USING HIBERNATE FOR PERSISTENCE IN JAVA APPLICATIONS

by
Ciprian Ioan Ileană, Paul-Mihai Bălşan

Abstract: In all Java applications one of the most important components is the persistence. The persistence that is used and the way it is used can make the difference between a slow and a fast application. Hibernate is a powerful OR persistence and query service for Java, that lets you develop persistent objects following common Java idiom (this includes association, inheritance, polymorphism, composition and the Java collections framework). The Hibernate Query Language is designed as a minimal object-oriented extension to SQL and provides an elegant bridge between the object and relational worlds. The paper contains a full description of Hibernate (including its architecture) and examples of how to use it (including code samples).

1. Hibernate Architecture

1.1 Architecture Overview

First, let's see a very high-level view of Hibernate architecture. In this diagram you will see how Hibernate uses the database and the configuration data in order to provide the persistence services and the persistent objects to the application.

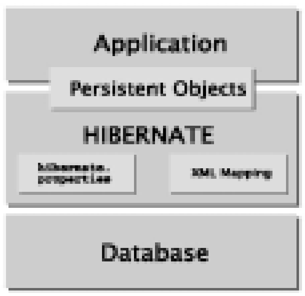

Figure 1 - High Level view of Hibernate architecture

Now we should look at something a little more detailed view of the runtime architecture, but this is hard to achieve because of the extreme flexibility of Hibernate. Anyway, we will present to you the most important ones (the extremes).

The first one is the "lite" architecture which has the application provide and manage its own JDBC connections and transactions. This approach uses a minimal subset of Hibernate's APIs.

In the "full cream" architecture the application is abstracted away from the underlying JDBC/JTA APIs and lets Hibernate take care of the details.

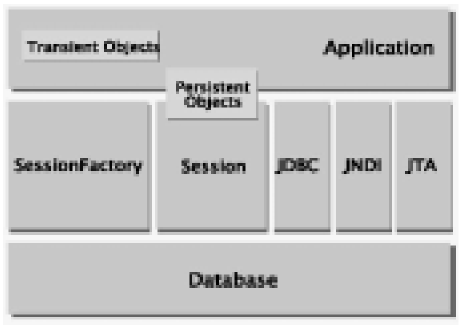

Figure 2 - Hibernate "lite" architecture

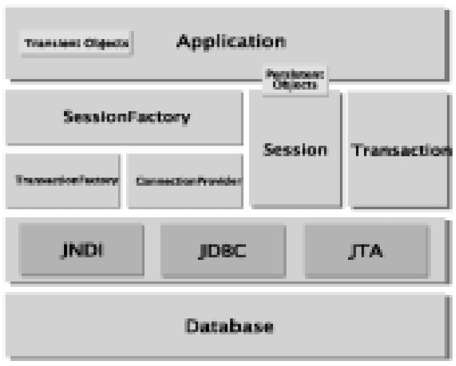

Figure 3 - Hibernate "full cream" architecture

For a better understanding of the diagrams presented above, we'll present here some definitions of the objects in the diagrams:

SessionFactory. A threadsafe (immutable) cache of compiled mappings. A factory for Session. A client of Connection-Provider. Might hold a cache of data that is be reusable between transactions.

Session. A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC connection. Factory for Transaction. Holds a cache of persistent objects.

Persistent Objects and Collections. Short-lived, single threaded objects containing persistent state and business function. These might be ordi-nary JavaBeans, the only special thing about them is that they are currently associated with (exactly one) Session.

Transient Objects and Colleactions. Instances of persistent classes that are not currently associated with a Session. They may have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed Session.

Transaction (Optional). A single-threaded, short-lived object used by the application to specify atomic units of work. Abstracts application from underlying JDBC, JTA or CORBA transaction. A Session might span several Transactions.

ConnectionProvider (Optional). A factory for (and pool of) JDBC connections. Abstracts application from underlying Data-source or DriverManager. Not exposed to application.

TransactionFactory (Optional). A factory for Transaction instances. Not exposed to the application.

1.2 Persistent Object Identity

The application may concurrently access the same persistent state in two different sessions. However, an in-stance of a persistent class is never shared between two Session instances. Hence there are two different no-tions of identity:

Table 1- Persistent Object Identity

Persistent Identity	JVM Identity
foo.getId().equals(bar.getId())	foo==bar

Then for objects returned by a particular Session, the two notions are equivalent. However, while the applica-tion might concurrently access the same (persistent identity) business object in two different sessions, the two instances will actually be different (JVM identity).

