Hibernate Basics

As a persistence service, Hibernate must work with multiple databases and within various application environments. Supporting these variations requires Hibernate to be highly configurable to adapt to different environments. After all, running a standalone application can be quite different from running a web application. Differences in obtaining database connections, for instance, can be significant. Hibernate is typically configured in two steps.

First, you configure the Hibernate service. This includes database connection parameters, caching, and the collection of persistent classes man–aged by Hibernate. Second, you must provide Hibernate with information about the classes to be persisted. Persistent class configuration allows you to bridge gaps between the class and databases.

Although it's commonly used within J2EE application servers, such as WebSphere and JBoss, Hibernate can also be used in standalone applications. Requirements vary for different environments, and Hibernate can be configured to adapt to them. Hibernate works with various support services, such as connection pools, caching services, and transaction managers. It also lets you maintain additional support services by implementing simple interfaces.

Individual persistent classes are also highly configurable. Each class may have a different method to generate identifier values, and it's possible to persist complex object hierarchies. You can also customize specific object properties mapping to a SQL type, depending on the data types available in the database. There is much more to configuring persistent classes, as we'll discuss in this article.

Article goals

In this article, we'll cover configuring Hibernate at the framework and persistent class level. More specifically, we'll discuss the following:

- Creating a basic hibernate.cfg.xml file
- Building mapping definition files to provide Hibernate with information about persistent classes
- The primary Hibernate classes used to persist and retrieve classes
- Advanced Hibernate configuration, including object caching and transaction management
- Persisting class hierarchies (inheritance) with Hibernate

Assumptions

This chapter requires that you've completed these steps:

- Ant, Hibernate, and MySQL are installed correctly.
- Download this basic project.

- Have basic knowledge of XML for the configuration file and mapping documents.

Configuring Hibernate

Hibernate must peacefully coexist in various deployment environments, from application servers to standalone applications. We refer to these as managed and nonmanaged environments, respectively. An application server is an example of a managed environment, providing services to hosted applications like connection pools and transaction management. Two of the commonly used application servers include WebSphere and JBoss.

The alternative is a nonmanaged environment, in which the application provides any required services. Nonmanaged environments typically lack the convenience services found in managed environments. A standalone Swing or SWT application is an example of a nonmanaged environment.

Hibernate supports a number of different configuration methods and options to support these scenarios. Configuring all of Hibernate's properties can be overwhelming, so we'll start slowly. Before we jump into configuration, look at figure 1, which shows the major Hibernate classes and configuration files.

The light gray boxes in the figure are the classes your application code will use most often. The dark gray boxes are the configuration files used by the Configuration class to create the SessionFactory , which in turn creates the Session instances. Session instances are your primary interface to the Hibernate persistence service.

Let's begin with the basic configuration that can be used in any Hibernate deployment. We'll discuss advanced configuration later in this article.

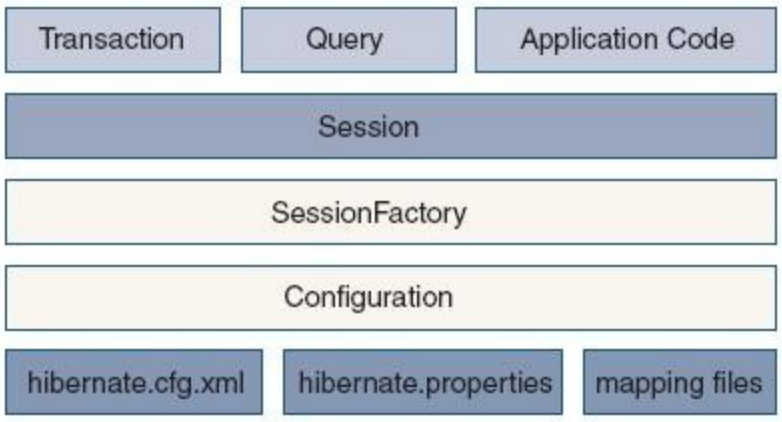

Figure 1. Primary Hibernate components

Basic configuration

Hibernate provides two alternative configuration files: a standard Java properties file called hibernate.properties and an XML formatted file called hibernate.cfg.xml. We'll

use the XML configuration file throughout this book, but it's important to realize that both configuration files perform the same function: configuring the Hibernate service. If both the hibernate.properties and hibernate.cfg.xml files are found in the application classpath, then hibernate.cfg.xml overrides the settings found in the hibernate.properties file. (Actually, we use both files in the example source code to avoid putting the database connection informationtion throughout the project directory tree.)

Before configuring Hibernate, you should first determine how the service obtains database connections. Database connections may be provided by the Hibernate framework or from a JNDI DataSource. A third method, user-provided JDBC connections, is also available, but it's rarely used.

Using Hibernate-managed JDBC connections

The sample configuration file in listing 1 uses Hibernate-managed JDBC connections. You would typically encounter this configuration in a nonmanaged environment, such as a standalone application.

