TheServerSide
Your J2EE Community

Published on TheServerSide July 15th, 2003

A major portion of the development of an enterprise application involves the creation and maintenance of the persistence layer used to store and retrieve objects from the database of choice. Many organizations resort to creating homegrown, often buggy, persistence layers. If changes are made to the underlying database schema, it can be expensive to propagate those changes to the rest of the application. Hibernate steps in to fill this gap, providing an easy-to-use and powerful object-relational persistence framework for Java applications.

Hibernate provides support for collections and object relations, as well as composite types. In addition to persisting objects, Hibernate provides a rich query language to retrieve objects from the database, as well as an efficient caching layer and Java Management Extensions (JMX) support. User-defined data types and dynamic beans are also supported.

Hibernate is released under the Lesser GNU Public License, which is sufficient for use in commercial as well as open source applications. It supports numerous databases, including Oracle and DB2, as well as popular open source databases such as PostgreSQL and MySQL. An active user community helps to provide support and tools to extend Hibernate and make using it easier.

This article covers Hibernate 2.0.1, which was released on June 17, 2003.

How Hibernate Works

Rather than utilize bytecode processing or code generation, Hibernate uses runtime reflection to determine the persistent properties of a class. The objects to be persisted are defined in a mapping document, which serves to describe the persistent fields and associations, as well as any subclasses or proxies of the persistent object. The mapping documents are compiled at application startup time and provide the framework with necessary information for a class. Additionally, they are used in support operations, such as generating the database schema or creating stub Java source files.

A `SessionFactory` is created from the compiled collection of mapping documents. The `SessionFactory` provides the mechanism for managing persistent classes, the `Session` interface. The `Session` class provides the interface between the persistent data store and the application. The `Session` interface wraps a JDBC connection, which can be user-managed or controlled by Hibernate, and is only intended to be used by a single application thread, then closed and discarded.

The Mapping Documents

Our example utilizes two trivial classes, `Team` and `Player`. The mappings for these classes are shown below.

```
<hibernate-mapping>
  <class name="example.Team" table="teams">
    <id name="id" column="team_id" type="long" unsaved-value="null">
      <generator class="hilo"/>
    </id>
    <property name="name" column="team_name" type="string"
              length="15" not-null="true"/>
    <property name="city" column="city" type="string"
              length="15" not-null="true"/>
    <set name="players" cascade="all" inverse="true" lazy="true">
      <key column="team_id"/>
      <one-to-many class="example.Player"/>
    </set>
  </class>
</hibernate-mapping>
```

Figure 1 – example.Team mapping document.

```
<hibernate-mapping>
  <class name="example.Player" table="players">
    <id name="id" column="player_id" type="long" unsaved-value="null">
      <generator class="hilo"/>
    </id>
    <property name="firstName" column="first_name" type="string"
              length="12" not-null="true"/>
    <property name="lastName" column="last_name" type="string"
              length="15" not-null="true"/>
    <property name="draftDate" column="draft_date" type="date"/>
    <property name="annualSalary" column="salary" type="float"/>
    <property name="jerseyNumber" column="jersey_number"
              type="integer" length="2" not-null="true"/>
    <many-to-one name="team" class="example.Team" column="team_id"/>
  </class>
</hibernate-mapping>
```

Figure 2 – example.Player mapping document

The mapping documents are reasonably clear, but certain areas warrant explanation. The `id` element block describes the primary key used by the persistent class. The attributes of the `id` element are:

- `name`: The property name used by the persistent class.
- `column`: The column used to store the primary key value.
- `type`: The Java data type used. In this case, we're going to use `longs`.
- `unsaved-value`: This is the value used to determine if a class has been made persistent, i.e., stored to the database. If the value of the id attribute is null, Hibernate knows that this object has not been persisted. This is important when calling the `saveOrUpdate()` method, discussed later.

The generator element describes the method used to generate primary keys. I've chosen to use the hilo generator for purposes of illustration. The hilo generator will use a supporting table to create the key values. If this method doesn't appeal to you, don't worry. In Hibernate 2.0, ten primary key generation methods are available and it's possible to create your own mechanism, including composite primary keys.

The property elements define standard Java attributes and how they are mapped to columns in the schema. Attributes are available to specify column length, specific SQL types, and whether or not to accept null values. The property element supports the column child element to specify additional properties, such as the index name on a column or a specific column type.

Our Team class has an additional element block for the collection of Players that belong to a Team:

```
    <set name="players" cascade=”all” inverse=”true” lazy=”true”>
      <key column="team_id"/>
      <one-to-many class="example.Player"/>
    </set>
```

Figure 3 – example.Team collection defintion

Figure 3 defines a set of Players that will be mapped to the Team using the bi-directional mapping defined in the Player class, which Hibernate will create when the schema is generated. The key element is used to distinguish an instance of the collection using a foreign key to the owning entity. The one-to-many element specifies the collected class and the column used to map to the entity.

