The ultimate Hibernate reference

SAMPLE
CHAPTER

HIBERNATE IN ACTION

Christian Bauer
Gavin King

MANNING

Hibernate in Action

by Christian Bauer

and

Gavin King

Chapter 2

contents

2 Introducing and integrating Hibernate

This chapter covers

- Hibernate in action with “Hello World”
- The Hibernate core programming interfaces
- Integration with managed and non-managed environments
- Advanced configuration options

It's good to understand the need for object/relational mapping in Java applications, but you're probably eager to see Hibernate in action. We'll start by showing you a simple example that demonstrates some of its power.

As you're probably aware, it's traditional for a programming book to start with a "Hello World" example. In this chapter, we follow that tradition by introducing Hibernate with a relatively simple "Hello World" program. However, simply printing a message to a console window won't be enough to really demonstrate Hibernate. Instead, our program will store newly created objects in the database, update them, and perform queries to retrieve them from the database.

This chapter will form the basis for the subsequent chapters. In addition to the canonical "Hello World" example, we introduce the core Hibernate APIs and explain how to configure Hibernate in various runtime environments, such as J2EE application servers and stand-alone applications.

2.1 "Hello World" with Hibernate

Hibernate applications define *persistent classes* that are "mapped" to database tables. Our "Hello World" example consists of one class and one mapping file. Let's see what a simple persistent class looks like, how the mapping is specified, and some of the things we can do with instances of the persistent class using Hibernate.

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

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

```
package hello;
public class Message {
    private Long id;
    private String text;
    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;
```

Identifier attribute

Message text

Reference to another Message

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

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

Our `Message` class has three attributes: the identifier attribute, the text of the message, and a reference to another `Message`. 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. We've chosen `Long` for the type of our 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 JavaBean-style property accessor methods. The class also has a constructor with no parameters. The persistent classes we use in our examples will almost always look something like this.

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

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

This code fragment does exactly what we've come to expect from "Hello World" applications: It prints `"Hello World"` to the console. It might 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, such as EJB entity beans. Our persistent class can be used in any execution context at all—no special container is needed. Of course, you came here to see Hibernate itself, so let's save a new `Message` to the database:

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
```

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

This code calls the Hibernate `Session` and `Transaction` interfaces. (We'll get to that `getSessionFactory()` call soon.) It 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. We didn't set the `id` property of `message` anywhere, so we would expect it to be `null`, right? Actually, the `id` property is special: It's an *identifier property*—it holds a generated unique value. (We'll discuss how the value is generated later.) The value is assigned to the `Message` instance by Hibernate when `save()` is called.

For this example, we assume that the `MESSAGES` table already exists. In chapter 9, we'll show you how to use Hibernate to automatically create the tables your application needs, using just the information in the mapping files. (There's some more SQL you won't need to write by hand!) Of course, we want our "Hello World" program to print the message to the console. Now that we have a message in the database, we're ready to demonstrate this. The next example retrieves all messages from the database, in alphabetical order, and prints them:

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages =
        newSession.find("from Message as m order by m.text asc");
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
    Message message = (Message) iter.next();
    System.out.println( message.getText() );
}
newTransaction.commit();
newSession.close();
```

The literal string `"from Message as m order by m.text asc"` is a Hibernate query, expressed in Hibernate's own object-oriented Hibernate Query Language (HQL). This query is internally translated into the following SQL when `find()` is called:

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

The code fragment prints

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

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

To allow this magic to occur, Hibernate needs more information about how the `Message` class should be made persistent. This information is usually provided in an *XML mapping document*. The mapping document defines, among other things, how properties of the `Message` class map to columns of the `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-2.0.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"/>
    </class>
</hibernate-mapping>
```

Note that Hibernate 2.0 and Hibernate 2.1 have the same DTD!

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 column named `NEXT_MESSAGE_ID`. (Don't worry about the other details for now.)

As you can see, the XML document isn't difficult to understand. You can easily write and maintain it by hand. In chapter 3, we discuss a way of generating the

XML file from comments embedded in the source code. Whichever method you choose, Hibernate has enough information to completely generate all the SQL statements that would be needed to insert, update, delete, and retrieve instances of the `Message` class. You no longer need to write these SQL statements by hand.

NOTE Many Java developers have complained of the "metadata hell" that accompanies J2EE development. Some have suggested a movement away from XML metadata, back to plain Java code. Although we applaud this suggestion for some problems, ORM represents a case where text-based metadata really is necessary. Hibernate has sensible defaults that minimize typing and a mature document type definition that can be used for auto-completion or validation in editors. You can even automatically generate metadata with various tools.

Now, let's change our first message and, while we're at it, create a new message associated with the first, as shown in listing 2.3.

Listing 2.3 Updating a message

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

// 1 is the generated id of the first message
Message message =
        (Message) session.load( Message.class, new Long(1) );
message.setText("Greetings Earthling");
Message nextMessage = new Message("Take me to your leader (please)");
message.setNextMessage( nextMessage );
tx.commit();
session.close();
```