Listing 1. Example hibernate.cfg.xml file

```
<?xml version="1.0">
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.username">uid</property>
    <property name="connection.password">pwd</property>
    <property name="connection.url">
       jdbc:mysql://localhost/db
    </property>
    <property name="connection.driver_class">
       com.mysql.jdbc.Driver
    </property>
    <property name="dialect">
       org.hibernate.dialect.MySQLDialect
    </property>
    <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

To use Hibernate-provided JDBC connections, the configuration file requires the following five properties:

- connection.driver_class -The JDBC connection class for the specific database
- connection.url -The full JDBC URL to the database
- connection.username -The username used to connect to the database

- connection.password -The password used to authenticate the username
- dialect -The name of the SQL dialect for the database

The connection properties are common to any Java developer who has worked with JDBC in the past. Since you're not specifying a connection pool, which we cover later in this chapter, Hibernate uses its own rudimentary connection-pooling mechanism. The internal pool is fine for basic testing, but you shouldn't use it in production.

The dialect property tells Hibernate which SQL dialect to use for certain operations. Although not strictly required, it should be used to ensure Hibernate Query Language (HQL) statements are correctly converted into the proper SQL dialect for the underlying database.

The dialect property tells the framework whether the given database supports identity columns, altering relational tables, and unique indexes, among other database-specific details. Hibernate ships with more than 20 SQL dialects, supporting each of the major database vendors, including Oracle, DB2, MySQL, and PostgreSQL.

Hibernate also needs to know the location and names of the mapping files describing the persistent classes. The mapping element provides the name of each mapping file as well as its location relative to the application classpath. There are different methods of configuring the location of the mapping file, which we'll examine later.

Using a JNDI DataSource

To use Hibernate with database connections provided by a JNDI DataSource, you need to make a few changes to the configuration file, as shown in listing 2.

Listing 2. Modified hibernate.cfg.xml file

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

<hibernate-configuration>                          Sets JNDI name
  <session-factory                                 of SessionFactory
    name="java:comp/env/hibernate/SessionFactory">
    <property name="connection.datasource">
        jdbc/myDataSource                  Specifies name of
    </property>                            JNDI DataSource
    <property name="dialect">
        org.hibernate.dialect.MySQLDialect
    </property>
    <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

You would typically use this type of configuration when using Hibernate with an application server. The connection.datasource property must have the same value as the JNDI DataSource name used in the application server configuration. The dialect property serves the same purpose as the previous configuration file example.

At this point, you have almost enough information to configure Hibernate. The next step is to create mapping definitions for the objects you intend to persist.

Creating mapping definitions

Mapping definitions, also called mapping documents, are used to provided Hibernate with information to persist objects to a relational database. The mapping files also provide support features, such as creating the database schema from a collection of mapping files.

Mapping definitions for persistent objects may be stored together in a single mapping file. Alternatively, the definition for each object can be stored in an individual mapping file. The latter approach is preferred, since storing the definitions for a large number of persistent classes in one mapping file can be cumbersome. We use the file-per-class method to organize our mapping documents throughout this book.

There is another advantage to having multiple mapping files: If you have all mapping definitions in a single file, it may be hard to debug and isolate any error to a specific class definition.

The naming convention for mapping files is to use the name of the persistent class with the hbm.xml extension. The mapping file for the Event class is thus Event.hbm.xml. The Event.hbm.xml file is shown in listing 3.

Listing 3 The Event.hbm.xml mapping file

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.manning.hq.ch03">
   <class name="Event"table="events">
      <id name="id"column="uid"type="long"unsaved-
   value="null">
         <generator class="native"/>
     </id>
     <property name="name"type="string"length="100"/>
     <property name="startDate"column="start_date"
               type="date"/>
     <property name="duration"type="integer"/>
     <many-to-one name="location"column="location_id"
                  class="Location"/>
     <set name="speakers">
        <key column="event_id"/>
        <one-to-many class="Speaker"/>
    </set>
    <set name="attendees"/>
```

```
        <key column="event_id"/>
        <one-to-many class="Attendee"/>
    </set>
  </class>
</hibernate-mapping>
```

Let's examine this mapping file in detail. The mapping definition starts with the hibernate-mapping element. The package attribute sets the default package for unqualified class names in the mapping. With this attribute set, you need only give the class name for other persistent classes listed in the mapping file, such as the Speaker and Attendee classes. To refer to a persistent class outside the given package, you must provide the fully qualified class name within the mapping document.

If Hibernate has trouble locating a class because of a missing package on, for instance, a many-to-one element, Hibernate throws a MappingException. This doesn't mean that Hibernate can't find the actual class file, but that it isn't able to navigate from one mapping definition to another.

Immediately after the hibernate-mapping tag, you encounter the class tag. The class tag begins the mapping definition for a specific persistent class. The table attribute names the relational table used to store the state of the object. The class element has a number of attributes available, altering how Hibernate persists instances of the class.

IDs and generators

The id element describes the primary key for the persistent class as well as how the key value is generated. Each persistent class must have an id element declaring the primary key for the relational table. Let's look at the id element:

```
<id name="id"column="uid"type="long"unsaved-value="null">
    <generator class="native"/>
</id>
```

The name attribute defines the property in your persistent class that will be used to store the primary key value. The id element implies that the Event class has a property also named id:

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

If the column for the primary key has a different name than your object property, the column attribute is used. For our example's purposes, this column name is uid . The values of the type and unsaved-value attributes depend on the generator used.

The generator creates the primary key value for the persistent class. Hibernate provides multiple generator implementations that use various methods to create primary key values. Some implementations increment a value stored in a shared database table, whereas others create hexadecimal strings. Another generator, called assigned , lets you generate and assign the object ID. The assigned generator allows applications to reuse legacy code, such as the UUID generator from an EJB application. A recent introduction is the select generator, which retrieves the primary key value by selecting a value from a database trigger. The generator type you choose determines its behavior based on the underlying database.

You've used the native generator class in mapping definitions. native generators provide portability for mapping documents since the framework can determine the generator method supported by the database. Generators using the native class will use identity or sequence columns depending on available database support. If neither method is supported, the native generator falls back to a high/low generator method to create unique primary key values. Databases supporting identity columns include Sybase, MySQL, Microsoft SQL Server, and IBM DB2. Oracle, PostgreSQL, and SAP DB support sequence columns.

The native generator returns a short, integer, or long value. You've set the type attribute to long, and the id property in the Event object has a type of java.lang.Long. The value of the type attribute and the property type in the object must be the same.

The unsaved-value attribute describes the value of the id property for transient instances of this class. The unsaved-value attribute affects how objects are stored. We'll discuss the impact of this attribute later in the article.

Properties

Property elements for the Event object are similar to the id element:

```
<property name="name"type="string"length="100"/>
<property name="startDate"column="start_date"type="date"/>
<property name="duration"type="integer"/>
```

Each property element corresponds to a property in the Event object. The name attribute contains the property name, whereas the type attribute specifies the property object type. The column used to store the property value defaults to the property name. The column attribute overrides this default behavior, as shown in the startDate property.

If the type attribute is omitted, Hibernate determines the type using runtime reflection. In certain cases, you must provide the property type, since reflection may not be able to determine the desired type (such as differentiating between the Hibernate DATE and TIMESTAMP types). Valid property types include the Hibernate basic types, such as integer, string, and timestamp, as well as the corresponding Java objects and primitives. However, you aren't limited to basic data types.

The property element may also contain the name of a serializable Java class or a user-defined type. You create a new user-defined type by implementing either the

org.hibernate.UserType or org.hibernate. CompositeUserType interface. The fully qualified class name of the user type or the serializable Java class is used as the property type value.