Two attributes in the set element are of interest: lazy and inverse. Marking a collection as lazy=”true” means that the collection will not be automatically populated when the object containing the collection is retrieved from the database. For example, if we retrieve a Team from the database, the set of Players will not be populated until the application accesses it. Lazy initialization of collections will be explained in more detail in the Performance Considerations section.

The inverse attribute allows this set to be bi-directional, meaning that we can determine the Team that a Player belongs to with the following entry from the Player mapping document:

```
    <many-to-one name="team" class="example.Team" column="team_id"/>
```

Figure 4 – bi-directional association from the Player class to the Team class

The line shown in Figure 4 will create a bi-directional association from the Player to its associated Team.

Hibernate Properties

The properties that Hibernate uses to connect to the database and generate the schema are stored in a file called `hibernate.properties`. For our purposes, this file only has five properties, but many more are available:

```
hibernate.connection.username=ralph
hibernate.connection.password=nader
hibernate.connection.url=jdbc:postgresql://localhost/example
hibernate.connection.driver_class=org.postgresql.Driver
hibernate.dialect=net.sf.hibernate.dialect.PostgreSQLDialect
```

Figure 5 – example hibernate.properties

The first four property values are familiar to any developer that has worked with JDBC. The last property, `hibernate.dialect`, defines the SQL dialect used when converting the Hibernate Query Language (HQL) into SQL, as well as when generating the database schema for initial use. If we chose to use Oracle instead of PostgreSQL in the future, we'd simply change the dialect used and update the connection parameters as necessary. The HQL statements would largely stay the same except for features unique to a given database, such as the lack of nested select statements in MySQL.

The Schema

Mapping files in hand, it's time to generate the database schema. Hibernate ships with the SchemaExport utility that will create the schema necessary for the mapping documents. This utility may be run from the command line or from an Ant build script to connect to the database and create the schema, or to export the schema to a file.

```
java -cp classpath \
net.sf.hibernate.tool.hbm2ddl.SchemaExport options mapping_files
```

Figure 6 – SchemaExport usage

This is what our schema looks like:

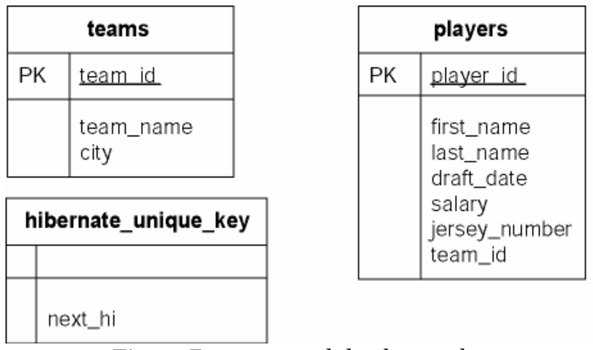

Figure 7 – generated database schema

The `hibernate_unique_key` table is used to store the id value used for the `hilo` generator type.

The Source Files

Rather than create the persistent classes by hand, I've chosen to use the `CodeGenerator` that ships with the Hibernate Extensions package. The `CodeGenerator` will create stub files based on the mapping documents described above, which are suitable for our needs. (The code bundle supporting this article can be found in the Resources section.) Using the `CodeGenerator` is similar to the `SchemaExport` utility:

```
java -cp classpath \
net.sf.hibernate.tool.hbm2java.CodeGenerator options mapping_files
```

Figure 8 - CodeGenerator usage

The generated classes have the following structure (constructors removed from diagram for brevity):

