument constructor is a requirement (tools like Hibernate use reflection on this constructor to instantiate objects).

Instances of the Message class can be managed (made persistent) by Hibernate, but they don't *have* to be. Because the Message object doesn't implement any Hibernate-specific classes or interfaces, you can use it just like any other Java class:

```
Message message = new Message("Hello World");
    System.out.println( message.getText() );
```

This code fragment does exactly what you've come to expect from "Hello World" applications: It prints *Hello World* to the console. It may look like we're trying to be cute here; in fact, we're demonstrating an important feature that distinguishes Hibernate from some other persistence solutions. The persistent class can be used in any execution context at all—no special container is needed. Note that this is also one of the benefits of the new JPA entities, which are also plain Java objects.

Save the code for the Message class into your source folder, in a directory and package named hello.

Mapping the class to a database schema

To allow the object/relational mapping magic to occur, Hibernate needs some more information about exactly how the Message class should be made persistent. In other words, Hibernate needs to know how instances of that class are supposed to be stored and loaded. This metadata can be written into an *XML mapping document*, which defines, among other things, how properties of the Message class map to columns of a MESSAGES table. Let's look at the mapping document in listing 2.2.

Listing 2.2 A simple Hibernate XML mapping

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">

        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>

        <property
            name="text"
            column="MESSAGE_TEXT"/>

        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"
            foreign-key="FK_NEXT_MESSAGE"/>

    </class>

</hibernate-mapping>
```

The mapping document tells Hibernate that the `Message` class is to be persisted to the `MESSAGES` table, that the identifier property maps to a column named `MESSAGE_ID`, that the `text` property maps to a column named `MESSAGE_TEXT`, and that the property named `nextMessage` is an association with *many-to-one multiplicity* that maps to a foreign key column named `NEXT_MESSAGE_ID`. Hibernate also generates the database schema for you and adds a foreign key constraint with the name `FK_NEXT_MESSAGE` to the database catalog. (Don't worry about the other details for now.)

The XML document isn't difficult to understand. You can easily write and maintain it by hand. Later, we discuss a way of using annotations directly in the source code to define mapping information; but whichever method you choose, Hibernate has enough information to generate all the SQL statements needed to insert, update, delete, and retrieve instances of the `Message` class. You no longer need to write these SQL statements by hand.

Create a file named Message.hbm.xml with the content shown in listing 2.2, and place it next to your Message.java file in the source package `hello`. The *hbm* suffix is a naming convention accepted by the Hibernate community, and most developers prefer to place mapping files next to the source code of their domain classes.

Let's load and store some objects in the main code of the "Hello World" application.

Storing and loading objects

What you really came here to see is Hibernate, so let's save a new `Message` to the database (see listing 2.3).

Listing 2.3 The "Hello World" main application code

```
package hello;

import java.util.*;

import org.hibernate.*;
import persistence.*;

public class HelloWorld {

    public static void main(String[] args) {

        // First unit of work
        Session session =
            HibernateUtil.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();

        Message message = new Message("Hello World");
        Long msgId = (Long) session.save(message);

        tx.commit();
        session.close();

        // Second unit of work
        Session newSession =
            HibernateUtil.getSessionFactory().openSession();
        Transaction newTransaction = newSession.beginTransaction();

        List messages =
            newSession.createQuery("from Message m order by
          ➥ m.text asc").list();

        System.out.println( messages.size() +
            " message(s) found:" );
        for ( Iterator iter = messages.iterator();
              iter.hasNext(); ) {
            Message loadedMsg = (Message) iter.next();
            System.out.println( loadedMsg.getText() );
        }

        newTransaction.commit();
        newSession.close();

        // Shutting down the application
        HibernateUtil.shutdown();
    }
}
```

Place this code in the file HelloWorld.java in the source folder of your project, in the `hello` package. Let's walk through the code.

The class has a standard Java `main()` method, and you can call it from the command line directly. Inside the main application code, you execute two separate units of work with Hibernate. The first unit stores a new `Message` object, and the second unit loads all objects and prints their text to the console.

You call the Hibernate `Session`, `Transaction`, and `Query` interfaces to access the database:

- `Session`—A Hibernate `Session` is many things in one. It's a single-threaded nonshared object that represents a particular unit of work with the database. It has the persistence manager API you call to load and store objects. (The `Session` internals consist of a queue of SQL statements that need to be synchronized with the database at some point and a map of managed persistence instances that are monitored by the `Session`.)
- `Transaction`—This Hibernate API can be used to set transaction boundaries programmatically, but it's optional (transaction boundaries aren't). Other choices are JDBC transaction demarcation, the JTA interface, or container-managed transactions with EJBs.
- `Query`—A database query can be written in Hibernate's own object-oriented query language (HQL) or plain SQL. This interface allows you to create queries, bind arguments to placeholders in the query, and execute the query in various ways.

Ignore the line of code that calls `HibernateUtil.getSessionFactory()`—we'll get to it soon.

The first unit of work, if run, results in the execution of something similar to the following SQL:

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
    values (1, 'Hello World', null)
```

Hold on—the MESSAGE_ID column is being initialized to a strange value. You didn't set the id property of message anywhere, so you expect it to be NULL, right? Actually, the id property is special. It's an *identifier property*: It holds a generated unique value. The value is assigned to the Message instance by Hibernate when save() is called. (We'll discuss how the value is generated later.)

Look at the second unit of work. The literal string "from Message m order by m.text asc" is a Hibernate query, expressed in HQL. This query is internally translated into the following SQL when list() is called:

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
    from MESSAGES m
    order by m.MESSAGE_TEXT asc
```

If you run this main() method (don't try this now—you still need to configure Hibernate), the output on your console is as follows:

```
1 message(s) found:
    Hello World
```

If you've never used an ORM tool like Hibernate before, you probably expected to see the SQL statements somewhere in the code or mapping metadata, but they aren't there. All SQL is generated at runtime (actually, at startup for all reusable SQL statements).

Your next step would normally be configuring Hibernate. However, if you feel confident, you can add two other Hibernate features—automatic dirty checking and cascading—in a third unit of work by adding the following code to your main application:

```
// Third unit of work
Session thirdSession =
    HibernateUtil.getSessionFactory().openSession();
Transaction thirdTransaction = thirdSession.beginTransaction();

// msgId holds the identifier value of the first message
message = (Message) thirdSession.get( Message.class, msgId );

message.setText( "Greetings Earthling" );
message.setNextMessage(
    new Message( "Take me to your leader (please)" )
);

thirdTransaction.commit();
thirdSession.close();
```

This code calls three SQL statements inside the same database transaction:

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Notice how Hibernate detected the modification to the `text` and `nextMessage` properties of the first message and automatically updated the database—Hibernate did *automatic dirty checking*. This feature saves you the effort of explicitly asking Hibernate to update the database when you modify the state of an object inside a unit of work. Similarly, the new message was made persistent when a reference was created from the first message. This feature is called *cascading save*. It saves you the effort of explicitly making the new object persistent by calling `save()`, as long as it's reachable by an already persistent instance.

Also notice that the ordering of the SQL statements isn't the same as the order in which you set property values. Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called *transactional write-behind*.

If you ran the application now, you'd get the following output (you'd have to copy the second unit of work after the third to execute the query-display step again):

```
2 message(s) found:
Greetings Earthling
Take me to your leader (please)
```

You now have domain classes, an XML mapping file, and the "Hello World" application code that loads and stores objects. Before you can compile and run this code, you need to create Hibernate's configuration (and resolve the mystery of the `HibernateUtil` class).

2.1.3 Hibernate configuration and startup

The regular way of initializing Hibernate is to build a `SessionFactory` object from a `Configuration` object. If you like, you can think of the `Configuration` as an object representation of a configuration file (or a properties file) for Hibernate.

Let's look at some variations before we wrap it up in the `HibernateUtil` class.

Building a SessionFactory

This is an example of a typical Hibernate startup procedure, in one line of code, using automatic configuration file detection:

```
SessionFactory sessionFactory =
    new Configuration().configure().buildSessionFactory();
```

Wait—how did Hibernate know where the configuration file was located and which one to load?

When `new Configuration()` is called, Hibernate searches for a file named `hibernate.properties` in the root of the classpath. If it's found, all `hibernate.*` properties are loaded and added to the `Configuration` object.

When `configure()` is called, Hibernate searches for a file named `hibernate.cfg.xml` in the root of the classpath, and an exception is thrown if it can't be found. You don't have to call this method if you don't have this configuration file, of course. If settings in the XML configuration file are duplicates of properties set earlier, the XML settings override the previous ones.

The location of the `hibernate.properties` configuration file is always the root of the classpath, outside of any package. If you wish to use a different file or to have Hibernate look in a subdirectory of your classpath for the XML configuration file, you must pass a path as an argument of the `configure()` method:

```
SessionFactory sessionFactory = new Configuration()
            .configure("/persistence/auction.cfg.xml")
            .buildSessionFactory();
```

Finally, you can always set additional configuration options or mapping file locations on the `Configuration` object programmatically, before building the `SessionFactory`:

```
SessionFactory sessionFactory = new Configuration()
          .configure("/persistence/auction.cfg.xml")
          .setProperty(Environment.DEFAULT_SCHEMA, "CAVEATEMPTOR")
          .addResource("auction/CreditCard.hbm.xml")
          .buildSessionFactory();
```

Many sources for the configuration are applied here: First the hibernate.properties file in your classpath is read (if present). Next, all settings from /persistence/auction.cfg.xml are added and override any previously applied settings. Finally, an additional configuration property (a default database schema name) is set programmatically, and an additional Hibernate XML mapping metadata file is added to the configuration.

You can, of course, set all options programmatically, or switch between different XML configuration files for different deployment databases. There is effectively no limitation on how you can configure and deploy Hibernate; in the end, you only need to build a `SessionFactory` from a prepared configuration.

NOTE *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 convenient in many cases, such as for the configuration snippets you've seen in this section. Here is how it works: Most Java developers declare setter or adder methods to be of type `void`, meaning they return no value; but in Smalltalk, which has no void type, setter or adder methods usually return the receiving object. We use this Smalltalk style in some code examples, but if you don't like it, you don't need to use it. If you do use this coding style, it's better to write each method invocation on a different line. Otherwise, it may be difficult to step through the code in your debugger.

Now that you know how Hibernate is started and how to build a `SessionFactory`, what to do next? You have to create a configuration file for Hibernate.

Creating an XML configuration file

Let's assume you want to keep things simple, and, like most users, you decide to use a single XML configuration file for Hibernate that contains all the configuration details.

We recommend that you give your new configuration file the default name hibernate.cfg.xml and place it directly in the source directory of your project, outside of any package. That way, it will end up in the root of your classpath after compilation, and Hibernate will find it automatically. Look at the file in listing 2.4.

Listing 2.4 A simple Hibernate XML configuration file

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
        org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
        jdbc:hsqldb:hsql://localhost
    </property>
    <property name="hibernate.connection.username">
        sa
    </property>
```

```
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>

        <!-- Use the C3P0 connection pool provider -->
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.idle_test_period">3000</property>

        <!-- Show and print nice SQL on stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- List of XML mapping files -->
        <mapping resource="hello/Message.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

The *document type declaration* is used by the XML parser to validate this document against the Hibernate configuration DTD. Note that this isn't the same DTD as the one for Hibernate XML mapping files. Also note that we added some line breaks in the property values to make this more readable—you shouldn't do this in your real configuration file (unless your database username contains a line break).

First in the configuration file are the database connection settings. You need to tell Hibernate which database JDBC driver you're using and how to connect to the database with a URL, a username, and a password (the password here is omitted, because HSQLDB by default doesn't require one). You set a `Dialect`, so that Hibernate knows which SQL variation it has to generate to talk to your database; dozens of dialects are packaged with Hibernate—look at the Hibernate API documentation to get a list.

In the XML configuration file, Hibernate properties may be specified without the `hibernate` prefix, so you can write either `hibernate.show_sql` or just `show_sql`. Property names and values are otherwise identical to programmatic configuration properties—that is, to the constants as defined in `org.hibernate.cfg.Environment`. The `hibernate.connection.driver_class` property, for example, has the constant `Environment.DRIVER`.

Before we look at some important configuration options, consider the last line in the configuration that names a Hibernate XML mapping file. The `Configuration` object needs to know about all your XML mapping files before you build the `SessionFactory`. A `SessionFactory` is an object that represents a particular

Hibernate configuration for a particular set of mapping metadata. You can either list all your XML mapping files in the Hibernate XML configuration file, or you can set their names and paths programmatically on the `Configuration` object. In any case, if you list them as a *resource*, the path to the mapping files is the relative location on the classpath, with, in this example, `hello` being a package in the root of the classpath.

You also enabled printing of all SQL executed by Hibernate to the console, and you told Hibernate to format it nicely so that you can check what is going on behind the scenes. We'll come back to logging later in this chapter.

Another, sometimes useful, trick is to make configuration options more dynamic with system properties:

```
...
<property name="show_sql">${displaysql}</property>
...
```

You can now specify a system property, such as with `java -displaysql=true`, on the command line when you start your application, and this will automatically be applied to the Hibernate configuration property.

The database connection pool settings deserve extra attention.

The database connection pool

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 connections. Each application thread that needs to do work on the database requests a connection from the pool and then returns it to the pool when all SQL operations have been executed. The pool maintains the connections and minimizes the cost of opening and closing connections.

There are three reasons for using a pool:

- Acquiring a new connection is expensive. Some database management systems even start a completely new server process for each connection.
- Maintaining many idle connections is expensive for a database management system, and the pool can optimize the usage of idle connections (or disconnect if there are no requests).
- Creating prepared statements is also expensive for some drivers, and the connection pool can cache statements for a connection across requests.

Figure 2.2 shows the role of a connection pool in an unmanaged application runtime environment (that is, one without any application server).

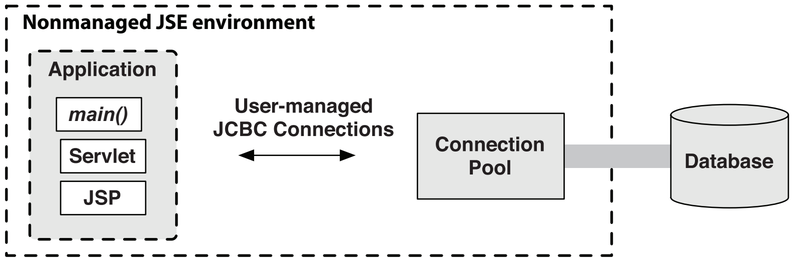


Figure 2.2 JDBC connection pooling in a nonmanaged environment

With no application server to provide a connection pool, an application either implements its own pooling algorithm or relies on a third-party library such as the open source C3P0 connection pooling software. Without Hibernate, the application code calls the connection pool to obtain a JDBC connection and then executes SQL statements with the JDBC programming interface. When the application closes the SQL statements and finally closes the connection, the prepared statements and connection aren't destroyed, but are returned to the pool.

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` API for persistence operations, and it manages database transactions (probably) with the Hibernate `Transaction` API.

Hibernate defines a plug-in architecture that allows integration with any connection-pooling software. However, support for C3P0 is built in, and the software comes bundled with Hibernate, so you'll use that (you already copied the c3p0.jar file into your library directory, right?). Hibernate maintains the pool for you, and configuration properties are passed through. How do you configure C3P0 through Hibernate?

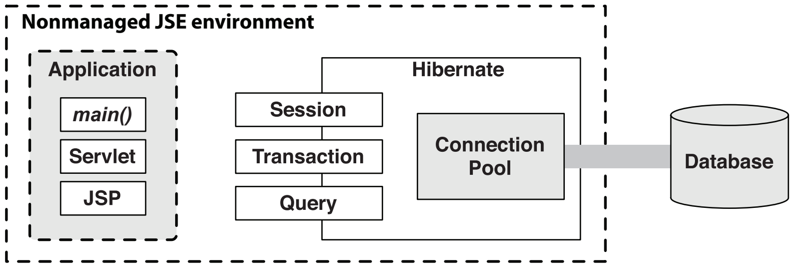


Figure 2.3 Hibernate with a connection pool in a nonmanaged environment

One way to configure the connection pool is to put the settings into your hibernate.cfg.xml configuration file, like you did in the previous section.

Alternatively, you can create a hibernate.properties file in the classpath root of the application. An example of a hibernate.properties file for C3P0 is shown in listing 2.5. Note that this file, with the exception of a list of mapping resources, is equivalent to the configuration shown in listing 2.4.

Listing 2.5 Using hibernate.properties for C3P0 connection pool settings

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
hibernate.dialect = org.hibernate.dialect.HSQLDialect

hibernate.c3p0.min_size = 5                  ❶
hibernate.c3p0.max_size = 20                 ❷
hibernate.c3p0.timeout = 300                 ❸
hibernate.c3p0.max_statements = 50           ❹
hibernate.c3p0.idle_test_period = 3000       ❺

hibernate.show_sql = true
hibernate.format_sql = true
```

❶ This is the minimum number of JDBC connections that C3P0 keeps ready at all times.

❷ This is the maximum number of connections in the pool. An exception is thrown at runtime if this number is exhausted.

❸ You specify the timeout period (in this case, 300 seconds) after which an idle connection is removed from the pool.

❹ A maximum of 50 prepared statements will be cached. Caching of prepared statements is essential for best performance with Hibernate.

❺ This is the idle time in seconds before a connection is automatically validated.

Specifying properties of the form `hibernate.c3p0.*` selects C3P0 as the connection pool (the `c3p0.max_size` option is needed—you don't need any other switch to enable C3P0 support). C3P0 has more features than shown in the previous example; refer to the properties file in the etc/ subdirectory of the Hibernate distribution to get a comprehensive example you can copy from.

The Javadoc for the class `org.hibernate.cfg.Environment` also documents every Hibernate configuration property. Furthermore, you can find an up-to-date table with all Hibernate configuration options in the Hibernate reference documentation. We'll explain the most important settings throughout the book, however. You already know all you need to get started.

FAQ *Can I supply my own connections?* Implement the `org.hibernate.connection.ConnectionProvider` interface, and name your implementation with the `hibernate.connection.provider_class` configuration option. Hibernate will now rely on your custom provider if it needs a database connection.

Now that you've completed the Hibernate configuration file, you can move on and create the `SessionFactory` in your application.

Handling the SessionFactory

In most Hibernate applications, the `SessionFactory` should be instantiated once during application initialization. The single instance should then be used by all code in a particular process, and any `Session` should be created using this single `SessionFactory`. The `SessionFactory` is thread-safe and can be shared; a `Session` is a single-threaded object.

A frequently asked question is where the factory should be stored after creation and how it can be accessed without much hassle. There are more advanced but comfortable options such as JNDI and JMX, but they're usually available only in full Java EE application servers. Instead, we'll introduce a pragmatic and quick solution that solves both the problem of Hibernate startup (the one line of code) and the storing and accessing of the `SessionFactory`: you'll use a static global variable and static initialization.

Both the variable and initialization can be implemented in a single class, which you'll call `HibernateUtil`. This helper class is well known in the Hibernate community—it's a common pattern for Hibernate startup in plain Java applications without Java EE services. A basic implementation is shown in listing 2.6.

Listing 2.6 The `HibernateUtil` class for startup and `SessionFactory` handling

```
package persistence;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

  private static SessionFactory sessionFactory;

  static {
    try {
       sessionFactory=new Configuration()
                           .configure()
```

```
                         .buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        // Alternatively, you could look up in JNDI here
        return sessionFactory;
    }

    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }

}
```

You create a static initializer block to start up Hibernate; this block is executed by the loader of this class exactly once, on initialization when the class is loaded. The first call of `HibernateUtil` in the application loads the class, builds the `SessionFactory`, and sets the static variable at the same time. If a problem occurs, any `Exception` or `Error` is wrapped and thrown out of the static block (that's why you catch `Throwable`). The wrapping in `ExceptionInInitializerError` is mandatory for static initializers.

You've created this new class in a new package called `persistence`. In a fully featured Hibernate application, you often need such a package—for example, to wrap up your custom persistence layer interceptors and data type converters as part of your infrastructure.

Now, whenever you need access to a Hibernate `Session` in your application, you can get it easily with `HibernateUtil.getSessionFactory().openSession()`, just as you did earlier in the `HelloWorld` main application code.

You're almost ready to run and test the application. But because you certainly want to know what is going on behind the scenes, you'll first enable logging.

Enabling logging and statistics

You've already seen the `hibernate.show_sql` configuration property. You'll need it continually when you develop software with Hibernate; it enables logging of all generated SQL to the console. You'll use it for troubleshooting, for performance tuning, and to see what's going on. If you also enable `hibernate.format_sql`, the output is more readable but takes up more screen space. A third option you haven't set so far is `hibernate.use_sql_comments`—it causes Hibernate to put comments inside all generated SQL statements to hint at their origin. For example, you can then easily see if a particular SQL statement was generated from an explicit query or an on-demand collection initialization.

Enabling the SQL output to `stdout` is only your first logging option. Hibernate (and many other ORM implementations) execute SQL statements *asynchronously*. An `INSERT` statement isn't usually executed when the application calls `session.save()`, nor is an `UPDATE` immediately issued when the application calls `item.setPrice()`. Instead, the SQL statements are usually issued at the end of a transaction.

This means that tracing and debugging ORM code is sometimes nontrivial. In theory, it's possible for the application to treat Hibernate as a black box and ignore this behavior. However, when you're troubleshooting a difficult problem, you need to be able to see *exactly* what is going on inside Hibernate. Because Hibernate is open source, you can easily step into the Hibernate code, and occasionally this helps a great deal! Seasoned Hibernate experts debug problems by looking at the Hibernate log and the mapping files only; we encourage you to spend some time with the log output generated by Hibernate and familiarize yourself with the internals.

Hibernate logs all interesting events through *Apache commons-logging*, a thin abstraction layer that directs output to either *Apache Log4j* (if you put log4j.jar in your classpath) or JDK 1.4 logging (if you're running under JDK 1.4 or above and Log4j isn't present). We recommend Log4j because it's more mature, more popular, and under more active development.

To see output from Log4j, you need a file named log4j.properties in your classpath (right next to hibernate.properties or hibernate.cfg.xml). Also, don't forget to copy the log4j.jar library to your lib directory. The Log4j configuration example in listing 2.7 directs all log messages to the console.

Listing 2.7 An example log4j.properties configuration file

```
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
    ➥%5p %c{1}:%L - %m%n

# Root logger option
log4j.rootLogger=INFO, stdout

# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=INFO
```

```
# Log JDBC bind parameter runtime arguments
log4j.logger.org.hibernate.type=INFO
```

The last category in this configuration file is especially interesting: It enables the logging of JDBC bind parameters if you set it to `DEBUG` level, providing information you usually don't see in the ad hoc SQL console log. For a more comprehensive example, check the log4j.properties file bundled in the etc/ directory of the Hibernate distribution, and also look at the Log4j documentation for more information. Note that you should never log anything at `DEBUG` level in production, because doing so can seriously impact the performance of your application.

You can also monitor Hibernate by enabling live statistics. Without an application server (that is, if you don't have a JMX deployment environment), the easiest way to get statistics out of the Hibernate engine at runtime is the `SessionFactory`:

```
Statistics stats =
    HibernateUtil.getSessionFactory().getStatistics();

stats.setStatisticsEnabled(true);
...

stats.getSessionOpenCount();
stats.logSummary();

EntityStatistics itemStats =
    stats.getEntityStatistics("auction.model.Item");
itemStats.getFetchCount();
```

The statistics interfaces are `Statistics` for global information, `Entity-Statistics` for information about a particular entity, `CollectionStatistics` for a particular collection role, `QueryStatistics` for SQL and HQL queries, and `Sec-ondLevelCacheStatistics` for detailed runtime information about a particular region in the optional second-level data cache. A convenient method is `logSum-mary()`, which prints out a complete summary to the console with a single call. If you want to enable the collection of statistics through the configuration, and not programmatically, set the `hibernate.generate_statistics` configuration property to `true`. See the API documentation for more information about the various statistics retrieval methods.

Before you run the "Hello World" application, check that your work directory has all the necessary files:

```
WORKDIR
build.xml
+lib
```

```
    <all required libraries>
  +src
    +hello
      HelloWorld.java
      Message.java
      Message.hbm.xml
    +persistence
      HibernateUtil.java
    hibernate.cfg.xml (or hibernate.properties)
    log4j.properties
```

The first file, build.xml, is the Ant build definition. It contains the Ant targets for building and running the application, which we'll discuss next. You'll also add a target that can generate the database schema automatically.

2.1.4 Running and testing the application

To run the application, you need to compile it first and start the database management system with the right database schema.

Ant is a powerful build system for Java. Typically, you'd write a build.xml file for your project and call the build targets you defined in this file with the Ant command-line tool. You can also call Ant targets from your Java IDE, if that is supported.