Many-to-one element

The many-to-one element defines the association to the Location class. You can also refer to this association as one-to-one-why can we call this association a many-to-one instead? Hibernate classifies one-to-one associations as two objects sharing the same primary key. One-to-one associations aren't often used with Hibernate, so we won't cover them in detail. Many-to-one associations use foreign keys to maintain the association between two persistent classes. Let's examine many-to-one associations using the association shown in figure 2.

From this figure, you can deduce that many Event instances are associated with a single Location instance. Although the figure doesn't display it, this association is unidirectional, meaning you can navigate

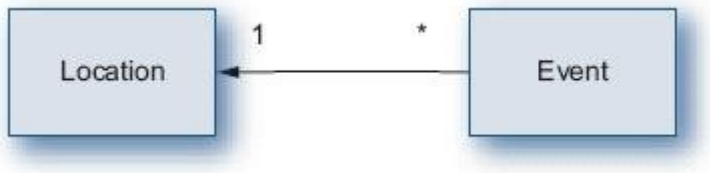

Figure 2. Association between Location and Event

from the Event instance to the Location but not from the Location to the Event instance. At this point, it's worthwhile to present the mapping file for the Location class, shown in listing 4.

Listing 4. Location.hbm.xml

```
<?xml version="1.0"?>
<hibernate-mapping package="com.manning.hq.ch03">
    <class name="Location"table="locations">
      <id name="id"column="uid"type="long">
        <generator class="native"/>
      </id>
      <property name="name"type="string"/>
      <property name="address"type="string"/>
   </class>
</hibernate-mapping>
```

The mapping for the Location class is similar to the Event mapping, although it doesn't have as many properties and lacks associations to other persistent objects. The association from Event to Location is a simple object reference.

For the Event mapping, the many-to-one element defines object references between persistent objects. Mapping a many-to-one association is straightforward:

```
   <many-to-one name="location"column="location_id"class="Location"/>
```

The name attribute gives the name of the property in the object, and the optional column attribute specifies the column used to store the foreign key to the locations table. If you don't give a column attribute, the name attribute is used as the column name. The class attribute names the associated persistent class. Remember that you don't need to give the fully qualified name of the Location class if it's in the package defined in the hibernate-mapping element.

A common question from developers new to Hibernate is how to make a many-to-one relationship lazy, meaning that the associated object won't be retrieved when the parent object is retrieved. The solution is to use proxied objects.

Proxies

An object proxy is just a way to avoid retrieving an object until you need it. Hibernate 2 does not proxy objects by default. However, experience has shown that using object proxies is preferred, so this is the default in Hibernate 3.

Object proxies can be defined in one of two ways. First, you can add a proxy attribute to the class element. You can either specify a different class or use the persistent class as the proxy. For example:

```
<class name="Location"
proxy="com.manning.hq.ch03.Location"...>...
</class>
```

The second method is to use the lazy attribute. Setting lazy="true"is a shorthand way of defining the persistent class as the proxy. Let's assume the Location class is defined as lazy:

```
<class name="Location"lazy="true"...>...</class>
```

The lazy attribute is true by default in Hibernate 3. An easy way to disable all proxies, including lazy collections, is to set the default-lazy attribute to true in the hibernate-mapping element for a given mapping file. Let's look at an example of using a proxied Location instance:

```
Session session =factory.openSession();
Event ev =(Event)session.load(Event.class,myEventId);
Location loc =ev.getLocation();
String name =loc.getName();
session.close();
```

The returned Location instance is a proxy. Hibernate populates the Location instance when getName()is called.

You'll be dealing with a proxy of Location generated by CGLIB until you call an instance method. (CGLIB is a code generation library used by Hibernate. You can find out more about it at http://cglib.sourceforge.net/.) What happens when you retrieve the Event instance from the database? All the properties for the Event are retrieved, along with the ID of the associated Location instance. The generated SQL looks something like this:

```
select event0_.id as id0_,event0_.name as name0_,
    event0_.location_id as location_id0_from events event0_
    where event0_.id=?
```

When you call loc.getName(), the following generated SQL is executed:

```
select location0_.id as id0_as id0_,location0_.name as name0_
    from locations location0_where location0_.id=?
```

If you've guessed that you can call loc.getId()without invoking a call to the database, you're correct. The proxied object already contains the ID value, so it can be safely accessed without retrieving the full object from the database.

Next, we'll look at collections of persistent objects. Like proxies, collections can also be lazily populated.

Collections

The mapping file defines the collections for Speaker's and Attendee's. Since the two collections are essentially the same, we're just going to look at the Speaker collection here. The collections are defined as sets, meaning Hibernate manages the collections with the same semantics as a java.util.Set:

```
<set name="speakers">
    <key column="event_id"/>
    <one-to-many class="Speaker"/>
</set>
```

This definition declares that the Event class has a property named speakers, and that it's a Set containing instances of the Speaker class. The Event class has the corresponding property:

```
public class Event {
    private Set speakers;
    ...
    public void setSpeakers(Set speakers){
    This.speakers =speakers;
    }

    public Set getSpeakers(){
        return this.speakers;
    }
    ...
}
```

The key element defines the foreign key from the collection table to the parent table. In this case, the speakers table has an event_id column referring to the id column in the events table. The one-to-many element defines the association to the Speaker class.

We've only touched on persisting collections with Hibernate. In addition to Sets, Hibernate also supports persistent Maps and Lists, as well as arrays of objects and primitive values.

ORGANIZING YOUR MAPPING FILES
Let's take a quick break from discussing Hibernate's persistence features and discuss a matter of practice: the location of mapping files. After you create mapping files for each persistent class, where should they be stored so the application can access them? Ideally, mapping files should be stored in the same JAR file as the classes they describe. Suppose the class file for the Event object is stored in the com/manning/hq directory and therefore in the com.manning.hq package. The Event.hbm.xml file should also be stored in the com/manning/hq directory inside the JAR archive.

Cascades

If you've worked with relational databases, you've no doubt encountered cascades. Cascades propagate certain operations on a table (such as a delete) to associated tables. (Remember that tables are associated through the use of foreign keys.) Suppose that when you delete an Event, you also want to delete each of the Speaker instances associated with the Event. Instead of having the application code perform the deletion, Hibernate can manage it for you.