This approach leaves Hibernate and the database to worry about concurrency (the application never needs to synchronize on any business object, as long as it sticks to a single thread per Session) or object identity (within a session the application may safely use == to compare objects).

1.3 JMX Integration

JMX is the J2EE standard for management of Java components. Hibernate may be managed via a JMX stan-dard MBean but because most application servers do not yet support JMX, Hibernate also affords some non-standard configuration mechanisms.

2 Configuring SessionFactory

Hibernate is designed to operate in many different environments and this results in a large number of configuration parameters. In order to help the developer, Hibernate is distributed with an example of hibernate.properties file that shows various options available.

When configuring Hibernate you have two options, either user programmatic configuration or load a hibernate.properties file where the configuration is already specified. We will not present those in this paper, but if you want to see more information on this, please take a look in hibernate_reference.pdf which is included in Hibernate 2.0.2 distribution.

2.1 Obtaining a SessionFactory

After loading the configuration, your application must obtain a factory for session instances. The obtained factory is intended to be used by all application threads. Anyway if you are working with more than one database, Hibernate allows your application to instantiate more than one SessionFactory.

Code sample 1 - Obtaining a SessionFactory

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

2.2 Database connection

A SessionFactory allows you to open a Session on a user-provided JDBC connection or, alternatively, you can let the SessionFactory to open the connection for you.

2.2.1 User provided JDBC connection

This design choice frees the application to obtain JDBC connections wherever it pleases, but the application must be careful not to open two connection sessions on the same connection.

Code sample 2 - User provided JDBC connection

```
java.sql.Connection conn=datasource.getConnection();
Session sess=sessions.openSession(conn);
Transaction tx=sess.beginTransaction(); // start a new transaction (optional)
```

Please observe the optional line (the last line in the above code sample). The application is able to choose to manage transactions by directly manipulation JTA or JDBC transactions. If you are using a Hibernate Transaction then you client code will be abstracted away from the underlying implementation.

2.2.2 Hibernate provided JDBC connection

If you choose to have the SessionFactory open connections, you must know that the SessionFactory must be provided with connection properties in one of the following ways:

- Pass an instance of java.util. Properties to Configuration.setProperties().
- Place hibernate.properties in a root directory of the classpath (probably the most used way).
- Set System properties using java –Dproperty=value.
- Include <property> elements in hibernate.cfg.xml.

If you take this approach, opening a Session is as simple as:

Code sample 3 - Hibernate provided JDBC connection

```
        Session sess=sessions.openSession(); //obtain JDBC connection and
instantiate a
        // new Session, start a new transaction (optional)
        Transaction tx=sess.beginTransaction();
```

2.3 Configuring hibernate.properties

In our case we are interested in working in IBM DB2 database, so what we'll have to do in the configuration file is let Hibernate know some of our database properties. You need to know that Hibernate is reading this reading that information from a file called hibernate.properties, file which must appear in your application classpath.

The first four parameters are very familiar to any one who ever wrote code to retrieve a JDBC connection. The hibernate.dialect property tells hibernate that we are using an IBM DB2 'dialect'. By setting up this dialect Hibernate enables several DB2-specific features by default, so that you don't have to set them up manually.

Table 2 - Configuring hibernate.properties

```
hibernate.connection.driver_class=COM.ibm.db2.jdbc.app.DB2Driver
hibernate.connection.url=jdbc:db2:test
hibernate.connection.username=omega
hibernate.connection.password=omega
hibernate.dialect=cirrus.hibernate.sql.DB2Dialect
```

2.4 Logging in Hibernate

Hibernate logs various events using Apache commons-logging. The commons-logging service will direct output to either log4j (you must include log4j.jar and log4j.properties in your classpath) or JDK 1.4 logging (if running under JDK 1.4 or above).

3 Sample application

Now that we have seen a little bit of Hibernate features, we'll proceed with a simple sample application where we'll have three classes: an abstract class (AbstractItem) and two classes (Item and EItem) that inherit the abstract class. In this application we'll use a DB2 database.

In the picture below you can see the class diagram for this simple application.

3.1 Configuring hibernate.properties for our application

In section **2.3 Configuring hibernate.properties** from this paper, we have presented an example of configuring hibernate.properties. That example presents exactly the configuration that we use in our application. Please take a look at that section to see exactly the configuration.