This code calls three SQL statements inside the same 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. We've taken advantage of a Hibernate feature called *automatic dirty checking:* This feature

saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction. Similarly, you can see that the new message was made persistent when a reference was created from the first message. This feature is called *cascading save:* It saves us 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 we 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 we run "Hello World" again, it prints

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

This is as far as we'll take the "Hello World" application. Now that we finally have some code under our belt, we'll take a step back and present an overview of Hibernate's main APIs.

2.2 Understanding the architecture

The programming interfaces are the first thing you have to learn about Hibernate in order to use it in the persistence layer of your application. A major objective of API design is to keep the interfaces between software components as narrow as possible. In practice, however, ORM APIs aren't especially small. Don't worry, though; you don't have to understand all the Hibernate interfaces at once. Figure 2.1 illustrates the roles of the most important Hibernate interfaces in the business and persistence layers. We show the business layer above the persistence layer, since the business layer acts as a client of the persistence layer in a traditionally layered application. Note that some simple applications might not cleanly separate business logic from persistence logic; that's okay—it merely simplifies the diagram.

The Hibernate interfaces shown in figure 2.1 may be approximately classified as follows:

- Interfaces called by applications to perform basic CRUD and querying operations. These interfaces are the main point of dependency of application business/control logic on Hibernate. They include `Session`, `Transaction`, and `Query`.

- Interfaces called by application infrastructure code to configure Hibernate, most importantly the `Configuration` class.
- *Callback* interfaces that allow the application to react to events occurring inside Hibernate, such as `Interceptor`, `Lifecycle`, and `Validatable`.
- Interfaces that allow extension of Hibernate's powerful mapping functionality, such as `UserType`, `CompositeUserType`, and `IdentifierGenerator`. These interfaces are implemented by application infrastructure code (if necessary).

Hibernate makes use of existing Java APIs, including JDBC), Java Transaction API (JTA, and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

In this section, we don't cover the detailed semantics of Hibernate API methods, just the role of each of the primary interfaces. You can find most of these interfaces in the package `net.sf.hibernate`. Let's take a brief look at each interface in turn.

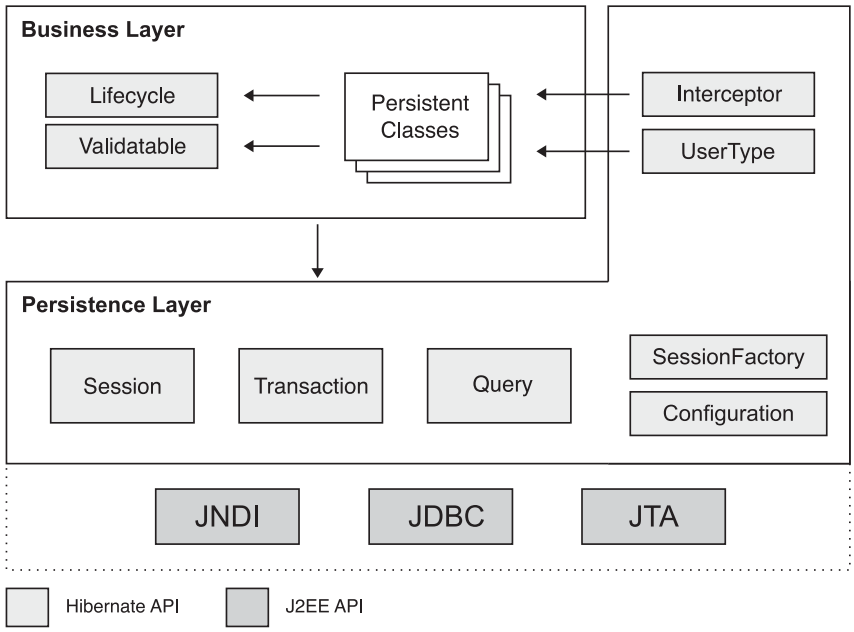

Figure 2.1 High-level overview of the HIbernate API in a layered architecture

2.2.1 The core interfaces

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

Session interface

The `Session` interface is the primary interface used by Hibernate applications. An instance of `Session` is lightweight and is inexpensive to create and destroy. This is important because your application will need to create and destroy sessions all the time, perhaps on every request. Hibernate sessions are *not* threadsafe and should by design be used by only one thread at a time.