Compiling the project with Ant

You'll now add a build.xml file and some targets to the "Hello World" project. The initial content for the build file is shown in listing 2.8—you create this file directly in your WORKDIR.

Listing 2.8 A basic Ant build file for "Hello World"

```
<project name="HelloWorld" default="compile" basedir=".">

    <!-- Name of project and version -->
    <property name="proj.name"      value="HelloWorld"/>
    <property name="proj.version"   value="1.0"/>

    <!-- Global properties for this build -->
    <property name="src.java.dir"   value="src"/>
    <property name="lib.dir"        value="lib"/>
    <property name="build.dir"      value="bin"/>

    <!-- Classpath declaration -->
    <path id="project.classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
            <include name="**/*.zip"/>
        </fileset>
    </path>

    <!-- Useful shortcuts -->
    <patternset id="meta.files">
        <include name="**/*.xml"/>
        <include name="**/*.properties"/>
    </patternset>

    <!-- Clean up -->
    <target name="clean">
        <delete dir="${build.dir}"/>
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- Compile Java source -->
    <target name="compile" depends="clean">
        <mkdir dir="${build.dir}"/>
        <javac
            srcdir="${src.java.dir}"
            destdir="${build.dir}"
            nowarn="on">
            <classpath refid="project.classpath"/>
        </javac>
    </target>

    <!-- Copy metadata to build classpath -->
    <target name="copymetafiles">
        <copy todir="${build.dir}">
            <fileset dir="${src.java.dir}">
                <patternset refid="meta.files"/>
            </fileset>
        </copy>
    </target>

    <!-- Run HelloWorld -->
    <target name="run" depends="compile, copymetafiles"
        description="Build and run HelloWorld">
        <java fork="true"
              classname="hello.HelloWorld"
              classpathref="project.classpath">
              <classpath path="${build.dir}"/>
        </java>
    </target>

</project>
```

The first half of this Ant build file contains property settings, such as the project name and global locations of files and directories. You can already see that this build is based on the existing directory layout, your WORKDIR (for Ant, this is the same directory as the basedir). The `default` target, when this build file is called with no named target, is `compile`.

Next, a name that can be easily referenced later, `project.classpath`, is defined as a shortcut to all libraries in the library directory of the project. Another shortcut for a pattern that will come in handy is defined as `meta.files`. You need to handle configuration and metadata files separately in the processing of the build, using this filter.

The `clean` target removes all created and compiled files, and cleans the project. The last three targets, `compile`, `copymetafiles`, and `run`, should be self-explanatory. Running the application depends on the compilation of all Java source files, and the copying of all mapping and property configuration files to the build directory.

Now, execute `ant compile` in your WORKDIR to compile the "Hello World" application. You should see no errors (nor any warnings) during compilation and find your compiled class files in the bin directory. Also call `ant copymetafiles` once, and check whether all configuration and mapping files are copied correctly into the bin directory.

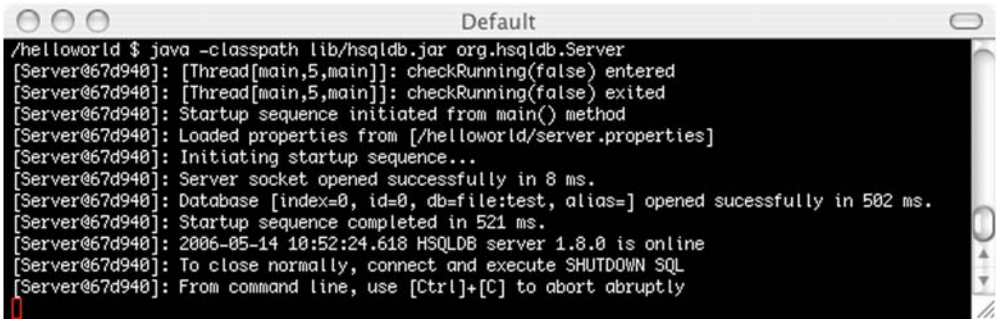
Before you run the application, start the database management system and export a fresh database schema.

Starting the HSQL database system

Hibernate supports more than 25 SQL database management systems out of the box, and support for any unknown dialect can be added easily. If you have an existing database, or if you know basic database administration, you can also replace the configuration options (mostly connection and dialect settings) you created earlier with settings for your own preferred system.

To say hello to the world, you need a lightweight, no-frills database system that is easy to install and configure. A good choice is HSQLDB, an open source SQL database management system written in Java. It can run in-process with the main application, but in our experience, running it stand-alone with a TCP port listening for connections is usually more convenient. You've already copied the hsqldb.jar file into the library directory of your WORKDIR—this library includes both the database engine and the JDBC driver required to connect to a running instance.

To start the HSQLDB server, open up a command line, change into your WORKDIR, and run the command shown in figure 2.4. You should see startup messages and finally a help message that tells you how to shut down the database system (it's OK to use Ctrl+C). You'll also find some new files in your WORKDIR, starting with `test`—these are the files used by HSQLDB to store your data. If you want to start with a fresh database, delete the files between restarts of the server.



```
/helloworld $ java -classpath lib/hsqldb.jar org.hsqldb.Server
[Server@67d940]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@67d940]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@67d940]: Startup sequence initiated from main() method
[Server@67d940]: Loaded properties from [/helloworld/server.properties]
[Server@67d940]: Initiating startup sequence...
[Server@67d940]: Server socket opened successfully in 8 ms.
[Server@67d940]: Database [index=0, id=0, db=file:test, alias=] opened sucessfully in 502 ms.
[Server@67d940]: Startup sequence completed in 521 ms.
[Server@67d940]: 2006-05-14 10:52:24.618 HSQLDB server 1.8.0 is online
[Server@67d940]: To close normally, connect and execute SHUTDOWN SQL
[Server@67d940]: From command line, use [Ctrl]+[C] to abort abruptly
```

Figure 2.4 Starting the HSQLDB server from the command line

You now have an empty database that has no content, not even a schema. Let's create the schema next.

Exporting the database schema

You can create the database schema by hand by writing SQL DDL with `CREATE` statements and executing this DDL on your database. Or (and this is much more convenient) you can let Hibernate take care of this and create a default schema for your application. The prerequisite in Hibernate for automatic generation of SQL DDL is always a Hibernate mapping metadata definition, either in XML mapping files or in Java source-code annotations. We assume that you've designed and implemented your domain model classes and written mapping metadata in XML as you followed the previous sections.

The tool used for schema generation is `hbm2ddl`; its class is `org.hibernate.tool.hbm2ddl.SchemaExport`, so it's also sometimes called `SchemaExport`.

There are many ways to run this tool and create a schema:

- You can run `<hbm2ddl>` in an Ant target in your regular build procedure.
- You can run `SchemaExport` programmatically in application code, maybe in your `HibernateUtil` startup class. This isn't common, however, because you rarely need programmatic control over schema generation.
- You can enable automatic export of a schema when your `SessionFactory` is built by setting the `hibernate.hbm2ddl.auto` configuration property to `create` or `create-drop`. The first setting results in `DROP` statements followed by `CREATE` statements when the `SessionFactory` is built. The second setting adds additional `DROP` statements when the application is shut down and the `SessionFactory` is closed—effectively leaving a clean database after every run.

Programmatic schema generation is straightforward:

```
Configuration cfg = new Configuration().configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

A new `SchemaExport` object is created from a `Configuration`; all settings (such as the database driver, connection URL, and so on) are passed to the `SchemaExport` constructor. The `create(false, true)` call triggers the DDL generation process, without any SQL printed to `stdout` (because of the `false` setting), but with DDL immediately executed in the database (`true`). See the `SchemaExport` API for more information and additional settings.

Your development process determines whether you should enable automatic schema export with the `hibernate.hbm2ddl.auto` configuration setting. Many new Hibernate users find the automatic dropping and re-creation on `Session-Factory` build a little confusing. Once you're more familiar with Hibernate, we encourage you to explore this option for fast turnaround times in integration testing.

An additional option for this configuration property, `update`, can be useful during development: it enables the built-in `SchemaUpdate` tool, which can make schema evolution easier. If enabled, Hibernate reads the JDBC database metadata on startup and creates new tables and constraints by comparing the old schema with the current mapping metadata. Note that this functionality depends on the quality of the metadata provided by the JDBC driver, an area in which many drivers are lacking. In practice, this feature is therefore less exciting and useful than it sounds.

WARNING We've seen Hibernate users trying to use `SchemaUpdate` to update the schema of a production database automatically. This can quickly end in disaster and won't be allowed by your DBA.

You can also run `SchemaUpdate` programmatically:

```
Configuration cfg = new Configuration().configure();
SchemaUpdate schemaUpdate = new SchemaUpdate(cfg);
schemaUpdate.execute(false);
```

The `false` setting at the end again disables printing of the SQL DDL to the console and only executes the statements directly on the database. If you export the DDL to the console or a text file, your DBA may be able to use it as a starting point to produce a quality schema-evolution script.

Another `hbm2ddl.auto` setting useful in development is `validate`. It enables `SchemaValidator` to run at startup. This tool can compare your mapping against the JDBC metadata and tell you if the schema and mappings match. You can also run `SchemaValidator` programmatically:

```
Configuration cfg = new Configuration().configure();
new SchemaValidator(cfg).validate();
```

An exception is thrown if a mismatch between the mappings and the database schema is detected.

Because you're basing your build system on Ant, you'll ideally add a `schemaexport` target to your Ant build that generates and exports a fresh schema for your database whenever you need one (see listing 2.9).

Listing 2.9 Ant target for schema export

```
<taskdef name="hibernatetool"
         classname="org.hibernate.tool.ant.HibernateToolTask"
         classpathref="project.classpath"/>

<target name="schemaexport" depends="compile, copymetafiles"
    description="Exports a generated schema to DB and file">

    <hibernatetool destdir="${basedir}">
        <classpath path="${build.dir}"/>

        <configuration
            configurationfile="${build.dir}/hibernate.cfg.xml"/>

        <hbm2ddl
            drop="true"
            create="true"
            export="true"
            outputfilename="helloworld-ddl.sql"
            delimiter=";"
            format="true"/>

    </hibernatetool>

</target>
```

In this target, you first define a new Ant task that you'd like to use, `HibernateToolTask`. This is a generic task that can do many things—exporting an SQL DDL schema from Hibernate mapping metadata is only one of them. You'll use it throughout this chapter in all Ant builds. Make sure you include all Hibernate libraries, required third-party libraries, and your JDBC driver in the classpath of the task definition. You also need to add the hibernate-tools.jar file, which can be found in the Hibernate Tools download package.

The `schemaexport` Ant target uses this task, and it also depends on the compiled classes and copied configuration files in the build directory. The basic use of the `<hibernatetool>` task is always the same: A *configuration* is the starting point for all code artifact generation. The variation shown here, `<configuration>`, understands Hibernate XML configuration files and reads all Hibernate XML mapping metadata files listed in the given configuration. From that information, an internal Hibernate metadata model (which is what *hbm* stands for everywhere) is produced, and this model data is then processed subsequently by exporters. We discuss tool configurations that can read annotations or a database for reverse engineering later in this chapter.

The other element in the target is a so-called *exporter*. The tool configuration feeds its metadata information to the exporter you selected; in the preceding example, it's the `<hbm2ddl>` exporter. As you may have guessed, this exporter understands the Hibernate metadata model and produces SQL DDL. You can control the DDL generation with several options:

- The exporter generates SQL, so it's mandatory that you set an SQL dialect in your Hibernate configuration file.
- If `drop` is set to `true`, SQL `DROP` statements will be generated first, and all tables and constraints are removed if they exist. If `create` is set to `true`, SQL `CREATE` statements are generated next, to create all tables and constraints. If you enable both options, you effectively drop and re-create the database schema on every run of the Ant target.
- If `export` is set to `true`, all DDL statements are directly executed in the database. The exporter opens a connection to the database using the connection settings found in your configuration file.
- If an `outputfilename` is present, all DDL statements are written to this file, and the file is saved in the `destdir` you configured. The `delimiter` character is appended to all SQL statements written to the file, and if `format` is enabled, all SQL statements are nicely indented.

You can now generate, print, and directly export the schema to a text file and the database by running `ant  schemaxport` in your WORKDIR. All tables and constraints are dropped and then created again, and you have a fresh database ready. (Ignore any error message that says that a table couldn't be dropped because it didn't exist.)

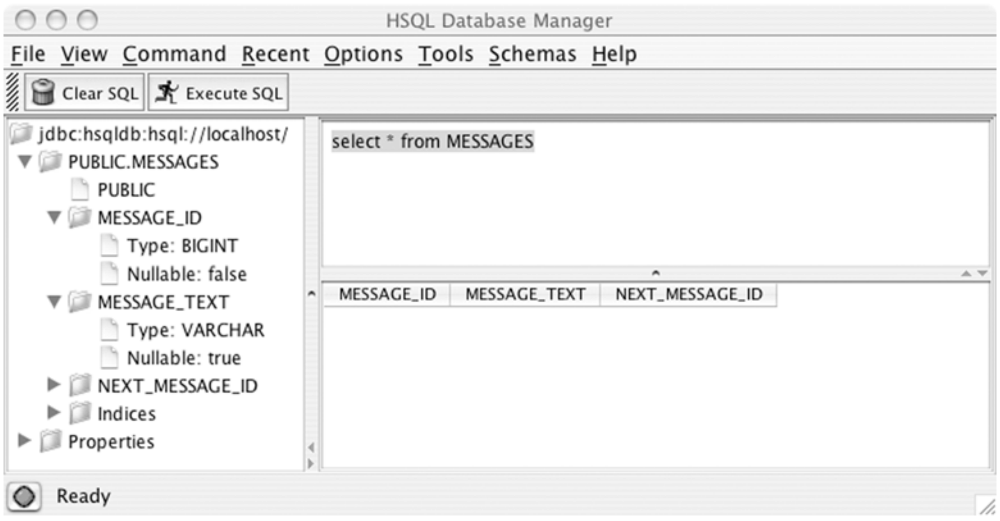
Check that your database is running and that it has the correct database schema. A useful tool included with HSQLDB is a simple database browser. You can call it with the following Ant target:

```
<target name="dbmanager" description="Start HSQLDB manager">
    <java
        classname="org.hsqldb.util.DatabaseManagerSwing"
        fork="yes"
        classpathref="project.classpath"
        failonerror="true">
        <arg value="-url"/>
        <arg value="jdbc:hsqldb:hsql://localhost/"/>
        <arg value="-driver"/>
        <arg value="org.hsqldb.jdbcDriver"/>
    </java>
</target>
```

You should see the schema shown in figure 2.5 after logging in.

Run your application with `ant run`, and watch the console for Hibernate log output. You should see your messages being stored, loaded, and printed. Fire an SQL query in the HSQLDB browser to check the content of your database directly.

You now have a working Hibernate infrastructure and Ant project build. You could skip to the next chapter and continue writing and mapping more complex business classes. However, we recommend that you spend some time with the



```
select * from MESSAGES
```

Figure 2.5 The HSQLDB browser and SQL console

"Hello World" application and extend it with more functionality. You can, for example, try different HQL queries or logging options. Don't forget that your database system is still running in the background, and that you have to either export a fresh schema or stop it and delete the database files to get a clean and empty database again.

In the next section, we walk through the "Hello World" example again, with Java Persistence interfaces and EJB 3.0.

2.2 Starting a Java Persistence project

In the following sections, we show you some of the advantages of JPA and the new EJB 3.0 standard, and how annotations and the standardized programming interfaces can simplify application development, even when compared with Hibernate. Obviously, designing and linking to standardized interfaces is an advantage if you ever need to port or deploy an application on a different runtime environment. Besides portability, though, there are many good reasons to give JPA a closer look.

We'll now guide you through another "Hello World" example, this time with Hibernate Annotations and Hibernate EntityManager. You'll reuse the basic project infrastructure introduced in the previous section so you can see where JPA differs from Hibernate. After working with annotations and the JPA interfaces, we'll show how an application integrates and interacts with other managed components—EJBs. We'll discuss many more application design examples later in the book; however, this first glimpse will let you decide on a particular approach as soon as possible.

2.2.1 Using Hibernate Annotations

Let's first use Hibernate Annotations to replace the Hibernate XML mapping files with inline metadata. You may want to copy your existing "Hello World" project directory before you make the following changes—you'll migrate from native Hibernate to standard JPA mappings (and program code later on).

Copy the Hibernate Annotations libraries to your WORKDIR/lib directory—see the Hibernate Annotations documentation for a list of required libraries. (At the time of writing, hibernate-annotations.jar and the API stubs in ejb3-persistence.jar were required.)

Now delete the src/hello/Message.hbm.xml file. You'll replace this file with annotations in the src/hello/Message.java class source, as shown in listing 2.10.

Listing 2.10 Mapping the `Message` class with annotations

```
package hello;

import javax.persistence.*;

@Entity
@Table(name = "MESSAGES")
public class Message {

    @Id @GeneratedValue
    @Column(name = "MESSAGE_ID")
    private Long id;

    @Column(name = "MESSAGE_TEXT")
    private String text;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "NEXT_MESSAGE_ID")
    private Message nextMessage;

    private Message() {}

    public Message(String text) {
        this.text = text;
    }

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

     public Message getNextMessage() {
         return nextMessage;
     }
     public void setNextMessage(Message nextMessage) {
         this.nextMessage = nextMessage;
     }
}
```

The first thing you'll probably notice in this updated business class is the import of the `javax.persistence` interfaces. Inside this package are all the standardized JPA annotations you need to map the `@Entity` class to a database `@Table`. You put annotations on the private fields of the class, starting with @Id and @Generated-Value for the database identifier mapping. The JPA persistence provider detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you'd enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy.

Note that the @Table, @Column, and @JoinColumn annotations aren't necessary. All properties of an entity are automatically considered persistent, with default strategies and table/column names. You add them here for clarity and to get the same results as with the XML mapping file. Compare the two mapping metadata strategies now, and you'll see that annotations are much more convenient and reduce the lines of metadata significantly. Annotations are also type-safe, they support autocompletion in your IDE as you type (like any other Java interfaces), and they make refactoring of classes and properties easier.

If you're worried that the import of the JPA interfaces will bind your code to this package, you should know that it's only required on your classpath when the annotations are used by Hibernate at runtime. You can load and execute this class without the JPA interfaces on your classpath as long as you don't want to load and store instances with Hibernate.

A second concern that developers new to annotations sometimes have relates to the inclusion of *configuration metadata* in Java source code. By definition, configuration metadata is metadata that can change for each deployment of the application, such as table names. JPA has a simple solution: You can override or replace all annotated metadata with XML metadata files. Later in the book, we'll show you how this is done.

Let's assume that this is all you want from JPA—annotations instead of XML. You don't want to use the JPA programming interfaces or query language; you'll use Hibernate Session and HQL. The only other change you need to make to your project, besides deleting the now obsolete XML mapping file, is a change in the Hibernate configuration, in hibernate.cfg.xml:

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->

    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>
```

```
</session-factory>
</hibernate-configuration>
```

The Hibernate configuration file previously had a list of all XML mapping files. This has been replaced with a list of all annotated classes. If you use programmatic configuration of a `SessionFactory`, the `addAnnotatedClass()` method replaces the `addResource()` method:

```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically

cfg.addAnnotatedClass(hello.Message.class);

SessionFactory sessionFactory = cfg.buildSessionFactory();
```

Note that you have now used `AnnotationConfiguration` instead of the basic Hibernate `Configuration` interface—this extension understands annotated classes. At a minimum, you also need to change your initializer in `HibernateUtil` to use that interface. If you export the database schema with an Ant target, replace `<configuration>` with `<annotationconfiguration>` in your build.xml file.

This is all you need to change to run the "Hello World" application with annotations. Try running it again, probably with a fresh database.

Annotation metadata can also be global, although you don't need this for the "Hello World" application. Global annotation metadata is placed in a file named package-info.java in a particular package directory. In addition to listing annotated classes, you need to add the packages that contain global metadata to your configuration. For example, in a Hibernate XML configuration file, you need to add the following:

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->

    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>

    <!-- List of packages with package-info.java -->
    <mapping package="hello"/>

</session-factory>
</hibernate-configuration>
```

Or you could achieve the same results with programmatic configuration:

```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically

cfg.addClass(hello.Message.class);

cfg.addPackage("hello");

SessionFactory sessionFactory = cfg.buildSessionFactory();
```

Let's take this one step further and replace the native Hibernate code that loads and stores messages with code that uses JPA. With Hibernate Annotations *and* Hibernate EntityManager, you can create portable and standards-compliant mappings and data access code.

2.2.2 Using Hibernate EntityManager

Hibernate EntityManager is a wrapper around Hibernate Core that provides the JPA programming interfaces, supports the JPA entity instance lifecycle, and allows you to write queries with the standardized Java Persistence query language. Because JPA functionality is a subset of Hibernate's native capabilities, you may wonder why you should use the EntityManager package on top of Hibernate. We'll present a list of advantages later in this section, but you'll see one particular simplification as soon as you configure your project for Hibernate EntityManager: You no longer have to list all annotated classes (or XML mapping files) in your configuration file.

Let's modify the "Hello World" project and prepare it for full JPA compatibility.

Basic JPA configuration

A `SessionFactory` represents a particular logical data-store configuration in a Hibernate application. The `EntityManagerFactory` has the same role in a JPA application, and you configure an `EntityManagerFactory` (EMF) either with configuration files or in application code just as you would configure a `SessionFactory`. The configuration of an EMF, together with a set of mapping metadata (usually annotated classes), is called the *persistence unit.*

The notion of a persistence unit also includes the packaging of the application, but we want to keep this as simple as possible for "Hello World"; we'll assume that you want to start with a standardized JPA configuration and no special packaging. Not only the content, but also the name and location of the JPA configuration file for a persistence unit are standardized.

Create a directory named WORKDIR/etc/META-INF and place the basic configuration file named persistence.xml, shown in listing 2.11, in that directory:

Listing 2.11 Persistence unit configuration file

```
<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_1_0.xsd"
   version="1.0">

   <persistence-unit name="helloworld">
      <properties>
          <property name="hibernate.ejb.cfgfile"
              value="/hibernate.cfg.xml"/>
      </properties>
   </persistence-unit>

</persistence>
```

Every persistence unit needs a name, and in this case it's `helloworld`.

NOTE The XML header in the preceding persistence unit configuration file declares what schema should be used, and it's always the same. We'll omit it in future examples and assume that you'll add it.

A persistence unit is further configured with an arbitrary number of properties, which are all vendor-specific. The property in the previous example, `hibernate.ejb.cfgfile`, acts as a catchall. It refers to a `hibernate.cfg.xml` file (in the root of the classpath) that contains all settings for this persistence unit—you're reusing the existing Hibernate configuration. Later, you'll move all configuration details into the `persistence.xml` file, but for now you're more interested in running "Hello World" with JPA.

The JPA standard says that the persistence.xml file needs to be present in the META-INF directory of a deployed persistence unit. Because you aren't really packaging and deploying the persistence unit, this means that you have to copy persistence.xml into a META-INF directory of the build output directory. Modify your build.xml, and add the following to the `copymetafiles` target:

```
<property name="src.etc.dir" value="etc"/>

<target name="copymetafiles">

    <!-- Copy metadata to build -->
    <copy todir="${build.dir}">
      <fileset dir="${src.java.dir}">
         <patternset refid="meta.files"/>
      </fileset>
    </copy>
    <!-- Copy configuration files from etc/ -->
    <copy todir="${build.dir}">
      <fileset dir="${src.etc.dir}">
         <patternset refid="meta.files"/>
      </fileset>
    </copy>

</target>
```

Everything found in WORKDIR/etc that matches the `meta.files` pattern is copied to the build output directory, which is part of the classpath at runtime.

Let's rewrite the main application code with JPA.

"Hello World" with JPA

These are your primary programming interfaces in Java Persistence:

- `javax.persistence.Persistence`—A startup class that provides a static method for the creation of an `EntityManagerFactory`.
- `javax.persistence.EntityManagerFactory`—The equivalent to a Hibernate `SessionFactory`. This runtime object represents a particular persistence unit. It's thread-safe, is usually handled as a singleton, and provides methods for the creation of `EntityManager` instances.
- `javax.persistence.EntityManager`—The equivalent to a Hibernate `Session`. This single-threaded, nonshared object represents a particular unit of work for data access. It provides methods to manage the lifecycle of entity instances and to create `Query` instances.
- `javax.persistence.Query`—This is the equivalent to a Hibernate `Query`. An object is a particular JPA query language or native SQL query representation, and it allows safe binding of parameters and provides various methods for the execution of the query.
- `javax.persistence.EntityTransaction`—This is the equivalent to a Hibernate `Transaction`, used in Java SE environments for the demarcation of `RESOURCE_LOCAL` transactions. In Java EE, you rely on the standardized `javax.transaction.UserTransaction` interface of JTA for programmatic transaction demarcation.

To use the JPA interfaces, you need to copy the required libraries to your WORKDIR/lib directory; check the documentation bundled with Hibernate EntityManager for an up-to-date list. You can then rewrite the code in WORKDIR/src/hello/HelloWorld.java and switch from Hibernate to JPA interfaces (see listing 2.12).

Listing 2.12 The "Hello World" main application code with JPA