Hibernate supports ten different types of cascades that can be applied to many-to-one associations as well as collections. The default cascade is none. Each cascade strategy specifies the operation or operations that should be propagated to child entities. The cascade types that you are most likely to use are the following:

- all -All operations are passed to child entities: save , update , and delete.
- save-update -Save and update (and UPDATE , respectively) are passed to child entities.
- delete -Deletion operations are passed to child entities.
- delete-orphan -All operations are passed to child entities, and objects no longer associated with the parent object are deleted.

The cascade element is added to the desired many-to-one or collection element. For example, the following configuration instructs Hibernate to delete the child Speaker elements when the parent Event is deleted:

```
<set name="speakers"cascade="delete">
  <key column="event_id"/>
  <one-to-many class="Speaker"/>
</set>
```

That's all there is to configuring cascades. It's important to note that Hibernate doesn't pass the cascade off to the database. Instead, the Hibernate service manages the cascades internally. This is necessary because Hibernate has to know exactly which objects are saved, updated, and deleted.

With the configuration and mapping files in hand, you're ready to persist objects to the database with Hibernate.

Fetching associated objects

When an object has one or more associated objects, it's important to consider how associated objects will be loaded. Hibernate 3 offers you two options. You can either retrieve associated objects using an outer join or by using a separate SELECT statement. The fetch attribute allows you to specify which method to use:

```
<many-to-one name="location"class="Location"fetch="join"/>
```

When an Event instance is loaded, the associated Location instance will be loaded using an outer join. If you wanted to use a separate select, the many-to-one element would look like this:

```
<many-to-one name="location"class="Location"fetch="select"/>
```

This also applies to child collections, but you can only fetch one collection using a join per persistent object. Additional collections must be fetched using the SELECT strategy.

If you're using Hibernate 2, the fetch attribute is not available. Instead, you must use the outer-join attribute for many-to-one associations. (There is no support for retrieving collections using a SELECT in Hibernate 2.) The outer-join attribute takes either a true or false value.

Building the SessionFactory

Hibernate's SessionFactory interface provides instances of the Session class, which represent connections to the database. Instances of SessionFactory are thread-safe and typically shared throughout an application. Session instances, on the other hand, aren't thread-safe and should only be used for a single transaction or unit of work in an application.

Configuring the SessionFactory

The Configuration class kicks off the runtime portion of Hibernate. It's used to load the mapping files and create a SessionFactory for those mapping files. Once these two functions are complete, the Configuration class can be discarded. Creating a Configuration and SessionFactory instance is simple, but you have some options. There are three ways to create and initialize a Configuration object.

This first snippet loads the properties and mapping files defined in the hibernate.cfg.xml file and creates the SessionFactory:

```
Configuration cfg =new Configuration();
SessionFactory factory =cfg.configure().buildSessionFactory();
```

The configure()method tells Hibernate to load the hibernate.cfg.xml file. Without that, only hibernate.properties would be loaded from the classpath. The Configuration class can also load mapping documents programmatically:

```
Configuration cfg =new Configuration();
cfg.addFile("com/manning/hq/ch03/Event.hbm.xml");
```

Another alternative is to have Hibernate load the mapping document based on the persistent class. This has the advantage of eliminating hard-coded filenames in the source code. For instance, the following code causes Hibernate to look for a file named com/manning/hq/ Event.hbm.xml in the classpath and load the associated class:

```
Configuration cfg =new Configuration();
cfg.addClass(com.manning.hq.ch03.Event.class);
```

Since applications can have tens or hundreds of mapping definitions, listing each definition can quickly become cumbersome. To get around this, the hibernate.cfg.xml file supports adding all mapping files in a JAR file. Suppose your build process creates a JAR file named application.jar, which contains all the classes and mapping definitions required. You then update the hibernate.cfg.xml file:

```
<mapping jar="application.jar"/>
```

Of course, you can also do this programmatically with the Configuration class:

```
Configuration.addJar(new java.io.File("application.jar"));
```

Keep in mind that the JAR file must be in the application classpath. If you're deploying a web application archive (WAR) file, your application JAR file should be in the /WEB-INF/lib directory in the WAR file.

The four methods used to specify mapping definitions to the Hibernate runtime can be combined, depending the requirements for your project. However, once you create the SessionFactory from the Configuration instance, any additional mapping files added to the Configuration instance won't be reflected in the SessionFactory . This means you can't add new persistent classes dynamically.

You can use the SessionFactory instance to create Session instances:

```
Session session =factory.openSession();
```

Instances of the Session class represent the primary interface to the Hibernate framework. They let you persist objects, query persistent objects, and make persistent objects transient. Let's look at persisting objects with Hibernate.

Persisting objects

Persisting a transient object with Hibernate is as simple as saving it with the Session instance:

```
Event event =new Event();
//populate the event
Session session =factory.openSession();
session.save(event);
session.flush();
```

Calling save(...)for the Event instance assigns a generated ID value to the instance and persists the instance. (Keep in mind that Hibernate doesn't set the ID value if the generator type is assigned.) The flush() call forces persistent objects held in memory to be synchronized to the database. Session's don't immediately write to the database when an object is saved. Instead, the Session queues a number of database writes to maximize performance.

If you would like to update an object that is already persistent, the update(...)method is available. Other than the type of SQL operation executed, the difference between save(...)and update(...)is that update(...)doesn't assign an ID value to the object. Because of this minor difference, the Session interface provides the saveOrUpdate(...) methods, which determine the correct operation to execute on the object. How does Hibernate know which method to call on an object?

When we described the mapping document, we mentioned the unsaved-value attribute. That attribute comes into play when you use the saveOrUpdate(...)method. Suppose you have a newly created Event instance. The id property is null until it's persisted by Hibernate. If the value is null, Hibernate assumes that the object is transient and assigns a new id value before saving the instance. A non-null id value indicates that the object is already persistent; the object is updated in the database, rather than inserted.

You could also use a long primitive to store the primary key value. However, using a primitive type also means that you must update the unsaved-value attribute value to 0, since primitive values can't be null.