The Hibernate notion of a *session* is something between *connection* and *transaction*. It may be easier to think of a session as a cache or collection of loaded objects relating to a single unit of work. Hibernate can detect changes to the objects in this unit of work. We sometimes call the `Session` a *persistence manager* because it's also the interface for persistence-related operations such as storing and retrieving objects. Note that a Hibernate session has nothing to do with the web-tier `HttpSession`. When we use the word *session* in this book, we mean the Hibernate session. We sometimes use *user session* to refer to the `HttpSession` object.

We describe the `Session` interface in detail in chapter 4, section 4.2, "The persistence manager."

SessionFactory interface

The application obtains `Session` instances from a `SessionFactory`. Compared to the `Session` interface, this object is much less exciting.

The `SessionFactory` is certainly not lightweight! It's intended to be shared among many application threads. There is typically a single `SessionFactory` for the whole application—created during application initialization, for example. However, if your application accesses multiple databases using Hibernate, you'll need a `SessionFactory` for each database.

The `SessionFactory` caches generated SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work (only if class and collection mappings specify that this *second-level cache* is desirable).

Configuration interface

The `Configuration` object is used to configure and bootstrap Hibernate. The application uses a `Configuration` instance to specify the location of mapping documents and Hibernate-specific properties and then create the `SessionFactory`.

Even though the `Configuration` interface plays a relatively small part in the total scope of a Hibernate application, it's the first object you'll meet when you begin using Hibernate. Section 2.3 covers the problem of configuring Hibernate in some detail.

Transaction interface

The `Transaction` interface is an optional API. Hibernate applications may choose not to use this interface, instead managing transactions in their own infrastructure code. A `Transaction` abstracts application code from the underlying transaction implementation—which might be a JDBC transaction, a JTA `UserTransaction`, or even a Common Object Request Broker Architecture (CORBA) transaction—allowing the application to control transaction boundaries via a consistent API. This helps to keep Hibernate applications portable between different kinds of execution environments and containers.

We use the Hibernate `Transaction` API throughout this book. Transactions and the `Transaction` interface are explained in chapter 5.

Query and Criteria interfaces

The `Query` interface allows you to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database. A `Query` instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

The `Criteria` interface is very similar; it allows you to create and execute object-oriented criteria queries.

To help make application code less verbose, Hibernate provides some shortcut methods on the `Session` interface that let you invoke a query in one line of code. We won't use these shortcuts in the book; instead, we'll always use the `Query` interface.

A `Query` instance is lightweight and can't be used outside the `Session` that created it. We describe the features of the `Query` interface in chapter 7.

2.2.2 Callback interfaces

Callback interfaces allow the application to receive a notification when something interesting happens to an object—for example, when an object is loaded, saved, or deleted. Hibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality, such as creating audit records.

The `Lifecycle` and `Validatable` interfaces allow a persistent object to react to events relating to its own *persistence lifecycle*. The persistence lifecycle is encompassed by an object's CRUD operations. The Hibernate team was heavily influenced by other ORM solutions that have similar callback interfaces. Later, they realized that having the persistent classes implement Hibernate-specific interfaces probably isn't a good idea, because doing so pollutes our persistent classes with nonportable code. Since these approaches are no longer favored, we don't discuss them in this book.

The `Interceptor` interface was introduced to allow the application to process callbacks without forcing the persistent classes to implement Hibernate-specific APIs. Implementations of the `Interceptor` interface are passed to the persistent instances as parameters. We'll discuss an example in chapter 8.

2.2.3 Types

A fundamental and very powerful element of the architecture is Hibernate's notion of a `Type`. A Hibernate `Type` object maps a Java type to a database column type (actually, the type may span multiple columns). All persistent properties of persistent classes, including associations, have a corresponding Hibernate type. This design makes Hibernate extremely flexible and extensible.

There is a rich range of built-in types, covering all Java primitives and many JDK classes, including types for `java.util.Currency`, `java.util.Calendar`, `byte[]`, and `java.io.Serializable`.