```
package hello;

import java.util.*;
import javax.persistence.*;

public class HelloWorld {

    public static void main(String[] args) {

        // Start EntityManagerFactory
        EntityManagerFactory emf =
               Persistence.createEntityManagerFactory("helloworld");

        // First unit of work
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();

        Message message = new Message("Hello World");
        em.persist(message);

        tx.commit();
        em.close();

        // Second unit of work
        EntityManager newEm = emf.createEntityManager();
        EntityTransaction newTx = newEm.getTransaction();
        newTx.begin();

        List messages = newEm
            .createQuery("select m from Message m
        ➥  order by m.text asc")
            .getResultList();


        System.out.println( messages.size() + " message(s) found" );

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }

        newTx.commit();
        newEm.close();

        // Shutting down the application
        emf.close();
    }
}
```

The first thing you probably notice in this code is that there is no Hibernate import anymore, only `javax.peristence.*`. The `EntityManagerFactory` is created with a static call to `Persistence` and the name of the persistence unit. The rest of the code should be self-explanatory—you use JPA just like Hibernate, though there are some minor differences in the API, and methods have slightly different names. Furthermore, you didn't use the `HibernateUtil` class for static initialization of the infrastructure; you can write a `JPAUtil` class and move the creation of an `EntityManagerFactory` there if you want, or you can remove the now unused `WORKDIR/src/persistence` package.

JPA also supports programmatic configuration, with a map of options:

```
Map myProperties = new HashMap();
myProperties.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory emf =
 Persistence.createEntityManagerFactory("helloworld", myProperties);
```

Custom programmatic properties override any property you've set in the persistence.xml configuration file.

Try to run the ported `HelloWorld` code with a fresh database. You should see the exact same log output on your screen as you did with native Hibernate—the JPA persistence provider engine is Hibernate.

Automatic detection of metadata

We promised earlier that you won't have to list all your annotated classes or XML mapping files in the configuration, but it's still there, in hibernate.cfg.xml. Let's enable the autodetection feature of JPA.

Run the "Hello World" application again after switching to `DEBUG` logging for the `org.hibernate` package. Some additional lines should appear in your log:

```
...
Ejb3Configuration:141
    - Trying to find persistence unit: helloworld
Ejb3Configuration:150
    - Analyse of persistence.xml:
        file:/helloworld/build/META-INF/persistence.xml
PersistenceXmlLoader:115
    - Persistent Unit name from persistence.xml: helloworld
Ejb3Configuration:359
    - Detect class: true; detect hbm: true
JarVisitor:178
    - Searching mapped entities in jar/par: file:/helloworld/build
JarVisitor:217
    - Filtering: hello.HelloWorld
JarVisitor:217
    - Filtering: hello.Message
```

```
JarVisitor:255
    - Java element filter matched for hello.Message
Ejb3Configuration:101
    - Creating Factory: helloworld
...
```

On startup, the `Persistence.createEntityManagerFactory()` method tries to locate the persistence unit named `helloworld`. It searches the classpath for all META-INF/persistence.xml files and then configures the EMF if a match is found. The second part of the log shows something you probably didn't expect. The JPA persistence provider tried to find all annotated classes and all Hibernate XML mapping files in the build output directory. The list of annotated classes (or the list of XML mapping files) in hibernate.cfg.xml isn't needed, because `hello.Message`, the annotated entity class, has already been found.

Instead of removing only this single unnecessary option from hibernate.cfg.xml, let's remove the whole file and move all configuration details into persistence.xml (see listing 2.13).

Listing 2.13 Full persistence unit configuration file

```
<persistence-unit name="helloworld">

   <provider>org.hibernate.ejb.HibernatePersistence</provider>

   <!-- Not needed, Hibernate supports auto-detection in JSE
       <class>hello.Message</class>
    -->

  <properties>
      <property name="hibernate.archive.autodetection"
          value="class, hbm"/>

      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>

      <property name="hibernate.connection.driver_class"
                value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.url"
                value="jdbc:hsqldb:hsql://localhost"/>
      <property name="hibernate.connection.username"
                value="sa"/>

      <property name="hibernate.c3p0.min_size"
                value="5"/>
      <property name="hibernate.c3p0.max_size"
                value="20"/>
      <property name="hibernate.c3p0.timeout"
                value="300"/>
      <property name="hibernate.c3p0.max_statements"
                value="50"/>
```

```
        <property name="hibernate.c3p0.idle_test_period"
                  value="3000"/>

        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.HSQLDialect"/>

        <property name="hibernate.hbm2ddl.auto" value="create"/>

    </properties>
  </persistence-unit>
```

There are three interesting new elements in this configuration file. First, you set an explicit `<provider>` that should be used for this persistence unit. This is usually required only if you work with several JPA implementations at the same time, but we hope that Hibernate will, of course, be the only one. Next, the specification requires that you list all annotated classes with `<class>` elements if you deploy in a non-Java EE environment—Hibernate supports autodetection of mapping metadata everywhere, making this optional. Finally, the Hibernate configuration setting `archive.autodetection` tells Hibernate what metadata to scan for automatically: annotated classes (`class`) and/or Hibernate XML mapping files (`hbm`). By default, Hibernate EntityManager scans for both. The rest of the configuration file contains all options we explained and used earlier in this chapter in the regular hibernate.cfg.xml file.

Automatic detection of annotated classes and XML mapping files is a great feature of JPA. It's usually only available in a Java EE application server; at least, this is what the EJB 3.0 specification guarantees. But Hibernate, as a JPA provider, also implements it in plain Java SE, though you may not be able to use the exact same configuration with any other JPA provider.

You've now created an application that is fully JPA specification-compliant. Your project directory should look like this (note that we also moved log4j.properties to the etc/ directory):

```
WORKDIR
+etc
  log4j.properties
  +META-INF
   persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java
```

All JPA configuration settings are bundled in persistence.xml, all mapping metadata is included in the Java source code of the `Message` class, and Hibernate automatically scans and finds the metadata on startup. Compared to pure Hibernate, you now have these benefits:

- Automatic scanning of deployed metadata, an important feature in large projects. Maintaining a list of annotated classes or mapping files becomes difficult if hundreds of entities are developed by a large team.
- Standardized and simplified configuration, with a standard location for the configuration file, and a deployment concept—the persistence unit—that has many more advantages in larger projects that wrap several units (JARs) in an application archive (EAR).
- Standardized data access code, entity instance lifecycle, and queries that are fully portable. There is no proprietary import in your application.

These are only some of the advantages of JPA. You'll see its real power if you combine it with the full EJB 3.0 programming model and other managed components.

2.2.3 Introducing EJB components

Java Persistence starts to shine when you also work with EJB 3.0 session beans and message-driven beans (and other Java EE 5.0 standards). The EJB 3.0 specification has been designed to permit the integration of persistence, so you can, for example, get automatic transaction demarcation on bean method boundaries, or a persistence context (think `Session`) that spans the lifecycle of a stateful session EJB.

This section will get you started with EJB 3.0 and JPA in a managed Java EE environment; you'll again modify the "Hello World" application to learn the basics. You need a Java EE environment first—a runtime container that provides Java EE services. There are two ways you can get it:

- You can install a full Java EE 5.0 application server that supports EJB 3.0 and JPA. Several open source (Sun GlassFish, JBoss AS, ObjectWeb EasyBeans) and other proprietary licensed alternatives are on the market at the time of writing, and probably more will be available when you read this book.
- You can install a modular server that provides only the services you need, selected from the full Java EE 5.0 bundle. At a minimum, you probably want an EJB 3.0 container, JTA transaction services, and a JNDI registry. At the time of writing, only JBoss AS provided modular Java EE 5.0 services in an easily customizable package.

To keep things simple and to show you how easy it is to get started with EJB 3.0, you'll install and configure the modular JBoss Application Server and enable only the Java EE 5.0 services you need.

Installing the EJB container

Go to http://jboss.com/products/ejb3, download the modular embeddable server, and unzip the downloaded archive. Copy all libraries that come with the server into your project's WORKDIR/lib directory, and copy all included configuration files to your WORKDIR/src directory. You should now have the following directory layout:

```
WORKDIR
+etc
  default.persistence.properties
  ejb3-interceptors-aop.xml
  embedded-jboss-beans.xml
  jndi.properties
  log4j.properties
  +META-INF
    helloworld-beans.xml
    persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java
```

The JBoss embeddable server relies on Hibernate for Java Persistence, so the default.persistence.properties file contains default settings for Hibernate that are needed for all deployments (such as JTA integration settings). The ejb3-interceptors-aop.xml and embedded-jboss-beans.xml configuration files contain the services configuration of the server—you can look at these files, but you don't need to modify them now. By default, at the time of writing, the enabled services are JNDI, JCA, JTA, and the EJB 3.0 container—exactly what you need.

To migrate the "Hello World" application, you need a managed datasource, which is a database connection that is handled by the embeddable server. The easiest way to configure a managed datasource is to add a configuration file that deploys the datasource as a managed service. Create the file in listing 2.14 as WORKDIR/etc/META-INF/helloworld-beans.xml.

Listing 2.14 Datasource configuration file for the JBoss server

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
  xmlns="urn:jboss:bean-deployer:2.0">

  <!-- Enable a JCA datasource available through JNDI -->
  <bean name="helloWorldDatasourceFactory"
```

```
     class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">

        <property name="jndiName">java:/HelloWorldDS</property>

        <!-- HSQLDB -->
        <property name="driverClass">
            org.hsqldb.jdbcDriver
        </property>
        <property name="connectionURL">
            jdbc:hsqldb:hsql://localhost
        </property>
        <property name="userName">sa</property>

        <property name="minSize">0</property>
        <property name="maxSize">10</property>
        <property name="blockingTimeout">1000</property>
        <property name="idleTimeout">100000</property>

        <property name="transactionManager">
            <inject bean="TransactionManager"/>
        </property>
        <property name="cachedConnectionManager">
            <inject bean="CachedConnectionManager"/>
        </property>
        <property name="initialContextProperties">
            <inject bean="InitialContextProperties"/>
        </property>
    </bean>

    <bean name="HelloWorldDS" class="java.lang.Object">
        <constructor factoryMethod="getDatasource">
            <factory bean="helloWorldDatasourceFactory"/>
        </constructor>
    </bean>

</deployment>
```

Again, the XML header and schema declaration aren't important for this example. You set up two beans: The first is a factory that can produce the second type of bean. The `LocalTxDataSource` is effectively now your database connection pool, and all your connection pool settings are available on this factory. The factory binds a managed datasource under the JNDI name `java:/HelloWorldDS`.

The second bean configuration declares how the registered object named `HelloWorldDS` should be instantiated, if another service looks it up in the JNDI registry. Your "Hello World" application asks for the datasource under this name, and the server calls `getDatasource()` on the `LocalTxDataSource` factory to obtain it.

Also note that we added some line breaks in the property values to make this more readable—you shouldn't do this in your real configuration file (unless your database username contains a line break).

Configuring the persistence unit

Next, you need to change the persistence unit configuration of the "Hello World" application to access a managed JTA datasource, instead of a resource-local connection pool. Change your WORKDIR/etc/META-INF/persistence.xml file as follows:

```
<persistence ...>

   <persistence-unit name="helloworld">
      <jta-data-source>java:/HelloWorldDS</jta-data-source>
      <properties>
          <property name="hibernate.show_sql" value="true"/>
          <property name="hibernate.format_sql" value="true"/>
          <property name="hibernate.dialect"
                    value="org.hibernate.dialect.HSQLDialect"/>
          <property name="hibernate.hbm2ddl.auto" value="create"/>

      </properties>
   </persistence-unit>

</persistence>
```

You removed many Hibernate configuration options that are no longer relevant, such as the connection pool and database connection settings. Instead, you set a `<jta-data-source>` property with the name of the datasource as bound in JNDI. Don't forget that you still need to configure the correct SQL dialect and any other Hibernate options that aren't present in default.persistence.properties.

The installation and configuration of the environment is now complete, (we'll show you the purpose of the jndi.properties files in a moment) and you can rewrite the application code with EJBs.