TIP

In general, we suggest that you use object wrapper classes for primitive types in your persistent classes. To illustrate this, suppose you have a legacy database with a boolean column, which can be null. Your persistent class, mapped to the legacy table, also has a boolean property. When you encounter a row in the legacy table with a null boolean value, Hibernate throws a Property-AccessException since a boolean primitive can't be null-only true or false. However, you can avoid this problem if your persistent class property is of type java.lang.Boolean, which can be null, true, or false.

Here's the necessary code to persist an Event instance:

```
Configuration cfg =new Configuration();
SessionFactory factory =cfg.buildSessionFactory();
Event event =new Event();
//populate the Event instance
Session session =factory.openSession();
session.saveOrUpdate(event);
session.flush();
session.close();
```

The first two lines create the SessionFactory after loading the configuration file from the classpath. After the Event instance is created and populated, the Session instance, provided by the SessionFactory , persists the Event. The Session is then flushed and closed, which closes the JDBC connection and performs some internal cleanup. That's all there is to persisting objects.

Once you've persisted a number of objects, you'll probably want to retrieve them from the database. Retrieving persistent objects is the topic of the next section.

Retrieving objects

Suppose you want to retrieve an Event instance from the database. If you have the Event ID, you can use a Session to return it:

```
Event event =(Event)session.load(Event.class,eventId);
session.close();
```

This code tells Hibernate to return the instance of the Event class with an ID equal to eventId. Notice that you're careful to close the Session, returning the database connection to the pool. There is no need to flush the Session, since you're not persisting objects-only retrieving them. What if you don't know the ID of the object you want to retrieve? This is where HQL enters the picture.

The Session interface allows you to create Query objects to retrieve persistent objects. (In Hibernate 2, the Session interface supported a number of overloaded find methods. They were deprecated in Hibernate 3.) HQL statements are object-oriented, meaning that you query on object properties instead of database table and column names. Let’s look at some examples using the Query interface.

This example returns a collection of all Event instances. Notice that you don't need to provide a select ...clause when returning entire objects:

```
Query query =session.createQuery("from Event");
List events =query.list();
```

This query is a little more interesting since we're querying on a property of the Event class:

```
Query query =session.createQuery("from Event where name ="+
                                 "'Opening Presentation'");
```

```
List events =query.list();
```

We've hardcoded the name value in the query, which isn't optimal. Let's rewrite it:

```
Query query =session.createQuery("from Event where name =?",
                                 "Opening Presentation");
query.setParameter(0,"Opening Presentation",Hibernate.STRING);
List events =query.list();
```

The question mark in the query string represents the variable, which is similar to the JDBC PreparedStatement interface. The second method parameter is the value bound to the variable, and the third parameter tells Hibernate the type of the value. (The Hibernate class provides constants for the built-in types, such as STRING , INTEGER , and LONG , so they can be referenced programmatically.)

One topic we haven't touched on yet is the cache maintained by the Session. The Session cache tends to cause problems for developers new to Hibernate, so we'll talk about it next.

The Session cache

One easy way to improve performance within the Hibernate service, as well as your applications, is to cache objects. By caching objects in memory, Hibernate avoids the overhead of retrieving them from the database each time. Other than saving overhead when retrieving objects, the Session cache also impacts saving and updating objects. Let's look at a short code listing:

```
Session session =factory.openSession();
Event e =(Event)session.load(Event.class,myEventId);
e.setName("New Event Name");
session.saveOrUpdate(e);
//later,with the same Session instance
Event e =(Event)session.load(Event.class,myEventId);
e.setDuration(180);
session.saveOrUpdate(e);
session.flush();
```

This code first retrieves an Event instance, which the Session caches internally. It then does the following: updates the Event name, saves or updates the Event instance, retrieves the same Event instance (which is stored in the Session cache), updates the duration of the Event,and saves or updates the Event instance. Finally, you flush the Session.

All the updates made to the Event instance are combined into a single update when you flush the Session. This is made possible in part by the Session cache.

The Session interface supports a simple instance cache for each object that is loaded or saved during the lifetime of a given Session. Each object placed into the cache is keyed on the class type, such as The Session cache com.manning.hq.ch03.Event, and the primary key value. However, this cache presents some interesting problems for unwary developers.

A common problem new developers run into is associating two instances of the same object with the same Session instance, resulting in a NonUniqueObjectException. The following code generates this exception:

```
Session session =factory.openSession();
Event firstEvent =(Event)session.load(Event.class,myEventId);
//...perform some operation on firstEvent
Event secondEvent =new Event();
secondEvent.setId(myEventId);
session.save(secondEvent);
```

This code opens the Session instance, loads an Event instance with a given ID, creates a second Event instance with the same ID, and then attempts to save the second Event instance, resulting in the Non-UniqueObjectException.

Any time an object passes through the Session instance, it's added to the Session's cache. By passes through, we're referring to saving or retrieving the object to and from the database. To see whether an object is contained in the cache, call the Session.contains()method. Objects can be evicted from the cache by calling the Session.evict() method. Let's revisit the previous code, this time evicting the first Event instance:

```
Session session =factory.openSession();
Event firstEvent =(Event)session.load(Event.class,myEventId);
//...perform some operation on firstEvent
if (session.contains(firstEvent)){
     session.evict(firstEvent);
}
Event secondEvent =new Event();
secondEvent.setId(myEventId);
session.save(secondEvent);
```

The code first opens the Session instance and loads an Event instance with a given ID. Next, it determines whether the object is contained in the Session cache and evicts the object if necessary. The code then creates a second Event instance with the same ID and successfully saves the second Event instance.

If you simply want to clear all the objects from the Session cache, you can call the aptly named Session.clear()method.

So far, we've covered the basics of Hibernate configuration and use. Now we'll address some of the advanced configuration options that come into play when you deploy Hibernate in an application server.

Advanced configuration

Applications usually require more than a simple database connection. Scalability, stability, and performance are core aspects of any enterprise application. Popular

solutions to achieve these goals include database connection pooling, transaction strategies, and object caching. Hibernate supports each of these solutions.

Connection pools

Connection pools are a common way to improve application performance. Rather than opening a separate connection to the database for each request, the connection pool maintains a collection of open database connections that are reused. Application servers often provide their own connection pools using a JNDI DataSource, which Hibernate can take advantage of when configured to use a DataSource.