Even better, Hibernate supports user-defined *custom types*. The interfaces `UserType` and `CompositeUserType` are provided to allow you to add your own types. You can use this feature to allow commonly used application classes such as `Address`, `Name`, or `MonetaryAmount` to be handled conveniently and elegantly. Custom types are considered a central feature of Hibernate, and you're encouraged to put them to new and creative uses!

We explain Hibernate types and user-defined types in chapter 6, section 6.1, "Understanding the Hibernate type system."

2.2.4 Extension interfaces

Much of the functionality that Hibernate provides is configurable, allowing you to choose between certain built-in strategies. When the built-in strategies are insufficient, Hibernate will usually let you plug in your own custom implementation by implementing an interface. Extension points include:

- Primary key generation (`IdentifierGenerator` interface)
- SQL dialect support (`Dialect` abstract class)
- Caching strategies (`Cache` and `CacheProvider` interfaces)
- JDBC connection management (`ConnectionProvider` interface)
- Transaction management (`TransactionFactory`, `Transaction`, and `TransactionManagerLookup` interfaces)
- ORM strategies (`ClassPersister` interface hierarchy)
- Property access strategies (`PropertyAccessor` interface)
- Proxy creation (`ProxyFactory` interface)

Hibernate ships with at least one implementation of each of the listed interfaces, so you don't usually need to start from scratch if you wish to extend the built-in functionality. The source code is available for you to use as an example for your own implementation.

By now you can see that before we can start writing any code that uses Hibernate, we must answer this question: How do we get a `Session` to work with?

2.3 Basic configuration

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

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

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

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

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

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

2.3.1 Creating a SessionFactory

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

The following code starts Hibernate:

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

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

METHOD CHAINING

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Configuration in non-managed environments

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

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

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

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

Using a connection pool

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

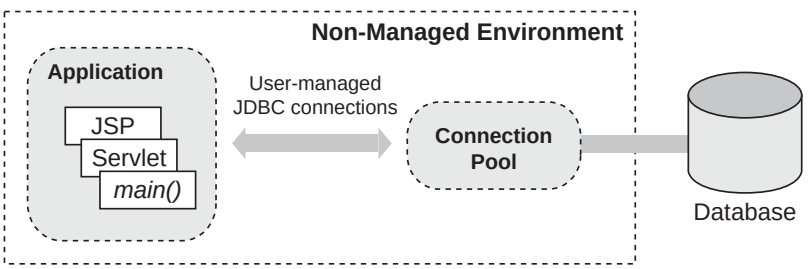

Figure 2.2 JDBC `connection` pooling in a non-managed environment

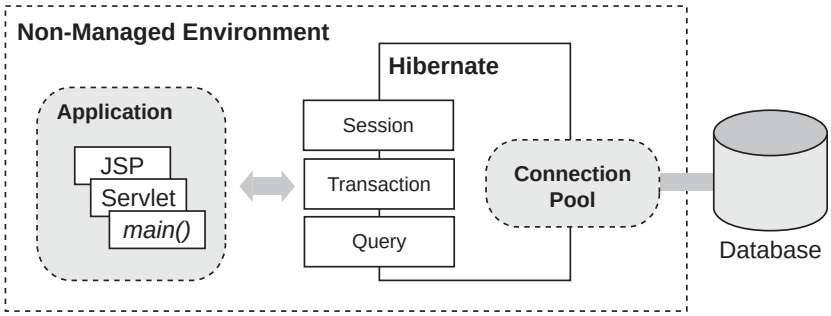

Figure 2.3 Hibernate with a connection pool in a non-managed environment

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

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

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

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

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

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

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

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

Starting Hibernate

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

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

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

3 Choose a JDBC connection pool supported by Hibernate and configure it with a properties file. Don't forget to specify the SQL dialect.

4 Let the `Configuration` know about these properties by placing them in a `hibernate.properties` file in the classpath.

5 Create an instance of `Configuration` in your application and load the XML mapping files using either `addResource()` or `addClass()`. Build a `SessionFactory` from the `Configuration` by calling `buildSessionFactory()`.

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

2.3.3 Configuration in managed environments

A managed environment handles certain cross-cutting concerns, such as application security (authorization and authentication), connection pooling, and transaction management. J2EE application servers are typical managed environments. Although application servers are generally designed to support EJBs, you can still take advantage of the other managed services provided, even if you don't use EJB entity beans.

Hibernate is often used with session or message-driven EJBs, as shown in figure 2.4. EJBs call the same Hibernate APIs as servlets, JSPs, or stand-alone applications: `Session`, `Transaction`, and `Query`. The Hibernate-related code is fully portable between non-managed and managed environments. Hibernate handles the different connection and transaction strategies transparently.