Writing EJBs

There are many ways to design and create an application with managed components. The "Hello World" application isn't sophisticated enough to show elaborate examples, so we'll introduce only the most basic type of EJB, a *stateless session bean.* (You've already seen entity classes—annotated plain Java classes that can have persistent instances. Note that the term *entity bean* only refers to the old EJB 2.1 entity beans; EJB 3.0 and Java Persistence standardize a lightweight programming model for plain *entity classes.*)

Every EJB session bean needs a *business interface*. This isn't a special interface that needs to implement predefined methods or extend existing ones; it's plain Java. Create the following interface in the WORKDIR/src/hello package:

```
package hello;

public interface MessageHandler {

    public void saveMessages();

    public void showMessages();
}
```

A MessageHandler can save and show messages; it's straightforward. The actual EJB implements this business interface, which is by default considered a local interface (that is, remote EJB clients cannot call it); see listing 2.15.

Listing 2.15 The "Hello World" EJB session bean application code

```
package hello;

import javax.ejb.Stateless;
import javax.persistence.*;
import java.util.List;

@Stateless
public class MessageHandlerBean implements MessageHandler {

    @PersistenceContext
    EntityManager em;

    public void saveMessages() {
        Message message = new Message("Hello World");
        em.persist(message);
    }

    public void showMessages() {
        List messages =
            em.createQuery("select m from Message m
          ➥ order by m.text asc")
                .getResultList();

        System.out.println(messages.size() + " message(s) found:");

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }

    }
}
```

There are several interesting things to observe in this implementation. First, it's a plain Java class with no hard dependencies on any other package. It becomes an EJB only with a single metadata annotation, `@Stateless`. EJBs support container-managed services, so you can apply the `@PersistenceContext` annotation, and the server injects a fresh `EntityManager` instance whenever a method on this stateless bean is called. Each method is also assigned a transaction automatically by the container. The transaction starts when the method is called, and commits when the method returns. (It would be rolled back when an exception is thrown inside the method.)

You can now modify the `HelloWorld` main class and delegate all the work of storing and showing messages to the `MessageHandler`.

Running the application

The main class of the "Hello World" application calls the `MessageHandler` stateless session bean after looking it up in the JNDI registry. Obviously, the managed environment and the whole application server, including the JNDI registry, must be booted first. You do all of this in the `main()` method of `HelloWorld.java` (see listing 2.16).

Listing 2.16 "Hello World" main application code, calling EJBs

```
package hello;

import org.jboss.ejb3.embedded.EJB3StandaloneBootstrap;
import javax.naming.InitialContext;

public class HelloWorld {

  public static void main(String[] args) throws Exception {

    // Boot the JBoss Microcontainer with EJB3 settings, automatically
    // loads ejb3-interceptors-aop.xml and embedded-jboss-beans.xml
    EJB3StandaloneBootstrap.boot(null);

    // Deploy custom stateless beans (datasource, mostly)
    EJB3StandaloneBootstrap
        .deployXmlResource("META-INF/helloworld-beans.xml");

    // Deploy all EJBs found on classpath (slow, scans all)
    // EJB3StandaloneBootstrap.scanClasspath();

    // Deploy all EJBs found on classpath (fast, scans build directory)
    // This is a relative location, matching the substring end of one
    // of java.class.path locations. Print out the value of
    // System.getProperty("java.class.path") to see all paths.
    EJB3StandaloneBootstrap.scanClasspath("helloworld-ejb3/bin");

    // Create InitialContext from jndi.properties
```

```
        InitialContext initialContext = new InitialContext();

        // Look up the stateless MessageHandler EJB
        MessageHandler msgHandler = (MessageHandler) initialContext
                                 .lookup("MessageHandlerBean/local");

        // Call the stateless EJB
        msgHandler.saveMessages();
        msgHandler.showMessages();

        // Shut down EJB container
        EJB3StandaloneBootstrap.shutdown();
    }
}
```

The first command in `main()` boots the server's kernel and deploys the base services found in the service configuration files. Next, the datasource factory configuration you created earlier in helloworld-beans.xml is deployed, and the datasource is bound to JNDI by the container. From that point on, the container is ready to deploy EJBs. The easiest (but often not the fastest) way to deploy all EJBs is to let the container search the whole classpath for any class that has an EJB annotation. To learn about the many other deployment options available, check the JBoss AS documentation bundled in the download.

To look up an EJB, you need an `InitialContext`, which is your entry point for the JNDI registry. If you instantiate an `InitialContext`, Java automatically looks for the file jndi.properties on your classpath. You need to create this file in WORKDIR/etc with settings that match the JBoss server's JNDI registry configuration:

```
java.naming.factory.initial
    ➥ org.jnp.interfaces.LocalOnlyContextFactory
java.naming.factory.url.pkgs org.jboss.naming:org.jnp.interfaces
```

You don't need to know exactly what this configuration means, but it basically points your `InitialContext` to a JNDI registry running in the local virtual machine (remote EJB client calls would require a JNDI service that supports remote communication).

By default, you look up the `MessageHandler` bean by the name of an implementation class, with the `/local` suffix for a local interface. How EJBs are named, how they're bound to JNDI, and how you look them up varies and can be customized. These are the defaults for the JBoss server.

Finally, you call the `MessageHandler` EJB and let it do all the work automatically in two units—each method call will result in a separate transaction.

This completes our first example with managed EJB components and integrated JPA. You can probably already see how automatic transaction demarcation and `EntityManager` injection can improve the readability of your code. Later, we'll show you how stateful session beans can help you implement sophisticated conversations between the user and the application, with transactional semantics. Furthermore, the EJB components don't contain any unnecessary glue code or infrastructure methods, and they're fully reusable, portable, and executable in any EJB 3.0 container.

NOTE *Packaging of persistence units*—We didn't talk much about the packaging of persistence units—you didn't need to package the "Hello World" example for any of the deployments. However, if you want to use features such as hot redeployment on a full application server, you need to package your application correctly. This includes the usual combination of JARs, WARs, EJB-JARs, and EARs. Deployment and packaging is often also vendor-specific, so you should consult the documentation of your application server for more information. JPA persistence units can be scoped to JARs, WARs, and EJB-JARs, which means that one or several of these archives contains all the annotated classes and a META-INF/persistence.xml configuration file with all settings for this particular unit. You can wrap one or several JARs, WARs, and EJB-JARs in a single enterprise application archive, an EAR. Your application server should correctly detect all persistence units and create the necessary factories automatically. With a unit name attribute on the `@PersistenceContext` annotation, you instruct the container to inject an `EntityManager` from a particular unit.

Full portability of an application isn't often a primary reason to use JPA or EJB 3.0. After all, you made a decision to use Hibernate as your JPA persistence provider. Let's look at how you can fall back and use a Hibernate native feature from time to time.

2.2.4 Switching to Hibernate interfaces

You decided to use Hibernate as a JPA persistence provider for several reasons: First, Hibernate is a good JPA implementation that provides many options that don't affect your code. For example, you can enable the Hibernate second-level data cache in your JPA configuration, and transparently improve the performance and scalability of your application without touching any code.

Second, you can use native Hibernate mappings or APIs when needed. We discuss the mixing of mappings (especially annotations) in chapter 3, section 3.3, "Object/relational mapping metadata," but here we want to show how you can use a Hibernate API in your JPA application, when needed. Obviously, importing a Hibernate API into your code makes porting the code to a different JPA provider more difficult. Hence, it becomes critically important to isolate these parts of your code properly, or at least to document why and when you used a native Hibernate feature.

You can fall back to Hibernate APIs from their equivalent JPA interfaces and get, for example, a `Configuration`, a `SessionFactory`, and even a `Session` whenever needed.

For example, instead of creating an `EntityManagerFactory` with the `Persistence` static class, you can use a Hibernate `Ejb3Configuration`:

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
  cfg.configure("/custom/hibernate.cfg.xml")
     .setProperty("hibernate.show_sql", "false")
     .setInterceptor( new MyInterceptor() )
     .addAnnotatedClass( hello.Message.class )
     .addResource( "/Foo.hbm.xml")
     .buildEntityManagerFactory();

AnnotationConfiguration
        hibCfg = cfg.getHibernateConfiguration();
```

The `Ejb3Configuration` is a new interface that duplicates the regular Hibernate `Configuration` instead of extending it (this is an implementation detail). This means you can get a plain `AnnotationConfiguration` object from an `Ejb3Configuration`, for example, and pass it to a `SchemaExport` instance programmatically.

The `SessionFactory` interface is useful if you need programmatic control over the second-level cache regions. You can get a `SessionFactory` by casting the `EntityManagerFactory` first:

```
HibernateEntityManagerFactory hibEMF =
        (HibernateEntityManagerFactory) emf;
SessionFactory sf = hibEMF.getSessionFactory();
```

The same technique can be applied to get a `Session` from an `EntityManager`:

```
HibernateEntityManager hibEM =
    (HibernateEntityManager) em;
Session session = hibEM.getSession();
```

This isn't the only way to get a native API from the standardized `EntityManager`. The JPA specification supports a `getDelegate()` method that returns the underlying implementation:

```
Session session = (Session) entityManager.getDelegate();
```

Or you can get a Session injected into an EJB component (although this only works in the JBoss Application Server):

```
@Stateless
public class MessageHandlerBean implements MessageHandler {

    @PersistenceContext
    Session session;

    ...
}
```

In rare cases, you can fall back to plain JDBC interfaces from the Hibernate Session:

```
Connection jdbcConnection = session.connection();
```

This last option comes with some caveats: You aren't allowed to close the JDBC Connection you get from Hibernate—this happens automatically. The exception to this rule is that in an environment that relies on aggressive connection releases, which means in a JTA or CMT environment, you *have* to close the returned connection in application code.

A better and safer way to access a JDBC connection directly is through resource injection in a Java EE 5.0. Annotate a field or setter method in an EJB, an EJB listener, a servlet, a servlet filter, or even a JavaServer Faces backing bean, like this:

```
@Resource(mappedName="java:/HelloWorldDS") DataSource ds;
```

So far, we've assumed that you work on a new Hibernate or JPA project that involves no legacy application code or existing database schema. We now switch perspectives and consider a development process that is bottom-up. In such a scenario, you probably want to automatically reverse-engineer artifacts from an existing database schema.

2.3 Reverse engineering a legacy database

Your first step when mapping a legacy database likely involves an automatic reverse-engineering procedure. After all, an entity schema already exists in your database system. To make this easier, Hibernate has a set of tools that can read a schema and produce various artifacts from this metadata, including XML mapping files and Java source code. All of this is template-based, so many customizations are possible.

You can control the reverse-engineering process with tools and tasks in your Ant build. The HibernateToolTask you used earlier to export SQL DDL from

Hibernate mapping metadata has many more options, most of which are related to reverse engineering, as to how XML mapping files, Java code, or even whole application skeletons can be generated automatically from an existing database schema.

We'll first show you how to write an Ant target that can load an existing database into a Hibernate metadata model. Next, you'll apply various exporters and produce XML files, Java code, and other useful artifacts from the database tables and columns.

2.3.1 Creating a database configuration

Let's assume that you have a new WORKDIR with nothing but the lib directory (and its usual contents) and an empty src directory. To generate mappings and code from an existing database, you first need to create a configuration file that contains your database connection settings:

```
hibernate.dialect = org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
```

Store this file directly in WORKDIR, and name it helloworld.db.properties. The four lines shown here are the minimum that is required to connect to the database and read the metadata of all tables and columns. You could have created a Hibernate XML configuration file instead of hibernate.properties, but there is no reason to make this more complex than necessary.

Write the Ant target next. In a build.xml file in your project, add the following code:

```
<taskdef name="hibernatetool"
     classname="org.hibernate.tool.ant.HibernateToolTask"
     classpathref="project.classpath"/>

<target name="reveng.hbmxml"
        description="Produces XML mapping files in src directory">

    <hibernatetool destdir="${basedir}/src">

        <jdbcconfiguration
            propertyfile="${basedir}/helloworld.db.properties"
            revengfile="${basedir}/helloworld.reveng.xml"/>

        <hbm2hbmxml/> <!-- Export Hibernate XML files -->
        <hbm2cfgxml/> <!-- Export a hibernate.cfg.xml file -->

    </hibernatetool>

</target>
```

The HibernateToolTask definition for Ant is the same as before. We assume that you'll reuse most of the build file introduced in previous sections, and that references such as project.classpath are the same. The <hibernatetool> task is set with WORKDIR/src as the default destination directory for all generated artifacts.

A <jdbconfiguration> is a Hibernate tool configuration that can connect to a database via JDBC and read the JDBC metadata from the database catalog. You usually configure it with two options: database connection settings (the properties file) and an optional reverse-engineering customization file.

The metadata produced by the tool configuration is then fed to exporters. The example Ant target names two such exporters: the hbm2hbmxml exporter, as you can guess from its name, takes Hibernate metadata (hbm) from a configuration, and generates Hibernate XML mapping files; the second exporter can prepare a hibernate.cfg.xml file that lists all the generated XML mapping files.

Before we talk about these and various other exporters, let's spend a minute on the reverse-engineering customization file and what you can do with it.

2.3.2 *Customizing reverse engineering*

JDBC metadata—that is, the information you can read from a database about itself via JDBC—often isn't sufficient to create a perfect XML mapping file, let alone Java application code. The opposite may also be true: Your database may contain information that you want to ignore (such as particular tables or columns) or that you wish to transform with nondefault strategies. You can customize the reverse-engineering procedure with a reverse-engineering configuration file, which uses an XML syntax.

Let's assume that you're reverse-engineering the "Hello World" database you created earlier in this chapter, with its single MESSAGES table and only a few columns. With a helloworld.reveng.xml file, as shown in listing 2.17, you can customize this reverse engineering.

Listing 2.17 Configuration for customized reverse engineering

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering SYSTEM     ❶
  "http://hibernate.sourceforge.net/
  ➥ hibernate-reverse-engineering-3.0.dtd">

<hibernate-reverse-engineering>

    <table-filter match-name=".*" package="hello"/>     ❷

    <table name="MESSAGES" schema="PUBLIC" class="Message">     ❸

        <primary-key>     ❹
```

```
            <generator class="increment"/>
            <key-column name="MESSAGE_ID" property="id" type="long"/>
        </primary-key>
                                                          ❺
        <column name="MESSAGE_TEXT" property="text"/>  <─┘

        <foreign-key constraint-name="FK_NEXT_MESSAGE">  <─┐
            <many-to-one property="nextMessage"/>          ❻
            <set exclude="true"/>
        </foreign-key>

    </table>

</hibernate-reverse-engineering>
```

❶ This XML file has its own DTD for validation and autocompletion.