If you're running a standalone application or your application server doesn't support connection pools, Hibernate supports three connection pooling services: C3P0, Apache's DBCP library, and Proxool. C3P0 is distributed with Hibernate; the other two are available as separate distributions.

When you choose a connection pooling service, you must configure it for your environment. Hibernate supports configuring connection pools from the hibernate.cfg.xml file. The connection.provider_class property sets the pooling implementation:

```
<property name="connection.provider_class">
  org.hibernate.connection.C3P0ConnectionProvider
</property>
```

Once the provider class is set, the specific properties for the pooling service can also be configured from the hibernate.cfg.xml file:

```
<property name="c3p0.minPoolSize">
  5
</property>
...
<property name="c3p0.timeout">
  1000
</property>
```

As you can see, the prefix for the C3P0 configuration parameters is c3p0. Similarly, the prefixes for DBCP and Proxool are dbcp and proxool, respectively. Specific configuration parameters for each pooling service are available in the documentation with each service. Table 1 lists information for the supported connection pools.

Hibernate ships with a basic connection pool suitable for development and testing purposes. However, it should not be used in production. You should always use one of the available connection pooling services, like C3P0, when deploying your application to production.

If your preferred connection pool API isn't currently supported by Hibernate, you can add support for it by implementing the org.hibernate.connection.ConnectionProvider interface. Implementing the interface is straightforward.

Table 1: Connection pooling services			
Pooling Service	**Provider Class**	**Configuration Prefix**	
C3PO	Apache DBCP	Proxool	
org.hibernate.connec-tion.C3P0ConnectionProvider	org.hibernate.connec-tion.ProxoolConnectionProvider	org.hibernate.connec-tion.DBCPConnectionProvider	
c3p0	Dbcp	proxool	

There isn't much to using a connection pool, since Hibernate does most of the work behind the scenes. The next configuration topic we'll look at deals with transaction management with the Hibernate Transaction API.

Transactions

Transactions group many operations into a single unit of work. If any operation in the batch fails, all of the previous operations are rolled back, and the unit of work stops. Hibernate can run in many different environments supporting various notions of transactions. Standalone applications and some application servers only support simple JDBC transactions, whereas others support the Java Transaction API (JTA).

Hibernate needs a way to abstract the various transaction strategies from the environment. Hibernate has its own Transaction class that is accessible from the Session interface, demonstrated here:

```
Session session =factory.openSession();
Transaction tx =session.beginTransaction();
Event event =new Event();
//...populate the Event instance
session.saveOrUpdate(event);
tx.commit();
```

In this example, factory is an initialized SessionFactory instance. This code creates an instance of the org.hibernate.Transaction class and then commits the Transaction instance.

Notice that you don't need to call session.flush(). Committing a transaction automatically flushes the Session object. The Event instance is persisted to the database when the transaction is committed. The transaction strategy you use (JDBC or JTA) doesn't matter to the application code-it's set in the Hibernate configuration file.

The transaction.factory_class property defines the transaction strategy that Hibernate uses. The default setting is to use JDBC transactions since they're the most common. To use JTA transactions, you need to set the following properties in hibernate.cfg.xml:

```
<property name="transaction.factory_class">
```

```
    org.hibernate.transaction.JTATransactionFactory
</property>
<property name="jta.UserTransaction">
    java:comp/UserTransaction
</property>
```

The transaction.factory_class property tells Hibernate that you'll be using JTA transactions. Currently, the only other option to JTA is JBDC transactions, which is the default. JTA transactions are retrieved from a JNDI URI, which is specified using the jta.User-Transaction property. If you don't know the URI for your specific application server, the default value is java:comp/UserTransaction.

There is some confusion about another property related to JTA transactions: transaction.manager_lookup_class. You only need to specify the manager lookup class when you're using a transactional cache. (We discuss caches in the next section–don't worry.) However, if you don't define the jta.UserTransaction property and transaction.manager_lookup_class is defined, the user transaction name in the lookup factory class is used. If neither of the properties are used, Hibernate falls back to java:comp/UserTransaction.

What's the benefit of using JTA transactions? JTA transactions are useful if you have multiple transactional resources, such as a database and a message queue. JTA allows you to treat the disparate transactions as a single transaction. Combining multiple transactions also applies within Hibernate. If you attempt to create multiple transactions from the same Session instance, all of the operations are batched into the first transaction. Let's look at an example that includes two transactions:

```
Transaction tx0 =session.beginTransaction();
Event event =new Event();
//...populate the event instance
session.saveOrUpdate(event);
Transaction tx1 =session.beginTransaction();
Location location =new Location();
//...populate the Location instance
session.saveOrUpdate(location);
tx0.commit();
tx1.commit();
```

This example begins by creating a new transaction. The second use of session.beginTransaction()just returns the first transaction instance. session.saveOrUpdate(location)commits the first transaction, and tx0.commit()recommits the first transaction.

Although you explicitly create two Transaction objects, only one is used. Of course, this creates a problem. Let's assume you have a Session object being used by two application threads. The first application thread begins the JTA transaction and starts adding objects. Meanwhile, the second thread, using the same transaction, deletes an object and commits the transaction. Where does this leave the first thread?

The first thread won't be committed, which is what you'd expect. The problem is that this issue can be hard to debug, bringing up an important point: Sessions should be

used by only one application thread at a time. This is a common concern in web applications, which are multithreaded by their very nature.

In the next section, we discuss Hibernate's support for various caching providers.

Cache providers

As we mentioned earlier, caching is a common method used to improve application performance. Caching can be as simple as having a class store frequently used data, or a cache can be distributed among multiple computers. The logic used by caches can also vary widely, but most use a simple least recently used (LRU) algorithm to determine which objects should be removed from the cache after a configurable amount of time.

Before you get confused, let's clarify the difference between the Session–level cache, also called the first–level cache, and what this section covers. The Session–level cache stores object instances for the lifetime of a given Session instance. The caching services described in this section cache data outside of the lifetime of a given Session. Another way to think about the difference is that the Session cache is like a transactional cache that only caches the data needed for a given operation or set of operations, whereas a second–level cache is an application-wide cache.

NOTE
Caching services are typically referred to as second-level caches elsewhere in this book and in other Hibernate documentation. When you see it mentioned in the text, we're referring to external caching services.