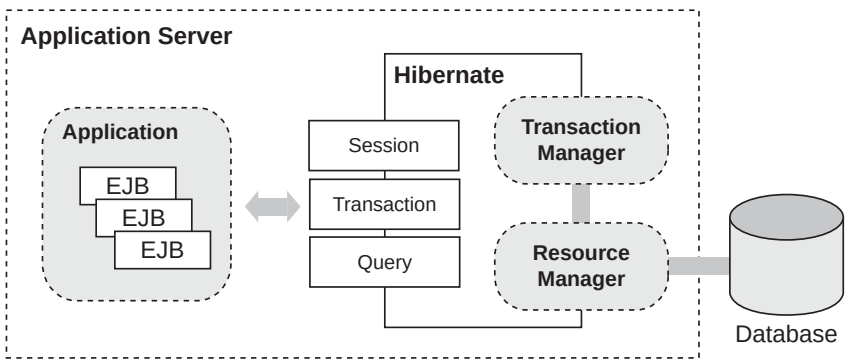

Figure 2.4 Hibernate in a managed environment with an application server

An application server exposes a connection pool as a JNDI-bound *datasource*, an instance of `javax.jdbc.Datasource`. You need to tell Hibernate where to find the datasource in JNDI, by supplying a fully qualified JNDI name. An example Hibernate configuration file for this scenario is shown in listing 2.5.

Listing 2.5 Sample `hibernate.properties` for a container-provided datasource

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

This file first gives the JNDI name of the datasource. The datasource must be configured in the J2EE enterprise application deployment descriptor; this is a vendor-specific setting. Next, you enable Hibernate integration with JTA. Now Hibernate needs to locate the application server's `TransactionManager` in order to integrate fully with the container transactions. No standard approach is defined by the J2EE specification, but Hibernate includes support for all popular application servers. Finally, of course, the Hibernate SQL dialect is required.

Now that you've configured everything correctly, using Hibernate in a managed environment isn't much different than using it in a non-managed environment: Just create a `Configuration` with mappings and build a `SessionFactory`. However, some of the transaction environment–related settings deserve some extra consideration.

Java already has a standard transaction API, JTA, which is used to control transactions in a managed environment with J2EE. This is called *container-managed transactions* (CMT). If a JTA transaction manager is present, JDBC connections are enlisted with this manager and under its full control. This isn't the case in a non-managed environment, where an application (or the pool) manages the JDBC connections and JDBC transactions directly.

Therefore, managed and non-managed environments can use different transaction methods. Since Hibernate needs to be portable across these environments, it defines an API for controlling transactions. The Hibernate `Transaction` interface abstracts the underlying JTA or JDBC transaction (or, potentially, even a CORBA transaction). This underlying transaction strategy is set with the property `hibernate.connection.factory_class`, and it can take one of the following two values:

- `net.sf.hibernate.transaction.JDBCTransactionFactory` delegates to direct JDBC transactions. This strategy should be used with a connection pool in a non-managed environment and is the default if no strategy is specified.
- `net.sf.hibernate.transaction.JTATransactionFactory` delegates to JTA. This is the correct strategy for CMT, where connections are enlisted with JTA. Note that if a JTA transaction is already in progress when `beginTransaction()` is called, subsequent work takes place in the context of that transaction (otherwise a new JTA transaction is started).

For a more detailed introduction to Hibernate's `Transaction` API and the effects on your specific application scenario, see chapter 5, section 5.1, "Transactions." Just remember the two steps that are necessary if you work with a J2EE application server: Set the factory class for the Hibernate `Transaction` API to JTA as described earlier, and declare the transaction manager lookup specific to your application server. The lookup strategy is required only if you use the second-level caching system in Hibernate, but it doesn't hurt to set it even without using the cache.

HIBERNATE WITH TOMCAT

Tomcat isn't a full 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'll need to edit `server.xml` according to instructions in the Tomcat JNDI/JDBC documentation. You can configure Hibernate to use this datasource by setting `hibernate.connection.datasource`. Keep in mind that Tomcat doesn't ship with a transaction manager, so this situation is still more like a non-managed environment as described earlier.

You should now have a running Hibernate system, whether you use a simple servlet container or an application server. Create and compile a persistent class (the initial `Message`, for example), copy Hibernate and its required libraries to the classpath together with a `hibernate.properties` file, and build a `SessionFactory`.