❷ A table filter can exclude tables by name with a regular expression. However, in this example, you define a a default package for all classes produced for the tables matching the regular expression.

❸ You can customize individual tables by name. The schema name is usually optional, but HSQLDB assigns the PUBLIC schema to all tables by default so this setting is needed to identify the table when the JDBC metadata is retrieved. You can also set a custom class name for the generated entity here.

❹ The primary key column generates a property named `id`, the default would be `messageId`. You also explicitly declare which Hibernate identifier generator should be used.

❺ An individual column can be excluded or, in this case, the name of the generated property can be specified—the default would be `messageText`.

❻ If the foreign key constraint `FK_NEXT_MESSAGE` is retrieved from JDBC metadata, a many-to-one association is created by default to the target entity of that class. By matching the foreign key constraint by name, you can specify whether an inverse collection (one-to-many) should also be generated (the example excludes this) and what the name of the many-to-one property should be.

If you now run the Ant target with this customization, it generates a Message.hbm.xml file in the `hello` package in your source directory. (You need to copy the Freemarker and jTidy JAR files into your library directory first.) The customizations you made result in the same Hibernate mapping file you wrote earlier by hand, shown in listing 2.2.

In addition to the XML mapping file, the Ant target also generates a Hibernate XML configuration file in the source directory:

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
        org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
        jdbc:hsqldb:hsql://localhost
    </property>
    <property name="hibernate.connection.username">
        sa
    </property>
    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>

    <mapping resource="hello/Message.hbm.xml" />

  </session-factory>
</hibernate-configuration>
```

The exporter writes all the database connection settings you used for reverse engineering into this file, assuming that this is the database you want to connect to when you run the application. It also adds all generated XML mapping files to the configuration.

What is your next step? You can start writing the source code for the `Message` Java class. Or you can let the Hibernate Tools generate the classes of the domain model for you.

2.3.3 Generating Java source code

Let's assume you have an existing Hibernate XML mapping file for the `Message` class, and you'd like to generate the source for the class. As discussed in chapter 3, a plain Java entity class ideally implements `Serializable`, has a no-arguments constructor, has getters and setters for all properties, and has an encapsulated implementation.

Source code for entity classes can be generated with the Hibernate Tools and the `hbm2java` exporter in your Ant build. The source artifact can be anything that can be read into a Hibernate metadata model—Hibernate XML mapping files are best if you want to customize the Java code generation.

Add the following target to your Ant build:

```
<target name="reveng.pojos"
        description="Produces Java classes from XML mappings">

    <hibernatetool destdir="${basedir}/src">

        <configuration>
```

```
            <fileset dir="${basedir}/src">
                <include name="**/*.hbm.xml"/>
            </fileset>
        </configuration>

        <hbm2java/> <!-- Generate entity class source -->

    </hibernatetool>

</target>
```

The `<configuration>` reads all Hibernate XML mapping files, and the `<hbm2-java>` exporter produces Java source code with the default strategy.

Customizing entity class generation

By default, `hbm2java` generates a simple entity class for each mapped entity. The class implements the `Serializable` marker interface, and it has accessor methods for all properties and the required constructor. All attributes of the class have private visibility for fields, although you can change that behavior with the `<meta>` element and attributes in the XML mapping files.

The first change to the default reverse engineering behavior you make is to restrict the visibility scope for the `Message`'s attributes. By default, all accessor methods are generated with public visibility. Let's say that `Message` objects are immutable; you wouldn't expose the setter methods on the public interface, but only the getter methods. Instead of enhancing the mapping of each property with a `<meta>` element, you can declare a meta-attribute at the class level, thus applying the setting to all properties in that class:

```
<class name="Message"
   table="MESSAGES">

       <meta attribute="scope-set">private</meta>
        ...

</class>
```

The `scope-set` attribute defines the visibility of property setter methods.

The `hbm2java` exporter also accepts meta-attributes on the next higher-level, in the root `<hibernate-mapping>` element, which are then applied to all classes mapped in the XML file. You can also add fine-grained meta-attributes to single property, collection, or component mappings.

One (albeit small) improvement of the generated entity class is the inclusion of the `text` of the `Message` in the output of the generated `toString()` method. The text is a good visual control element in the log output of the application. You can change the mapping of `Message` to include it in the generated code:

```
<property name="text" type="string">
    <meta attribute="use-in-tostring">true</meta>
    <column name="MESSAGE_TEXT" />
</property>
```

The generated code of the `toString()` method in `Message.java` looks like this:

```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(getClass().getName())
        .append("@")
        .append( Integer.toHexString(hashCode()) )
        .append(" [");
        .append("text").append("='").append(getText()).append("' ");
        .append("]");

    return buffer.toString();
}
```

Meta-attributes can be inherited; that is, if you declare a `use-in-tostring` at the level of a `<class>` element, all properties of that class are included in the `toString()` method. This inheritance mechanism works for all `hbm2java` meta-attributes, but you can turn it off selectively:

```
<meta attribute="scope-class" inherit="false">public abstract</meta>
```

Setting `inherit` to `false` in the `scope-class` meta-attribute creates only the parent class of this `<meta>` element as `public abstract`, but not any of the (possibly) nested subclasses.

The `hbm2java` exporter supports, at the time of writing, 17 meta-attributes for fine-tuning code generation. Most are related to visibility, interface implementation, class extension, and predefined Javadoc comments. Refer to the Hibernate Tools documentation for a complete list.

If you use JDK 5.0, you can switch to automatically generated static imports and generics with the `jdk5="true"` setting on the `<hbm2java>` task. Or, you can produce EJB 3.0 entity classes with annotations.

Generating Java Persistence entity classes

Normally, you use either Hibernate XML mapping files or JPA annotations in your entity class source code to define your mapping metadata, so generating Java Persistence entity classes with annotations from XML mapping files doesn't seem reasonable. However, you can create entity class source code with annotations directly from JDBC metadata, and skip the XML mapping step. Look at the following Ant target:

```
<target name="reveng.entities"
        description="Produces Java entity classes in src directory">

    <hibernatetool destdir="${basedir}/src">

        <jdbcconfiguration
            propertyfile="${basedir}/helloworld.db.properties"
            revengfile="${basedir}/helloworld.reveng.xml"/>

        <hbm2java jdk5="true" ejb3="true"/>
        <hbm2cfgxml ejb3="true"/>

    </hibernatetool>

</target>
```

This target generates entity class source code with mapping annotations and a hibernate.cfg.xml file that lists these mapped classes. You can edit the Java source directly to customize the mapping, if the customization in helloworld.reveng.xml is too limited.

Also note that all exporters rely on templates written in the FreeMarker template language. You can customize the templates in whatever way you like, or even write your own. Even programmatic customization of code generation is possible. The Hibernate Tools reference documentation shows you how these options are used.

Other exporters and configurations are available with the Hibernate Tools:

- An `<annotationconfiguration>` replaces the regular `<configuration>` if you want to read mapping metadata from annotated Java classes, instead of XML mapping files. Its only argument is the location and name of a hibernate.cfg.xml file that contains a list of annotated classes. Use this approach to export a database schema from annotated classes.
- An `<ejb3configuration>` is equivalent to an `<annotationconfiguration>`, except that it can scan for annotated Java classes automatically on the classpath; it doesn't need a hibernate.cfg.xml file.
- The `<hbm2dao>` exporter can create additional Java source for a persistence layer, based on the *data access object* pattern. At the time of writing, the templates for this exporter are old and need updating. We expect that the finalized templates will be similar to the DAO code shown in chapter 16, section 16.2, "Creating a persistence layer."
- The `<hbm2doc>` exporter generates HTML files that document the tables and Java entities.

- The `<hbmtemplate>` exporter can be parameterized with a set of custom FreeMarker templates, and you can generate anything you want with this approach. Templates that produce a complete runable skeleton application with the JBoss Seam framework are bundled in the Hibernate Tools.

You can get creative with the import and export functionality of the tools. For example, you can read annotated Java classes with `<annotationconfiguration>` and export them with `<hbm2hbmxml>`. This allows you to develop with JDK 5.0 and the more convenient annotations but deploy Hibernate XML mapping files in production (on JDK 1.4).

Let's finish this chapter with some more advanced configuration options and integrate Hibernate with Java EE services.

2.4 Integration with Java EE services

We assume that you've already tried the "Hello World" example shown earlier in this chapter and that you're familiar with basic Hibernate configuration and how Hibernate can be integrated with a plain Java application. We'll now discuss more advanced native Hibernate configuration options and how a regular Hibernate application can utilize the Java EE services provided by a Java EE application server.

If you created your first JPA project with Hibernate Annotations and Hibernate EntityManager, the following configuration advice isn't really relevant for you—you're already deep inside Java EE land if you're using JPA, and no extra integration steps are required. Hence, you can skip this section if you use Hibernate EntityManager.

Java EE application servers such as JBoss AS, BEA WebLogic, and IBM WebSphere implement the standard (Java EE-specific) managed environment for Java. The three most interesting Java EE services Hibernate can be integrated with are JTA, JNDI, and JMX.

JTA allows Hibernate to participate in transactions on managed resources. Hibernate can look up managed resources (database connections) via JNDI and also bind itself as a service to JNDI. Finally, Hibernate can be deployed via JMX and then be managed as a service by the JMX container and monitored at runtime with standard JMX clients.

Let's look at each service and how you can integrate Hibernate with it.

2.4.1 Integration with JTA

The *Java Transaction API* (JTA) is the standardized service interface for transaction control in Java enterprise applications. It exposes several interfaces, such as the `UserTransaction` API for transaction demarcation and the `TransactionManager` API for participation in the transaction lifecycle. The transaction manager can coordinate a transaction that spans several resources—imagine working in two Hibernate `Sessions` on two databases in a single transaction.

A JTA transaction service is provided by all Java EE application servers. However, many Java EE services are usable stand-alone, and you can deploy a JTA provider along with your application, such as JBoss Transactions or ObjectWeb JOTM. We won't have much to say about this part of your configuration but focus on the integration of Hibernate with a JTA service, which is the same in full application servers or with stand-alone JTA providers.

Look at figure 2.6. You use the Hibernate `Session` interface to access your database(s), and it's Hibernate's responsibility to integrate with the Java EE services of the managed environment.

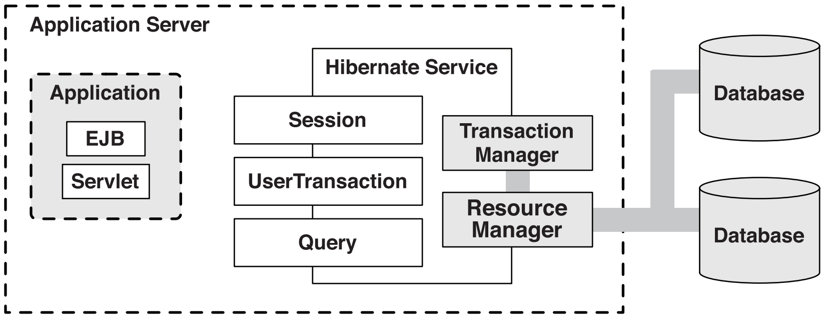


Figure 2.6 Hibernate in an environment with managed resources

In such a managed environment, Hibernate no longer creates and maintains a JDBC connection pool—Hibernate obtains database connections by looking up a `Datasource` object in the JNDI registry. Hence, your Hibernate configuration needs a reference to the JNDI name where managed connections can be obtained.

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>

    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>
    ...

</session-factory>
</hibernate-configuration>
```

With this configuration file, Hibernate looks up database connections in JNDI using the name `java:/MyDatasource`. When you configure your application server and deploy your application, or when you configure your stand-alone JTA provider, this is the name to which you should bind the managed datasource. Note that a dialect setting is still required for Hibernate to produce the correct SQL.