3.2 Application class diagram

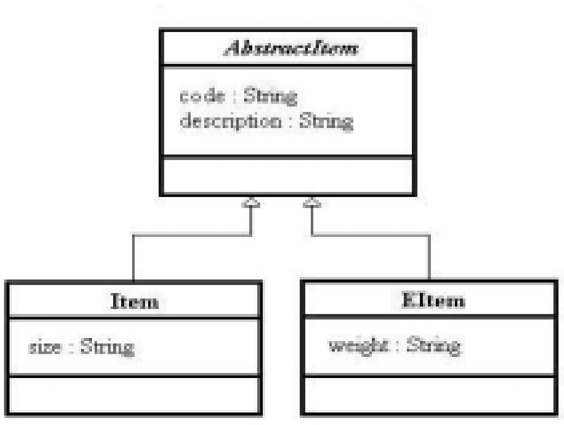

Figure 4 - Application class diagram

Below you can see the class diagram for our simple application. In the diagram you can observe the abstract class AbstractItem and the two classes that inherit it Item and EItem.

3.3 Create database schema

Now that we have configured the property file for Hibernate, we'll have to create the database and the tables we need. In order to do that we'll use the following scripts:

Code sample 4 - Database scripts for TEST databse (the databse used in our application)

```
create db test
connect to test user omega using omega
create table Abstract_Item(
  Id bigint NOT NULL PRIMARY KEY,
  Code varchar(16),
  Description varchar(60),
  Subclass char(1));
create table Hi_Value (
Next_Value bigint);
insert into Hi_Value (Next_Value) values (1);
```

3.4 Create JavaBeans to be mapped

In this section we'll present the JavaBeans corresponding to the class diagram presented in section 3.2 Application Class Diagram.

Code sample 5 - Create net.test.AbstractItem JavaBean

```
package net.test;
public class AbstractItem{
   private String code;
   private String description;
   void setCode(String code){ // setter for code;
     this.code=code;
   }
   String getCode(){ // getter for code;
     return code;
   }
   void setDescription(String description){ // setter for description;
     this.description=description;
   }
   String getDescription(){ // getter for description;
     return description;
   }
}
```

Code sample 6 - Create net.test.Item JavaBean

```
package net.test;
public class Item extends AbstractItem{
   private String size;
   void setSize(String size){ // setter for size;
     this.size=size;
   }
   String getSize(){// getter for size;
     return size;
   }
}
```

Code sample 7 - Create net.test.EItem JavaBean

```
package net.test;
public class EItem extends AbstractItem{
        String weight;
        void setWeight(String weight){ // setter for weight;
                this.weight=weight;
        }
        String getWeight(){ // setter for weight;
```

```
                    return weight;
            }
        }
```

You must know that it is mandatory that all of your getters and setters must exist, but their visibility doesn't matter.

If you wanted to maintain immutable objects, then you could set the state of your object during the construction and all the getters and setters should be private. You would have to provide a default constructor in addition to any other constructors you created; but, the default constructor can have private visibility as well. The reason the setter and getter methods and the default constructor must exist is because Hibernate abides by the JavaBean syntax and uses these method signatures in order to persist data during O/R mapping.