By default, Hibernate supports four different caching services, listed in table 2. EHCache (Easy Hibernate Cache) is the default service. If you prefer to use an alternative cache, you need to set the cache.provider_class property in the hibernate.cfg.xml file:

```
<property name="cache.provider_class">
  org.hibernate.cache.OSCacheProvider
</property>
```

This snippet sets the cache provider to the OSCache caching service.

Table 2: Caching Services Supported by Hibernate

Caching Service	Provider Class	Type
EHCache	org.hibernate.cache.EhCacheProvider	Memory,disk
OSCache	org.hibernate.cache.OSCacheProvider	Memory,disk
SwarmCache	org.hibernate.cache.SwarmCacheProvider	Clustered
TreeCache	org.hibernate.cache.TreeCacheProvider	Clustered

The caching services support the caching of classes as well as collections belonging to persistent classes. For instance, suppose you have a large number of Attendee instances associated with a particular Event instance. Instead of repeatedly fetching the collection of Attendee s, you can cache it. Caching for classes and collections is configured in the mapping files, with the cache element:

```
<class name="Event"table="events">
   <cache usage="read-write"/>
   ...
</class>
```

Collections can also be cached:

```
<set name="attendees">
   <cache usage="read-write"/>
   ...
</set>
```

Once you've chosen a caching service, what do you, the developer, need to do differently to take advantage of cached objects? Thankfully, you don't have to do anything. Hibernate works with the cache behind the scenes, so concerns about retrieving an outdated object from the cache can be avoided. You only need to select the correct value for the usage attribute.

The usage attribute specifies the caching concurrency strategy used by the underlying caching service. The previous configuration sets the usage to read–write , which is desirable if your application needs to update data. Alternatively, you may use the nonstrict–read–write strategy if it's unlikely two separate transaction threads could update the same object. If a persistent object is never updated, only read from the database, you may specify set usage to read-only.

Some caching services, such as the JBoss TreeCache, use transactions to batch multiple operations and perform the batch as a single unit of work. If you choose to use a transactional cache, you may set the usage attribute to transactional to take advantage of this feature. If you happen to be using a transactional cache, you'll also need to set the transaction.manager_lookup_class mentioned in the previous section.

The supported caching strategies differ based on the service used. Table 3 shows the supported strategies.

Table 3: Supported Caching Service Strategies

Caching Service	Read-only	Read-write	Nonstrict-read-write	Transactional

EHCache	Y	Y	Y	N
OSCache	Y	Y	Y	N
SwarmCache	Y	Y	Y	N
TreeCache	Y	N	N	Y

Clearly, the caching service you choose will depend on your application requirements and environment. Next, let's look at configuring EHCache.

Configuring EHCache

By now you're probably tired of reading about configuring Hibernate, but EHCache is pretty simple. It's a single XML file, placed in a directory listed in your classpath. You'll probably want to put the ehcache.xml file in the same directory as the hibernate.cfg.xml file.

Listing 5 shows a simple configuration file for EHCache.

Listing 5. ehcache.xml file

```
<ehcache>
  <diskStore path="java.io.tmp"/>
  <defaultCache
    maxElementsInMemory="10"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
  <cache name="com.manning.hq.ch03.Event"
    maxElementsInMemory="20"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="180"
    overflowToDisk="true"/>
</ehcache>
```

In this example, the diskStore property sets the location of the disk cache store. Then, the listing declares two caches. The defaultCache element contains the settings for all cached objects that don't have a specific cache element: the number of cached objects held in memory, whether objects in the cache expire (if eternal is true , then objects don't expire), the number of seconds an object should remain the cache after it was last accessed, the number of seconds an object should remain in the cache after it was created, and whether objects exceeding maxElementsInMemory should be spooled to the diskStore. Next, for custom settings based on the class, the code defines a cache element with the fully qualified class name listed in the name attribute. (This listing only demonstrates a subset of the available configuration for EHCache. Please refer to the documentation found at http:// ehcache.sf.net for more information.)

With pooling, transactions, and caching behind us, we can look at a difference topic: how Hibernate handles inheritance.

Inheritance

Inheritance is a fundamental concept of object-oriented languages. Through inheritance, objects can inherit the state and behavior of their ancestor, or superclass. The most common use of object inheritance in applications is to create a generic base type with one or more specialized subclasses. Persisting a class hierarchy can be difficult, since each hierarchy can have its own unique requirements.

To address the problems found in hierarchy persistence, Hibernate supports three different inheritance persistence strategies:

- Table per class hierarchy
- Table per subclass
- Table per concrete class

Each mapping strategy is incrementally more complicated. In the following sections, we'll discuss the first two inheritance strategies. We've never needed to use the third, and most complicated, strategy.

Table per class hierarchy

This strategy is the most basic and easiest to use. All the classes in the hierarchy are stored in a single table. Suppose you have the base Event class, with ConferenceEvent and NetworkingEvent as subclasses. The mapping definition for this hierarchy is shown in listing 6.

Listing 6. Table per class hierarchy mapping

```
<class name="Event"table="events"discriminator-value="EVENT">
   <id name="id"type="long">
     <generator class="native"/>
   </id>
   <discriminator column="event_type"type="string"length="15"/>
   ...
   <subclass name="ConferenceEvent"discriminator-
value="CONF_EVENT">
      <property name="numberOfSeats"column="num_seats"/>
      ...
  </subclass>
  <subclass name="NetworkingEvent"discriminator-
value="NET_EVENT">
    <property name="foodProvided"column="food_provided"/>
    ...
  </subclass>
</class>
```

We've introduced a few new features in the mapping definition. The most important is the inclusion of the discriminator element. The discriminator column is what Hibernate uses to tell the different sub-classes apart when retrieving classes from the database. If you don't specify a discriminator value, Hibernate uses the object's class name. The discriminator element in the example mapping tells Hibernate to look in the event_type column for a string describing the class type.

The discriminator is only a column in the relational table-you don't need to define it as a property in your Java object.

The subclass element contains the properties and associations belonging to the subclass. Any association element is allowed between sub- class tags. You can't have an id element or a nested subclass element.

The table per class hierarchy strategy requires a single table, events, to store the three types of Event instances. Let's look at what our events table would look like with the table per hierarchy strategy, as shown in figure 3.