NOTE *Hibernate with Tomcat*—Tomcat isn't a Java EE application server; it's just a servlet container, albeit a servlet container with some features usually found only in application servers. One of these features may be used with Hibernate: the Tomcat connection pool. Tomcat uses the DBCP connection pool internally but exposes it as a JNDI datasource, just like a real application server. To configure the Tomcat datasource, you need to edit server.xml, according to instructions in the Tomcat JNDI/JDBC documentation. Hibernate can be configured to use this datasource by setting `hibernate.connection.datasource`. Keep in mind that Tomcat doesn't ship with a transaction manager, so you still have plain JDBC transaction semantics, which Hibernate can hide with its optional `Transaction` API. Alternatively, you can deploy a JTA-compatible stand-alone transaction manager along with your web application, which you should consider to get the standardized `UserTransaction` API. On the other hand, a regular application server (especially if it's modular like JBoss AS) may be easier to configure than Tomcat plus DBCP plus JTA, and it provides better services.

To fully integrate Hibernate with JTA, you need to tell Hibernate a bit more about your transaction manager. Hibernate has to hook into the transaction lifecycle, for example, to manage its caches. First, you need to tell Hibernate what transaction manager you're using:

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>
```

```
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>

        <property name="hibernate.transaction.manager_lookup_class">
            org.hibernate.transaction.JBossTransactionManagerLookup
        </property>

        <property name="hibernate.transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>

        ...

    </session-factory>
    </hibernate-configuration>
```

You need to pick the appropriate lookup class for your application server, as you did in the preceding code—Hibernate comes bundled with classes for the most popular JTA providers and application servers. Finally, you tell Hibernate that you want to use the JTA transaction interfaces in the application to set transaction boundaries. The `JTATransactionFactory` does several things:

- It enables correct `Session` scoping and propagation for JTA if you decide to use the `SessionFactory.getCurrentSession()` method instead of opening and closing every `Session` manually. We discuss this feature in more detail in chapter 11, section 11.1, "Propagating the Hibernate session."
- It tells Hibernate that you're planning to call the JTA `UserTransaction` interface in your application to start, commit, or roll back system transactions.
- It also switches the Hibernate `Transaction` API to JTA, in case you don't want to work with the standardized `UserTransaction`. If you now begin a transaction with the Hibernate API, it checks whether an ongoing JTA transaction is in progress and, if possible, joins this transaction. If no JTA transaction is in progress, a new transaction is started. If you commit or roll back with the Hibernate API, it either ignores the call (if Hibernate joined an existing transaction) or sets the system transaction to commit or roll back. We don't recommend using the Hibernate `Transaction` API if you deploy in an environment that supports JTA. However, this setting keeps existing code portable between managed and nonmanaged environments, albeit with possibly different transactional behavior.

There are other built-in `TransactionFactory` options, and you can write your own by implementing this interface. The `JDBCTransactionFactory` is the default in a nonmanaged environment, and you have used it throughout this chapter in the simple "Hello World" example with no JTA. The `CMTTransactionFactory` should be enabled if you're working with JTA *and* EJBs, and if you plan to set transaction boundaries declaratively on your managed EJB components—in other words, if you deploy your EJB application on a Java EE application server but don't set transaction boundaries programmatically with the `UserTransaction` interface in application code.

Our recommended configuration options, ordered by preference, are as follows:

- If your application has to run in managed and nonmanaged environments, you should move the responsibility for transaction integration and resource management to the deployer. Call the JTA `UserTransaction` API in your application code, and let the deployer of the application configure the application server or a stand-alone JTA provider accordingly. Enable `JTATransactionFactory` in your Hibernate configuration to integrate with the JTA service, and set the right lookup class.
- Consider setting transaction boundaries declaratively, with EJB components. Your data access code then isn't bound to any transaction API, and the `CMT-TransactionFactory` integrates and handles the Hibernate `Session` for you behind the scenes. This is the easiest solution—of course, the deployer now has the responsibility to provide an environment that supports JTA *and* EJB components.
- Write your code with the Hibernate `Transaction` API and let Hibernate switch between the different deployment environments by setting either `JDBCTransactionFactory` or `JTATransactionFactory`. Be aware that transaction semantics may change, and the start or commit of a transaction may result in a no-op you may not expect. This is always the last choice when portability of transaction demarcation is needed.

FAQ *How can I use several databases with Hibernate?* If you want to work with several databases, you create several configuration files. Each database is assigned its own `SessionFactory`, and you build several `SessionFactory` instances from distinct `Configuration` objects. Each `Session` that is opened, from any `SessionFactory`, looks up a managed datasource in JNDI. It's now the responsibility of the transaction and resource manager to coordinate these resources—Hibernate only executes SQL statements on these database connections. Transaction boundaries are either set programmatically with JTA or handled by the container with EJBs and a declarative assembly.

Hibernate can not only look up managed resources in JNDI, it can also bind itself to JNDI. We'll look at that next.

2.4.2 JNDI-bound SessionFactory

We already touched on a question that every new Hibernate user has to deal with: How should a SessionFactory be stored, and how should it be accessed in application code? Earlier in this chapter, we addressed this problem by writing a HibernateUtil class that held a SessionFactory in a static field and provided the static getSessionFactory() method. However, if you deploy your application in an environment that supports JNDI, Hibernate can bind a SessionFactory to JNDI, and you can look it up there when needed.

NOTE The *Java Naming and Directory Interface* API (JNDI) allows objects to be stored to and retrieved from a hierarchical structure (directory tree). JNDI implements the *Registry* pattern. Infrastructural objects (transaction contexts, datasources, and so on), configuration settings (environment settings, user registries, and so on) and even application objects (EJB references, object factories, and so on) can all be bound to JNDI.

The Hibernate SessionFactory automatically binds itself to JNDI if the hibernate.session_factory_name property is set to the name of the JNDI node. If your runtime environment doesn't provide a default JNDI context (or if the default JNDI implementation doesn't support instances of Referenceable), you need to specify a JNDI initial context using the hibernate.jndi.url and hibernate.jndi.class properties.

Here is an example Hibernate configuration that binds the SessionFactory to the name java:/hibernate/MySessionFactory using Sun's (free) file-system-based JNDI implementation, fscontext.jar:

```
hibernate.connection.datasource = java:/MyDatasource
    hibernate.transaction.factory_class = \
        org.hibernate.transaction.JTATransactionFactory
    hibernate.transaction.manager_lookup_class = \
        org.hibernate.transaction.JBossTransactionManagerLookup
    hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
    hibernate.session_factory_name = java:/hibernate/MySessionFactory
    hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
    hibernate.jndi.url = file:/auction/jndi
```

You can, of course, also use the XML-based configuration for this. This example isn't realistic, because most application servers that provide a connection pool through JNDI also have a JNDI implementation with a writable default context.

JBoss AS certainly has, so you can skip the last two properties and just specify a name for the SessionFactory.

NOTE *JNDI with Tomcat*—Tomcat comes bundled with a read-only JNDI context, which isn't writable from application-level code after the startup of the servlet container. Hibernate can't bind to this context: You have to either use a full context implementation (like the Sun FS context) or disable JNDI binding of the SessionFactory by omitting the session_factory_name property in the configuration.

The SessionFactory is bound to JNDI when you build it, which means when Configuration.buildSessionFactory() is called. To keep your application code portable, you may want to implement this build and the lookup in HibernateUtil, and continue using that helper class in your data access code, as shown in listing 2.18.

Listing 2.18 HibernateUtil for JNDI lookup of SessionFactory

```
public class HibernateUtil {

    private static Context jndiContext;

    static {
        try {
            // Build it and bind it to JNDI
            new Configuration().buildSessionFactory();

            // Get a handle to the registry (reads jndi.properties)
            jndiContext = new InitialContext();

        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory(String sfName) {
        SessionFactory sf;
        try {
            sf = (SessionFactory) jndiContext.lookup(sfName);
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
        return sf;
    }
}
```

Alternatively, you can look up the `SessionFactory` directly in application code with a JNDI call. However, you still need at least the `new Configuration().build-SessionFactory()` line of startup code somewhere in your application. One way to remove this last line of Hibernate startup code, and to completely eliminate the `HibernateUtil` class, is to deploy Hibernate as a JMX service (or by using JPA and Java EE).

2.4.3 JMX service deployment

The Java world is full of specifications, standards, and implementations of these. A relatively new, but important, standard is in its first version: the *Java Management Extensions* (JMX). JMX is about the management of systems components or, better, of system services.

Where does Hibernate fit into this new picture? Hibernate, when deployed in an application server, makes use of other services, like managed transactions and pooled datasources. Also, with Hibernate JMX integration, Hibernate can be a managed JMX service, depended on and used by others.

The JMX specification defines the following components:

- The *JMX MBean*—A reusable component (usually infrastructural) that exposes an interface for *management* (administration)
- The *JMX container*—Mediates generic access (local or remote) to the MBean
- The *JMX client*—May be used to administer any MBean via the JMX container

An application server with support for JMX (such as JBoss AS) acts as a JMX container and allows an MBean to be configured and initialized as part of the application server startup process. Your Hibernate service may be packaged and deployed as a JMX MBean; the bundled interface for this is `org.hibernate.jmx.HibernateService`. You can start, stop, and monitor the Hibernate core through this interface with any standard JMX client. A second MBean interface that can be deployed optionally is `org.hibernate.jmx.StatisticsService`, which lets you enable and monitor Hibernate's runtime behavior with a JMX client.

How JMX services and MBeans are deployed is vendor-specific. For example, on JBoss Application Server, you only have to add a jboss-service.xml file to your application's EAR to deploy Hibernate as a managed JMX service.

Instead of explaining every option here, see the reference documentation for JBoss Application Server. It contains a section that shows Hibernate integration and deployment step by step (http://docs.jboss.org/jbossas). Configuration and deployment on other application servers that support JMX should be similar, and you can adapt and port the JBoss configuration files.

2.5 Summary

In this chapter, you have completed a first Hibernate project. We looked at how Hibernate XML mapping files are written and what APIs you can call in Hibernate to interact with the database.

We then introduced Java Persistence and EJB 3.0 and explained how it can simplify even the most basic Hibernate application with automatic metadata scanning, standardized configuration and packaging, and dependency injection in managed EJB components.

If you have to get started with a legacy database, you can use the Hibernate toolset to reverse engineer XML mapping files from an existing schema. Or, if you work with JDK 5.0 and/or EJB 3.0, you can generate Java application code directly from an SQL database.

Finally, we looked at more advanced Hibernate integration and configuration options in a Java EE environment—integration that is already done for you if you rely on JPA or EJB 3.0.

A high-level overview and comparison between Hibernate functionality and Java Persistence is shown in table 2.1. (You can find a similar comparison table at the end of each chapter.)

Table 2.1 Hibernate and JPA comparison

Hibernate Core	Java Persistence and EJB 3.0
Integrates with everything, everywhere. Flexible, but sometimes configuration is complex.	Works in Java EE and Java SE. Simple and standardized configuration; no extra integration or special configuration is necessary in Java EE environments.
Configuration requires a list of XML mapping files or annotated classes.	JPA provider scans for XML mapping files and annotated classes automatically.
Proprietary but powerful. Continually improved native programming interfaces and query language.	Standardized and stable interfaces, with a sufficient subset of Hibernate functionality. Easy fallback to Hibernate APIs is possible.

In the next chapter, we introduce a more complex example application that we'll work with throughout the rest of the book. You'll see how to design and implement a domain model, and which mapping metadata options are the best choices in a larger project.