3.5 Create XML mapping

Code sample 8 - O/R Mapping file for our application

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="net.test.AbstractItem" table="ABSTRACT_ITEM"
discriminator-value="A">
    <id name="id" type="long" column="id">
      <generator class="hilo">
        <param name="table">hi_value</param>
        <param name="column">next_value</param>
        <param name="max_lo">1</param>
      </generator>
    </id>
    <discriminator column="Subclass" type="character"/>
    <property name="code" column="Code"/>
    <property name="description" column="Description"/>
    <subclass name="net.test.Item" discriminator-value="I">
    </subclass>
  </class>
</hibernate-mapping>
<!-- parsed in 0ms -->
```

Here we'll present the O/R mapping file and then we'll explain it. Another interesting thing to see in this section is represented by the problems that appeared when we first tried to use a generator for our Primary Key.

The document type definition file defined in the DOCTYPE exits at the specified URL, but it also exists in the hibernate.jar file so you don't have to worry about importing it manually. This DTD file is used to define the valid tags that may be used in the XML O/R mapping file.

<hibernate-mapping>	This tag is the base tag in the XML file and it has two optional attributes, but we will not need them for this simple example application.
<class>	This tag represents a persistent Java class. The name attribute of this tag refers to the fully qualified, dot delimited, class name of the Java class that we are mapping. The table attribute refers to the database table that our class maps to (in our case, the ABSTRACT_ITEM table). Because the Hibernator didn't generated this for us, we'll have to add this by hand (please see the next section of this paper).
<id>	The <id> element specifies which field acts as the primary key for the database table, and to specify how the primary key is generated.
<generator>	This element is used to specify how the primary key (<id>) will be generated.
<discriminator>	The <discriminator> element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. A restricted set of types may be used: string, character, integer, byte, short, boolean, yes_no, true_false. Actual values of the discriminator column are specified by the discriminator-value attribute of the <class> and <subclass> elements.
<property>	In conformity with the Hibernate documentation, "the <property> element declares a persistent, JavaBean style property of the class". This is used mainly for instance variables that are primitives or String, and in our case the code of the abstract item would be represented by a <property> element.
<subclass>	Finally, polymorphic persistence requires the declaration of each subclass of the root persistent class. For the (recommended) table-per-class-hierarchy mapping strategy, the <subclass> declaration is used. Each subclass should declare its own persistent properties and subclasses. <version> and <id> properties are assumed to be inherited from the root class. Each subclass in a heirarchy must define a unique discriminator-value. If none is specified, the fully qualified Java class name is used.

3.6 Primary Key Generator

The required <generator> child (for <id>) element names a Java class used to generate unique identifiers for instances of the persistent class. If any parameters are required to configure or initialize the generator instance, they are passed using the <param> element.

Code sample 9 - Primary key generator with hilo algorithms

```
<id name="id" type="long" column="id">
 <generator class="hilo">
 <param name="table">hi_value</param>
 <param name="column">next_value</param>
 <param name="max_lo">1</param>
 </generator>
</id>
```

All generators implement the interface net.sf.hibernate.id.IdentifierGenerator. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, Hibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

increment	Generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. Do not use in a cluster.
identity	Supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.
sequence	Uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int.
hilo	Uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. Do not use this generator with connections enlisted with JTA or with a user-supplied connection.
seqhilo	Uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid.hex	Uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.
uuid. string	Uses the same UUID algorithm. The UUID is encoded a string of length 16 consisting of (any) ASCII characters. Do not use with PostgreSQL.
native	Picks identity, sequence or hilo depending upon the capabilities of the underlying database.

assigned	Lets the application to assign an identifier to the object before save() is called.
foreign	Uses the identifier of another associated object. Used in conjunction with a <one-to-one> association.

3.7 Create data sources and sessions

The XML O/R mappings files must be loaded into an object representation so that Hibernate can use them. All we'll have to do is to create an instance of net.sf.hibernate.cfg.Configuration class and then tell to the Configuration instance to store the mapping information for a given class by calling the addClass method and providing it with the given class's Class object. Please observe that the addClass method knows to use the fully qualified name class name in order to look in the same package for a corresponding .hbm.xml mapping file.

Once we have a Configuration object we'll use it in order to create a SessionFactory that will be responsible for creating Session objects. Please notice that in the Hibernate documentation a session is defined as "a single-threaded, short-lived object representing a conversation between the application and the persistent store". A session wraps a JDBC connection, acts like a factory for Transaction objects, manages persistent objects in the application, can span several transactions but it doesn't necessary represent an atomic unit of work like a transaction does.

Code sample 10 - Create datasource and session

```
private static SessionFactory sessionFactory;
static {
  try {
     Configuration ds=new Configuration();
     ds.addClass(Item.class);
     ds.addClass(EItem.class);
     sessionFactory=ds.buildSessionFactory();
  }
  catch (Exception e) {
     throw new RuntimeException("can't get connection");
  }
}
```

Let's create a static initializer which will be responsible for creating a SessionFactory object. This static initializer will load once when the class is first referenced and we will no longer need to reload our Item and EItem class mappings.

In case that in your application you already have an infrastructure of connection management, you can provide a connection to openSession (Connection con) method, and Hibernate will use the connection you provide.

3.8 Manipulating database objects

In this section you will see how to write to the database, how to load objects from the database and how to update and query the database.

Writing to database

Code sample 11 - Writing to database

