25.07.2003

Hibernate & Tomcat Quickstart

Christian Bauer
<c.bauer@unix.net>

1. Object/relational persistence

Working with object-oriented software and a relational database can be cumbersome and time consuming in todays enterprise environments. Hibernate is an object/relational mapper for the Java environment. The term of object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational structure. Hibernate not only takes care of the mapping from Java classes to database tables, but also provides advanced data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernates goal is to relieve the developer from 95 percent of common data persistence related programming tasks.

2. Getting started with Hibernate

This article discusses a quick setup of Hibernate 2.0with the Apache Tomcat servlet container (using Tomcat 4.1.24) for a web-based application. Hibernate works well in a managed environment with all major J2EE application servers too. The database in the examples is PostgreSQL 7.3, but this can be easily changed to any of the other 16 Hibernate supported databases.

The first step is to copy all required libraries to the Tomcat installation. We use a separate web context (`webapps/quickstart`) for this tutorial, so we've to consider both the global library search path (`TOMCAT/common/lib`) and the classloader at the context level in `webapps/quickstart/WEB-INF/lib` (for JAR files) and `webapps/quickstart/WEB-INF/classes`. We refer to both classloader levels as the global classpath and the context classpath.

1. First, copy the JDBC driver for the database to the global classpath. This is required for the DBCP connection pool software that comes bundled with Tomcat, which we use. In our specific case, copy the `pg73jdbc3.jar` library (for PostgreSQL 7.3 and JDK 1.4) to the global classloaders path. If you'd like to use a different database, simply use its appropriate JDBC driver.

2. Hibernate is packaged as a JAR library. The `hibernate2.jar` file is placed in the context classpath along with other classes of the application. Hibernate requires some 3rd party libraries at runtime, they come bundled with the Hibernate distribution in the `lib` directory; see Table 1. Copy the required 3rd party libraries to the context classpath.

3. Configure both Tomcat and Hibernate for a database connection.

Table 1. Hibernate 3rd party libraries

Library	Description
dom4j (required)	Hibernate uses dom4j to parse XML configuration and XML mapping metadata files.
CGLIB (required)	Hibernate uses the code generation library to enhance classes at runtime (in combination with Java reflection).
Commons Beanutils, Commons Collections, Commons Lang, Commons Logging (required)	Hibernate uses the various utility libraries from the Apache Jakarta Commons project.
ODMG4 (required)	Hibernate provides an optional ODMG compliant persistence manager interface. It is required if you like to map collections, even if you don't intend to use the ODMG API.
Log4j (optional)	Hibernate uses the Commons Logging API, which in turn can use Log4j as the logging mechanism. If the Log4j library is placed in the context library directory, Commons Logging will use Log4j and its `log4j.properties` in the context classpath. An example properties file for log4j is delivered with the Hibernate distribution.

After all libraries have been placed, a resource declaration for the database JDBC connection pool has to be added to Tomcats main configuration file, `TOMCAT/conf/servlet.xml`:

```
<Context path="quickstart" docBase="/quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- DBCP database connection settings -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
        <parameter>
            <name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- DBCP connection pooling options -->
        <parameter>
            <name>maxWait</name>
            <value>5000</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>2</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>4</value>
        </parameter>

    </ResourceParams>
</Context>
```

Wit this configuration, Tomcat will use the DBCP connection pool and provide pooled JDBC `Connections` through JNDI at `java:comp/env/jdbc/quickstart`.

The next step is to configure Hibernate to use connections from a datasource factory located in JNDI. We will use Hibernates XML based configuration instead of the basic configuration via property files, but keep in mind that both are equivalent in features We only use the XML configuration because it is usually more convenient. The XML configuration file is placed in the context classpath (`WEB-INF/classes`), usually named `hibernate.cfg.xml`:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Mapping files -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

We turn logging of SQL commands off and tell Hibernate what database SQL dialect should be used. The latter is a required setting, because databases differ in their interpretation of the SQL "standard". Hibernate will take care of the differences and comes bundled with dialects for all major commercial and open source databases.

A `SessionFactory` is Hibernates concept of a single datastore, multiple databases can be used by declaring multiple `<session-factory>` elements.