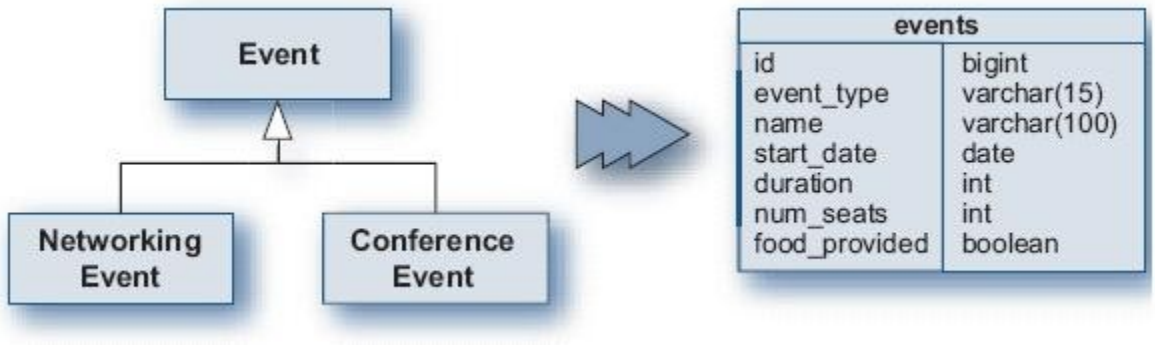

Figure 3: Table per hierarchy mapping

As you can see, one table contains the fields for all the objects in the hierarchy. The only obvious limitation is that your subclasses can‘t have columns declared as NOT NULL. Subclasses can't have non-null attributes because inserting the superclass, which doesn‘t even have the non-null attribute, will cause a null column violation when it‘s inserted into the database. The next inheritance strategy, table per sub-class, doesn't have this limitation.

Table per subclass

Instead of putting all the classes into a single table, you can choose to put each subclass into its own table. This approach eliminates the discriminator column and introduces a one-to-one mapping from the sub-class tables to the superclass table. The mapping definition for this strategy is shown in listing 7.

Listing 7. Table-per-subclass mapping

```
<class name="Event"table="events">
  <id name="event_id"type="long">
    <generator class="native"/>
```

```
    </id>
    <joined-subclass name="ConferenceEvent"table="conf_events">
       <key column="event_id"/>
       ...
    </joined-subclass>
    <joined-subclass name="NetworkingEvent"table="net_events">
       <key column="event_id"/>
       ...
    </joined-subclass>
  </class>
```

The joined-subclass element can contain the same elements as the subclass element. The key element contains the primary key associa-tion to the superclass, Event. Figure 4 shows the resulting relational schema.

Creating an association to an Event or one of its subclasses is a simple many–to–one element:

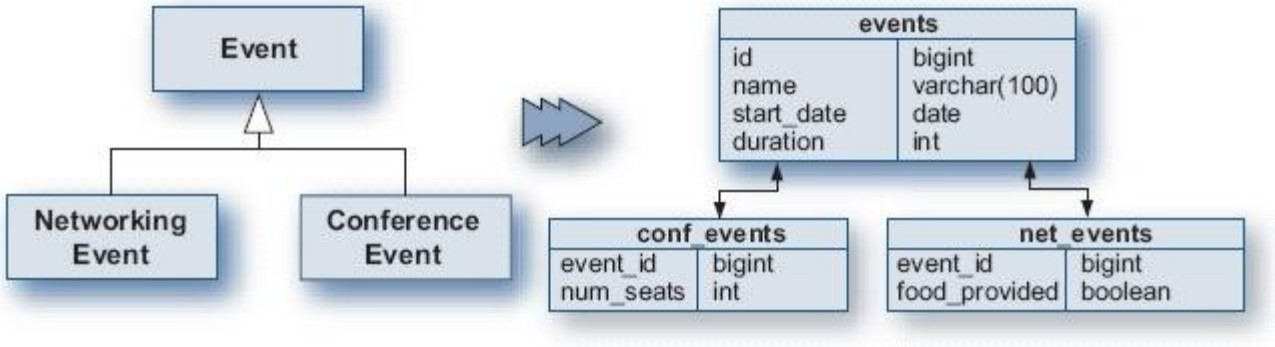

Figure 4: Table per subclass hierarchy

```
    <many-to-one class="Event"column="event"/>
```

Since this association can refer to any class in the Event hierarchy, the association is referred to as a polymorphic association. You can also create a concrete association by giving the name of the specific subclass:

```
    <many-to-one class="NetworkingEvent"column="event"/>
```

Persisting class hierarchies may seem like a complicated proposition, but Hibernate makes it fairly straightforward.

Summary

We've covered quite a bit of ground in this article. Starting with the most basic Hibernate configuration, we explored mapping file definitions and advanced configuration options.

As a persistence service, Hibernate operates in managed and nonmanaged environments. The configuration file, hibernate.cfg.xml, specifies how Hibernate

obtains database connections-either from a JNDI DataSource or from a JDBC connection pool. Additionally, the map-ping definition files describing the persistent classes may be specified in the configuration file.

Mapping files provide Hibernate with the necessary information to persist objects to a relational database. Each persistent property of a class is defined in the mapping file, including collections and associations to other persistent objects. The mapping file also defines the mandatory primary key for persistent objects.

The primary key is defined using the id element. The id element provides the name of the object property, the column used to persist the primary key, and the strategy used to generate the primary key value. Hibernate supports 10 generator strategies, including the assigned strategy that lets you assign a primary key value outside of Hibernate.

Once the configuration and mapping files are written, the Configuration object loads the files and is used to create a SessionFactory. The SessionFactory only needs to be initialized once and can be reused throughout the application. The SessionFactory creates instances of the Session interface. Session instances are basically database connections with some additional functionality.

The Session interface is the primary developer interface to Hibernate. Using it, you can persist transient objects and make persistent objects transient. It also provides querying capabilities and transaction support. Unlike the SessionFactory, Session instances should not be reused throughout the application. Instead, a new Session instance should be obtained for each transaction.

Additional pluggable components supported by Hibernate include database connection pool services, transaction management, and object caching services. These components can improve performance by reusing or caching objects and improving transaction management.

Hibernate is flexible enough to be used in any Java application environment. In this article, we examined how to configure it to support application persistence in managed and nonmanaged environments, as well as how to create the SessionFactory and persist objects.