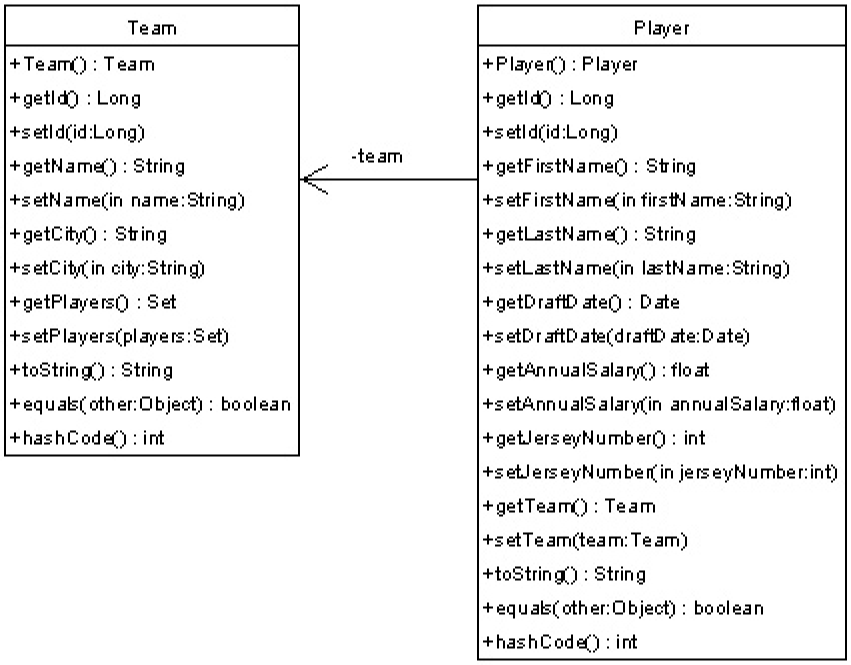

Figure 9 – diagram of example classes generated by Hibernate

Creating the SessionFactory

The `SessionFactory` stores the compiled mapping documents specified when the factory is created. Configuring the `SessionFactory` is fairly straightforward. All of the mappings are added to an instance of `net.sf.hibernate.cfg.Configuration`, which is then used to create the `SessionFactory` instance.

```
Configuration cfg = new Configuration()
    .addClass(example.Player.class)
    .addClass(example.Team.class);
SessionFactory factory = cfg.buildSessionFactory();
```

Figure 10 – Configuring and creating a SessionFactory

The `Configuration` class is only needed for the creation of the `SessionFactory` and can be discarded after the factory is built. Instances of `Session` are obtained by calling `SessionFactory.openSession()`. The logical lifecycle of a `Session` instance is the span of a database transaction.

The `SessionFactory` can also be configured using an XML mapping file, placed in the root of your classpath. The obvious advantage to this approach is that your configuration isn't hardcoded in the application.

Creating and Updating Persistent Classes

As far as Hibernate is concerned, classes are either transient or persistent. Transient classes are instances that have not been saved to the database. To make a transient instance persistent, simply save it using the `Session` class:

```
Player player = new Player();
// … populate player object
Session session = SessionFactory.openSession();
session.saveOrUpdate(player);
```

Figure 11 – saving persistent objects

The `saveOrUpdate(Object)` call will save the object if the id property is `null`, issuing a SQL `INSERT` to the database. This refers to the `unsaved-value` attribute that we defined in the `Player` mapping document. If the id is not `null`, the `saveOrUpdate(Object)` call would issue an update, and a SQL `UPDATE` would be issued to the database. (Please refer to the sidebar *Unsaved-Value Strategies* for more information on this topic.)

To create and save a `Team` with assigned `Players`, follow the same pattern of creating the object and saving it with a `Session` instance:

```
Team team = new Team();
team.setCity("Detroit");
team.setName("Pistons");

// add a player to the team.
Player player = new Player();
player.setFirstName("Chauncey");
player.setLastName("Billups");
player.setJerseyNumber(1);
player.setAnnualSalary(4000000f);
Set players = new HashSet();
players.add(player);

team.setPlayers(players);
// open a session and save the team
Session session = SessionFactory.openSession();
session.saveOrUpdate(team);
```

Figure 12 – persisting objects

This will persist the `Team` instance and each of the `Player` instances in the `Set`.

Unsaved Value Strategies

The `unsaved-value` attribute supported by the `id` element indicates when an object is newly created and transient, versus an object already persisted. The default value is `null`, which should be sufficient for most cases. However, if your identifier property doesn't default to null, you should give the default value for a transient (newly created) object.

Other values supported by the unsaved-value attribute are:

- `any`
- `none`
- `id-value`

Retrieving Persistent Classes

If you know the primary key value of the object that you want to retrieve, you can load it with the `Session.load()` method. This method is overloaded to provide support for standard classes and BMP entity beans.

```
// method 1: loading a persistent instance
Session session = SessionFactory.createSession();
Player player = session.load(Player.class, playerId);

// method 2: loading the Player's state
Player player = new Player();
session.load(player, playerId);
```

Figure 13 – Loading persistent instances

To retrieve a persistent class without knowing its primary key value, you can use the `Session.find()` methods. The `find()` method allows you to pass an HQL (Hibernate Query Language) statement and retrieve matching objects as a `java.util.List`. The `find()` method has three signatures, allowing you to pass arguments to JDBC-like "?" parameters as a single argument, named parameters, or as an `Object[]`. (Please refer to the sidebar *Hibernate Query Language* for more information on HQL.)

Hibernate Query Language

Queries written in HQL are essentially as powerful as their SQL counterparts. Inner and outer joins are supported, as are various functions such as `avg(…)`, `sum(…)`, `min(…)`, and `count(…)`. HQL also supports many other SQL-like functions and operations such as `distinct` and `like`. Subqueries are also supported if supported by the underlying database, as is the `group by` clause.