3. First persistent class

Take a look at the last element of the `hibernate.cfg.xml`: `Cat.hbm.xml` is the name of a Hibernate XML mapping file for the persistent class `Cat`, located in the context classpath. This file declares the mapping from `Cat` to database tables. We'll come back to that file soon.

Hibernate facilitates the Plain Old Java Object (POJO, sometimes called Plain Ordinary Java Object) approach for persistent classes. A POJO is much like a Java Bean, with properties of the class accessible via getter and setter methods:

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }
    public String getId() {
        return oid;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Hibernate is not restricted in its usage of property types, all Java JDK types (like `String`, `char` and `float`) can be mapped, including classes from the Java collections framework, which we won't cover in this tutorial.

No special interface has to be implemented for persistent classes nor do we have to subclass from a special root persistent class. Hibernate also doesn't use any compile time processing like byte-code manipulation, it relies solely on Java reflection and runtime class enhancement (through CGLIB) to detect object state changes.

4. Mapping the cat

The `cat.hbm.xml` mapping file contains the metadata required for the object/relational mapping.

The metadata includes declaration of persistent classes, their properties and which database tables and columns are used to hold objects of these classes:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="name">
```

```
            <column name="NAME" sql-type="varchar(16)" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

Every persistent class has to have an identifer attribute (actually, only classes representing first class objects, not dependent value objects, which are mapped as components of a first class object). This property is used to distinguish persistent objects: Two cats are equal if `catA.getId().equals(catB.getId())` is true, this concept is called *database identity*. Hibernate comes bundled with various identifer generators for different scenarios (including native generators for database sequences and hi/lo identifier patterns). We use the UUID generator and also specify the column `CAT_ID` of the table `CAT` for the generated identifier value (as a primary key of the table).

All other properties of `Cat` are mapped to the same table. In the case of the `name` property, we mapped it with an explicit database column declaration. This is especially useful when the database schema (with SQL DDL statements) is automatically generated from the mapping declaration with Hibernates *SchemaExport* tool. All other properties are mapped using Hibernates default settings. The table `CAT` in the database looks like this:

```
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 cat_id | character(32)         | not null
 name   | character varying(16) | not null
 sex    | character(1)          |
 weight | real                  |
Indexes: cat_pkey primary key btree (cat_id)
```

5. Playing with cats

We're now ready to start Hibernates `Session`, storing and retriving `Cats` from the database. But first, we've to get a `Session` (Hibernates persistence manager) from the `SessionFactory`:

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

A `SessionFactory` is responsible for one database and may only use one XML configuration file (`hibernate.cfg.xml`). The `SessionFactory` has to be only created once and can be stored somewhere (e.g. in a global registry) to be retrieved by each thread that requires persistence operations.

A `Session` is a non-threadsafe object that represents a single unit-of-work with the database. `Sessions` are opened by a `SessionFactory` and are closed when all work is completed:

```
session = sessionFactory.openSession();
transaction = session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
transaction.commit();
session.close();
```

Each unit-of-work has to take place in the scope of a transaction, we use Hibernates `Transaction` API to abstract from the underlying transaction strategy (in our case, JDBC transactions). This allows our application to be deployed with container managed transactions (using JTA) without any change in the source code, if so desired.

Hibernate has various methods that can be used to retrieve objects from the database. The most flexible way is using the Hibernate Query Language (HQL), which is an easy to learn and powerful object-oriented extension to SQL:

```
Query query = session.createQuery("select cat from Cat as cat where cat.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}
```

Hibernate also offers an object-oriented *query by criteria* API that can be used to formulate type-safe queries. Hibernate of course uses `PreparedStatements` and parameter binding for all SQL communication with the database.

6. Finally

We only scratched the surface of Hibernate in this small tutorial. The Hibernate reference documentation has detailed information about all Hibernate features and options:

- Advanced configuration options for J2EE environments, including distributed transactions and object caching.
- ORM with Hibernate, including mapping datatypes, collections, components, one-to-one/one-to-many/many-to-many associations, custom user-defined types and callback interfaces.
- Detailed information about transaction and session handling with disconnected sessions, long transactions (versioning) and pessimistic locking.
- Lots of useful mapping examples (Java classes as UML diagrams), best practices , tips & tricks.