The next section covers advanced Hibernate configuration options. Some of them are recommended, such as logging executed SQL statements for debugging or using the convenient XML configuration file instead of plain properties. However, you may safely skip this section and come back later once you have read more about persistent classes in chapter 3.

2.4 Advanced configuration settings

When you finally have a Hibernate application running, it's well worth getting to know all the Hibernate configuration parameters. These parameters let you optimize the runtime behavior of Hibernate, especially by tuning the JDBC interaction (for example, using JDBC batch updates).

We won't bore you with these details now; the best source of information about configuration options is the Hibernate reference documentation. In the previous section, we showed you the options you'll need to get started.

However, there is one parameter that we *must* emphasize at this point. You'll need it continually whenever you develop software with Hibernate. Setting the property `hibernate.show_sql` to the value `true` enables logging of all generated SQL to the console. You'll use it for troubleshooting, performance tuning, and just to see what's going on. It pays to be aware of what your ORM layer is doing—that's why ORM doesn't hide SQL from developers.

So far, we've assumed that you specify configuration parameters using a `hibernate.properties` file or an instance of `java.util.Properties` programmatically. There is a third option you'll probably like: using an XML configuration file.

2.4.1 Using XML-based configuration

You can use an XML configuration file (as demonstrated in listing 2.6) to fully configure a `SessionFactory`. Unlike `hibernate.properties`, which contains only configuration parameters, the `hibernate.cfg.xml` file may also specify the location of mapping documents. Many users prefer to centralize the configuration of Hibernate in this way instead of adding parameters to the `Configuration` in application code.

Listing 2.6 Sample `hibernate.cfg.xml` configuration file

```
?xml version='1.0'encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory name="java:/hibernate/HibernateFactory">
    <property name="show_sql">true</property>
    <property name="connection.datasource">
        java:/comp/env/jdbc/AuctionDB
    </property>
    <property name="dialect">
        net.sf.hibernate.dialect.PostgreSQLDialect
    </property>
```

Document type declaration ❶

❷ Name attribute

❸ Property specifications

```
    <property name="transaction.manager_lookup_class">
        net.sf.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <mapping resource="auction/Item.hbm.xml"/>
    <mapping resource="auction/Category.hbm.xml"/>
    <mapping resource="auction/Bid.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

❹ **Mapping document specifications**

❶ The *document type* declaration is used by the XML parser to validate this document against the Hibernate configuration DTD.

❷ The optional `name` attribute is equivalent to the property `hibernate.session_factory_name` and used for JNDI binding of the `SessionFactory`, discussed in the next section.

❸ Hibernate properties may be specified without the `hibernate` prefix. Property names and values are otherwise identical to programmatic configuration properties.

❹ Mapping documents may be specified as application resources or even as hard-coded filenames. The files used here are from our online auction application, which we'll introduce in chapter 3.

Now you can initialize Hibernate using

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

Wait—how did Hibernate know where the configuration file was located?

When `configure()` was called, Hibernate searched for a file named `hibernate.cfg.xml` in the classpath. If you wish to use a different filename or have Hibernate look in a subdirectory, you must pass a path to the `configure()` method:

```
SessionFactory sessions = new Configuration()
        .configure("/hibernate-config/auction.cfg.xml")
          .buildSessionFactory();
```

Using an XML configuration file is certainly more comfortable than a properties file or even programmatic property configuration. The fact that you can have the class mapping files externalized from the application's source (even if it would be only in a startup helper class) is a major benefit of this approach. You can, for example, use different sets of mapping files (and different configuration options), depending on your database and environment (development or production), and switch them programatically.

If you have both `hibernate.properties` and `hibernate.cfg.xml` in the classpath, the settings of the XML configuration file will override the settings used in the properties. This is useful if you keep some base settings in properties and override them for each deployment with an XML configuration file.

You may have noticed that the `SessionFactory` was also given a `name` in the XML configuration file. Hibernate uses this name to automatically bind the `SessionFactory` to JNDI after creation.

2.4.2 JNDI-bound 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`s should be created using this single `SessionFactory`. A frequently asked question is where this factory must be placed and how it can be accessed without much hassle.

In a J2EE environment, a `SessionFactory` bound to JNDI is easily shared between different threads and between various Hibernate-aware components. Or course, JNDI isn't the only way that application components might obtain a `SessionFactory`. There are many possible implementations of this Registry pattern, including use of the `ServletContext` or a `static final` variable in a singleton. A particularly elegant approach is to use an application scope IoC (Inversion of Control) framework component. However, JNDI is a popular approach (and is exposed as a JMX service, as you'll see later). We discuss some of the alternatives in chapter 8, section 8.1, "Designing layered applications."

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

The `SessionFactory` will automatically bind itself to JNDI if the property `hibernate.session_factory_name` is set to the name of the directory 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 properties `hibernate.jndi.url` and `hibernate.jndi.class`.

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

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

Of course, you can also use the XML-based configuration for this task. This example also isn't realistic, since most application servers that provide a connection pool through JNDI also have a JNDI implementation with a writable default context. JBoss certainly has, so you can skip the last two properties and just specify a name for the `SessionFactory`. All you have to do now is call `Configuration.configure().buildSessionFactory()` once to initialize the binding.

NOTE 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.

Let's look at some other very important configuration settings that log Hibernate operations.

2.4.3 Logging

Hibernate (and many other ORM implementations) executes SQL statements *asynchronously*. An `INSERT` statement isn't usually executed when the application calls `Session.save()`; an `UPDATE` isn't immediately issued when the application calls `Item.addBid()`. Instead, the SQL statements are usually issued at the end of a transaction. This behavior is called *write-behind*, as we mentioned earlier.

This fact is evidence 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. Certainly the Hibernate application can't detect this asynchronicity (at least, not without resorting to direct JDBC calls). However, when you find yourself troubleshooting a difficult problem, you need to be able to see *exactly* what's going on inside Hibernate. Since Hibernate is open source, you can

easily step into the Hibernate code. Occasionally, doing so helps a great deal! But, especially in the face of asynchronous behavior, debugging Hibernate can quickly get you lost. You can use logging to get a view of Hibernate's internals.

We've already mentioned the `hibernate.show_sql` configuration parameter, which is usually the first port of call when troubleshooting. Sometimes the SQL alone is insufficient; in that case, you must dig a little deeper.

Hibernate logs all interesting events using Apache `commons-logging`, a thin abstraction layer that directs output to either Apache log4j (if you put `log4j.jar` in your classpath) or JDK1.4 logging (if you're running under JDK1.4 or above and log4j isn't present). We recommend log4j, since it's more mature, more popular, and under more active development.

To see any output from log4j, you'll need a file named `log4j.properties` in your classpath (right next to `hibernate.properties` or `hibernate.cfg.xml`). This example directs all log messages to the console:

```
### 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=warn, stdout
### Hibernate logging options ###
log4j.logger.net.sf.hibernate=info
### log JDBC bind parameters ###
log4j.logger.net.sf.hibernate.type=info
### log PreparedStatement cache activity ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

With this configuration, you won't see many log messages at runtime. Replacing info with `debug` for the `log4j.logger.net.sf.hibernate` category will reveal the inner workings of Hibernate. Make sure you don't do this in a production environment—writing the log will be much slower than the actual database access.

Finally, you have the `hibernate.properties`, `hibernate.cfg.xml`, and `log4j.properties` configuration files.

There is another way to configure Hibernate, if your application server supports the Java Management Extensions.

2.4.4 Java Management Extensions (JMX)

The Java world is full of specifications, standards, and, of course, 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 database transactions. But why not make Hibernate a managed service itself, which others can depend on and use? This is possible with the Hibernate JMX integration, making Hibernate a managed JMX component.

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 (usually generic) JMX client*—May be used to administer any MBean via the JMX container

An application server with support for JMX (such as JBoss) acts as a JMX container and allows an MBean to be configured and initialized as part of the application server startup process. It's possible to monitor and administer the MBean using the application server's administration console (which acts as the JMX client).

An MBean may be packaged as a JMX service, which is not only portable between application servers with JMX support but also deployable to a running system (a *hot deploy*).

Hibernate may be packaged and administered as a JMX MBean. The Hibernate JMX service allows Hibernate to be initialized at application server startup and controlled (configured) via a JMX client. However, JMX components aren't automatically integrated with container-managed transactions. So, the configuration options in listing 2.7 (a JBoss service deployment descriptor) look similar to the usual Hibernate settings in a managed environment.