Named parameters allow you to specify names in the HQL statements instead of question marks as parameter flags. For example:

```
      select team.id from team in class example.Team where
      team.name=:name
```

To set the value of the `:name` parameter, use the `Query.setParameter(…)` method. For the aforementioned statement, it would look like:

```
      query.setParameter("name", "Pistons", Hibernate.STRING);
```

HQL is a very rich object query language and, because of its depth, will be the subject of a future article.

Deleting Persistent Classes

Making a persistent object transient is accomplished with the `Session.delete()` method. This method supports passing either a specific object to delete or a query string to delete multiple objects from the database.

```
// method 1 – deleting the Player loaded in figure 12
session.delete(player);

// method 2 – deleting all of the Players with a
//             salary greater than 4 million
session.delete("from player in class example.Player where player.annualSalary
> 4000000");
```

Figure 14 – deleting a persistent object

It's important to note that while the object may be deleted from the database, your application may still hold a reference to the object. Deleting an object with collections of objects, such as the `Team`'s set of `Players`, can cascade to child objects by specifying `cascade="delete"` for the `set` element in the mapping document.

Collections

Hibernate can manage the persistence of object collections, whether they are Sets, Maps, Lists, arrays of objects or primitive values. It also allows another form of collection called a "bag". A bag can be mapped to a `Collection` or `List`, and contains an unordered, unindexed collection of entities. Bags can contain the same element many times.

Additional semantics supported by implementing classes, such as `LinkedList`, are not maintained when persisted. Another note is that the property of a collection must be the interface type (`List`, `Map`, `Set`). This is because, in order to support lazy collections, Hibernate uses it's own implementations of the `List`, `Map` or `Set` interfaces.

When accessing a lazily initialized collection, it's important to remember that a `Session` must be open, or an exception will be thrown:

```
Session session = factory.openSession();
Team team = (Team) session.find("from team in class example.Team where
team.city = ?", cityName, Hibernate.STRING).get(0);

Set players = team.getPlayers();
session.close();

Player p = (Player) players.get(0);  // exception will be thrown here
```

Figure 15 – incorrect use of lazy initialization

The exception is thrown in Figure 15 because the `Session` needed to populate `players` was closed prematurely. Because of the potential for this bug, Hibernate defaults to non-lazy collections. However, lazy collections should be used for performance reasons.

Performance Considerations

Fortunately this functionality doesn't come at much of a performance cost. The Hibernate website claims that its "overhead is much less than 10% of the JDBC calls," and my experience in deploying applications using Hibernate supports this. Hibernate can make multiple optimizations when interacting with the database, including caching objects, efficient outer join fetching and executing SQL statements only when needed. It is difficult to achieve this level of sophistication with hand-coded JDBC.

A link to the performance FAQ on the Hibernate website can be found in the Resources section.

Alternative Persistence Frameworks

Hibernate isn't the only framework available for mapping objects to persistent data stores. I encourage you to evaluate each of them and choose the best one for your needs. Some alternative frameworks, listed in no particular order, are:

- OJB. "ObjectRelationalBridge (OJB) is an Object/Relational mapping tool that allows transparent persistence for Java Objects against relational databases." Apache license. http://db.apache.org/ojb/
- Castor. "Castor is an open source data binding framework for Java[tm]." BSD-like license. http://castor.exolab.org/
- CocoBase. "CocoBase® offers a simple to use, powerful Dynamic Object to Relational Mapping™ tool for Java developers writing applications on the J2EE, J2SE and J2ME platforms." Commercial. http://www.thoughtinc.com/cber_index.html
- TopLink. "With TopLink, developers can map both Java Objects and Entity Beans to a relational database schema." TopLink was recently purchased by Oracle. Commercial. http://www.oracle.com/features/9iAS/index.html?t1as_toplink.html

Conclusion

This article has given you an introduction to what Hibernate can do. Hibernate delivers a high-performance, open source persistence framework comparable to many of its open source and commercial counterparts. Developers utilizing Hibernate can greatly reduce the amount of time and effort needed to code, test, and deploy applications. However, we've only scratched the surface and I encourage you to explore Hibernate for yourself.

About the Author

Nick Heudecker is a software developer with more than six years of experience designing and building enterprise applications. His firm, System Mobile, Inc., specializes in application integration, custom software development and wireless applications. He is a Sun Certified Java Programmer and is located in Ann Arbor, Michigan.

Resources

- The example source code and mapping documents can be found: http://www.systemmobile.com/articles/hibernate.zip
- Hibernate website: http://hibernate.bluemars.net/
- Hibernate performance FAQ: http://hibernate.bluemars.net/15.html
- Hibernate feature list: http://hibernate.bluemars.net/4.html
- A comparison a various ORM tools: http://c2.com/cgi-bin/wiki?ObjectRelationalToolComparison
- The System Mobile website: http://www.systemmobile.com/