```
public void testCreateItemEItem() throws Exception {
   session=sessionFactory.openSession();
   myItem=new Item();
   myItem.setCode("itemCode01");
   myItem.setDescription("itemDescription01");
   myItem.setSize("itemSize01");
   myEItem=new EItem();
   myEItem.setCode("eitemCode01");
   myEItem.setDescription("eitemDescription01");
   myEItem.setWeight("eitemWeight01");
   session.save(myItem);
   session.save(myEItem);
   session.flush();
   session.connection().commit();
   session.close();
}
```

In order to write to the database, the first thing you have to do is to open a new Session. This will be made using a SessionFactory object. We will create an object (that we wish to persist) and save it in the session. Now all you will have to do is to flush the session, call commit on the connection and close the session.

By flushing the session, you will force Hibernate to synchronize the in-memory data with the database. Hibernate will perform a flush periodically, but because you can not be sure when this kind of flush will be made, it is recommended to explicitly flush the in-memory data into the database to make sure that it is written immediately. Another thing that you also must be sure is to commit the database connection before closing the session.

Loading an object from the database

Code sample 12 - Loading an object from the database

```
public void testLoadItemEItem(String itemID, String eitemID)
throws Exception {
   Session session=sessionFactory.openSession();
   Item loadedItem=(Item) session.load(Item.class, itemID);
   EItem loadedEItem=(EItem) session.load(EItem.class, eitemID);
   session.close();
}
```

By loading an object from the database we mean the process of bringing an object back into memory by using its identifier. Please observe that this is different than querying for an object. For more details please take a look to section 13 (Querying the database) in this document

In order to load an object from the database we will need a session and we also need the primary key of the object we wish to load. If we start from our previous example and we want to load our EItem back into an object, we will call the session.load method with the Class object representing EItem and our primary key of '211'.

Updating the database (same session and different session)

You are able to update an object either in the same session you have created the object or in an entirely different session. If you want to update an object in the same session is trivial, because you just modify the object's state. On the other hand, if you want to update an object in another session, then you will have to load or query first for that object and only then update it.

Code sample 13 - Updating the database in different sessions

```
        public void testUpdateItemNewSession() throws Exception {
          session=sessionFactory.openSession();
          session.save(item);
          session.flush();
          session.connection().commit();
          session.close();
          Session session2=sessionFactory.openSession();
          Item itemFromSession2=(Item) session2.load(Item.class, new
  String(itemCode));
          String newSize="someSize";
          itemFromSession2.setSize(newSize);
          session2.flush();
          session2.connection().commit();
          session2.close();
          Session session3=sessionFactory.openSession();
          Item itemFromSession3=(Item) session3.load(Item.class, new
  String(itemCode));
          session3.connection().commit();
          session3.close();
          assertEquals(newSize, itemFromSession3.getSize());
        }
```

Deleting persistent objects

Code sample 14 - Deleting persistent object

```
session.delete(myItem);
```

By using Session.delete() you will remove an object's state from the database. Of course, your application might still hold a reference to it, so it's best to think to delete() as making a persistent instance transient.

You may also want to delete many objects at once by passing a Hibernate query string to delete().

You may now delete objects in any order you like, without risk of foreign key constraint violations. Of course, it is still possible to violate a NOT NULL constraint on a foreign key column by deleting objects in the wrong order.

Querying the database

Code sample 15 - Querying the database

```
public void testQuery() throws Exception {
   session=sessionFactory.openSession();
   session.save(item);
   List list=session.find("from item in class net.test.Item where
item.size="+item.getSize()+ "");
   Item actualItem=(Item) list.get(0);
   assertEquals(item, actualItem);
   session.flush();
   session.connection().commit();
   session.close();
}
```

You may query the database if a few different ways, but here we'll present only a simple example of querying the database. The easiest way to query a database is to use the session.find method. You must provide a query for session.find, using Hibernate's object-oriented query language. The example you will se below is a fairly simple one.

References

Hibernate – Homepage, http://www.hibernate.org/, http://forum.hibernate.org/
Hibernate - Developer Mailing List, https://lists.sourceforge.net/lists/listinfo/hibernate-devel
Using Hibernate to Persist Your Java Objects to IBM DB2 Universal Database, by Javid Jamae and Kulvir Singh Bhogal , http://www-106.ibm.com/developerworks/db2/library/techarticle/0306bhogal/0306bhogal.html
Keel Framework - Developer Mailing List, https://keel.dev.java.net/

Authors:
Ciprian-Ioan Ileană, Java Developer, Schartner Innovations SRL România
Paul-Mihai Bălşan, Java Developer, Schartner Innovations SRL România