Listing 2.7 Hibernate `jboss-service.xml` JMX deployment descriptor

```
<server>
<mbean
     code="net.sf.hibernate.jmx.HibernateService"
     name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
  <depends>jboss.jca:service=RARDeployer</depends>
  <depends>jboss.jca:service=LocalTxCM,name=DataSource</depends>
  <attribute name="MapResources">
    auction/Item.hbm.xml, auction/Bid.hbm.xml
  </attribute>
```

```
  <attribute name="JndiName">
     java:/hibernate/HibernateFactory
  </attribute>
  <attribute name="Datasource">
     java:/comp/env/jdbc/AuctionDB
  </attribute>
  <attribute name="Dialect">
     net.sf.hibernate.dialect.PostgreSQLDialect
  </attribute>
  <attribute name="TransactionStrategy">
     net.sf.hibernate.transaction.JTATransactionFactory
  </attribute>
  <attribute name="TransactionManagerLookupStrategy">
     net.sf.hibernate.transaction.JBossTransactionManagerLookup
  </attribute>
  <attribute name="UserTransactionName">
     java:/UserTransaction
  </attribute>
</mbean>
</server>
```

The `HibernateService` depends on two other JMX services: `service=RARDeployer` and `service=LocalTxCM,name=DataSource`, both in the `jboss.jca` service domain name.

The Hibernate MBean may be found in the package `net.sf.hibernate.jmx`. Unfortunately, lifecycle management methods like starting and stopping the JMX service aren't part of the JMX 1.0 specification. The methods `start()` and `stop()` of the `HibernateService` are therefore specific to the JBoss application server.

NOTE If you're interested in the advanced usage of JMX, JBoss is a good open source starting point: All services (even the EJB container) in JBoss are implemented as MBeans and can be managed via a supplied console interface.

We recommend that you try to configure Hibernate programmatically (using the `Configuration` object) before you try to run Hibernate as a JMX service. However, some features (like hot-redeployment of Hibernate applications) may be possible only with JMX, once they become available in Hibernate. Right now, the biggest advantage of Hibernate with JMX is the automatic startup; it means you no longer have to create a `Configuration` and build a `SessionFactory` in your application code, but can simply access the `SessionFactory` through JNDI once the `HibernateService` has been deployed and started.

2.5 Summary

In this chapter, we took a high-level look at Hibernate and its architecture after running a simple "Hello World" example. You also saw how to configure Hibernate in various environments and with various techniques, even including JMX.

The `Configuration` and `SessionFactory` interfaces are the entry points to Hibernate for applications running in both managed and non-managed environments. Hibernate provides additional APIs, such as the `Transaction` interface, to bridge the differences between environments and allow you to keep your persistence code portable.

Hibernate can be integrated into almost every Java environment, be it a servlet, an applet, or a fully managed three-tiered client/server application. The most important elements of a Hibernate configuration are the database resources (connection configuration), the transaction strategies, and, of course, the XML-based mapping metadata.

Hibernate's configuration interfaces have been designed to cover as many usage scenarios as possible while still being easy to understand. Usually, a single file named `hibernate.cfg.xml` and one line of code are enough to get Hibernate up and running.

None of this is much use without some persistent classes and their XML mapping documents. The next chapter is dedicated to writing and mapping persistent classes. You'll soon be able to store and retrieve persistent objects in a real application with a nontrivial object/relational mapping.

JAVA

HIBERNATE IN ACTION

Christian Bauer and Gavin King

"The Bible of Hibernate"

—Ara Abrahamian, XDoclet Lead Developer

Hibernate practically exploded on the Java scene. Why is this open-source tool so popular? Because it automates a tedious task: persisting your Java objects to a relational database. The inevitable mismatch between your object-oriented code and the relational database requires you to write code that maps one to the other. This code is often complex, tedious and costly to develop. Hibernate does the mapping for you.

Not only that, Hibernate makes it easy. Positioned as a layer between your application and your database, Hibernate takes care of loading and saving of objects. Hibernate applications are cheaper, more portable, and more resilient to change. And they perform better than anything you are likely to develop yourself.

Hibernate in Action carefully explains the concepts you need, then gets you going. It builds on a single example to show you how to use Hibernate in practice, how to deal with concurrency and transactions, how to efficiently retrieve objects and use caching.

The authors created Hibernate and they field questions from the Hibernate community every day—they know how to make Hibernate sing. Knowledge and insight seep out of every pore of this book.

What's Inside

- ORM concepts
- Getting started
- Many real-world tasks
- The Hibernate application development process

Ask the Authors

Ebook edition

www.manning.com/bauer

A member of the core Hibernate developer team, **Christian Bauer** maintains the Hibernate documentation and website. He is a senior software engineer in Frankfurt, Germany. **Gavin King** is the Hibernate founder and principal developer. He is a J2EE consultant based in Melbourne, Australia.

$44.95 US/$62.95 Canada

ISBN 1-932394-15-X