Chapter 1. Quickstart with Tomcat

1.1. Getting started with Hibernate

This tutorial explains a setup of Hibernate 3.0 with the Apache Tomcat servlet container (we used version 4.1, the differences to 5.0 should be minimal) for a web-based application. Hibernate works well in a managed environment with all major J2EE application servers, or even in standalone Java applications. The database system used in this tutorial is PostgreSQL 7.4, support for other database is only a matter of changing the Hibernate SQL dialect configuration and connection properties.

First, we have 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.

Now, copy the libraries to the two classpaths:

1. Copy the JDBC driver for the database to the global classpath. This is required for the DBCP connection pool software which comes bundled with Tomcat. Hibernate uses JDBC connections to execute SQL on the database, so you either have to provide pooled JDBC connections or configure Hibernate to use one of the directly supported pools (C3P0, Proxool). For this tutorial, copy the `pg74jdbc3.jar` library (for PostgreSQL 7.4 and JDK 1.4) to the global classloaders path. If you'd like to use a different database, simply copy its appropriate JDBC driver.

2. Never copy anything else into the global classloader path in Tomcat, or you will get problems with various tools, including Log4j, commons-logging and others. Always use the context classpath for each web application, that is, copy libraries to `WEB-INF/lib` and your own classes and configuration/property files to `WEB-INF/classes`. Both directories are in the context level classpath by default.

3. Hibernate is packaged as a JAR library. The `hibernate3.jar` file should be copied in the context classpath together with other classes of the application. Hibernate requires some 3rd party libraries at runtime, these come bundled with the Hibernate distribution in the `lib/` directory; see Table 1.1, â Hibernate 3rd party libraries â. Copy the required 3rd party libraries to the context classpath.

Table 1.1. Hibernate 3rd party libraries

Library	Description
antlr (required)	Hibernate uses ANTLR to produce query parsers, this library is also needed at runtime.
dom4j (required)	Hibernate uses dom4j to parse XML configuration and XML mapping metadata files.
CGLIB, asm (required)	Hibernate uses the code generation library to enhance classes at runtime (in combination with Java reflection).
Commons Collections, Commons Logging (required)	Hibernate uses various utility libraries from the Apache Jakarta Commons project.
EHCache (required)	Hibernate can use various cache providers for the second-level cache. EHCache is the default cache provider if not changed in the configuration.
Log4j (optional)	Hibernate uses the Commons Logging API, which in turn can use Log4j as the underlying logging mechanism. If the Log4j library is available in the context library directory, Commons Logging will use Log4j and the `log4j.properties` configuration in the context classpath. An example properties file for Log4j is bundled with the Hibernate distribution. So, copy log4j.jar and the configuration file (from `src/`) to your context classpath if you want to see whats going on behind the scenes.
Required or not?	Have a look at the file `lib/README.txt` in the Hibernate distribution. This is an up-to-date list of 3rd party libraries distributed with Hibernate. You will find all required and optional libraries listed there (note that "buildtime required" here means for Hibernate's build, not your application).

We now set up the database connection pooling and sharing in both Tomcat and Hibernate. This means Tomcat will provide pooled JDBC connections (using its builtin DBCP pooling feature), Hibernate requests these connections through JNDI. Alternatively, you can let Hibernate manage the connection pool. Tomcat binds its connection pool to JNDI; we add a resource declaration to Tomcat's main configuration file, `TOMCAT/conf/server.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>3000</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

The context we configure in this example is named `quickstart`, its base is the `TOMCAT/webapp/quickstart` directory. To access any servlets, call the path `http://localhost:8080/quickstart` in your browser (of course, adding the name of the servlet as mapped in your `web.xml`). You may also go ahead and create a simple servlet now that has an empty `process()` method.

Tomcat provides connections now through JNDI at `java:comp/env/jdbc/quickstart`. If you have trouble getting the connection pool running, refer to the Tomcat documentation. If you get JDBC driver exception messages, try to setup JDBC connection pool without Hibernate first. Tomcat & JDBC tutorials are available on the Web.

Your next step is to configure Hibernate. Hibernate has to know how it should obtain JDBC connections. We use Hibernate's XML-based configuration. The other approach, using a properties file, is almost equivalent but misses a few features the XML syntax allows. The XML configuration file is placed in the context classpath (`WEB-INF/classes`), as `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-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">org.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 is used and where to get the JDBC connections (by declaring the JNDI address of the Tomcat bound pool). The dialect is a required setting, 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 Hibernate's concept of a single datastore, multiple databases can be used by creating multiple XML configuration files and creating multiple `Configuration` and `SessionFactory` objects in your application.

The last element of the `hibernate.cfg.xml` declares `Cat.hbm.xml` as the name of a Hibernate XML mapping file for the persistent class `Cat`. This file contains the metadata for the mapping of the POJO class `Cat` to a database table (or tables). We'll come back to that file soon. Let's write the POJO class first and then declare the mapping metadata for it.

1.2. First persistent class

Hibernate works best with the Plain Old Java Objects (POJOs, sometimes called Plain Ordinary Java Objects) programming model for persistent classes. A POJO is much like a JavaBean, with properties of the class accessible via getter and setter methods, shielding the internal representation from the publicly visible interface (Hibernate can also access fields directly, if needed):

```
package org.hibernate.examples.quickstart;

public class Cat {

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

    public Cat() {
    }

    public String getId() {
        return id;
    }

    private 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 and primitives (like `String`, `char` and `Date`) can be mapped, including classes from the Java collections framework. You can map them as values, collections of values, or associations to other entities. The `id` is a special property that represents the database identifer (primary key) of that class, it is highly recommended for entities like a `Cat`. Hibernate can use identifiers only internally, but we would lose some of the flexibility in our application architecture.

No special interface has to be implemented for persistent classes nor do you have to subclass from a special root persistent class. Hibernate also doesn't require any build time processing, such as byte-code manipulation, it relies solely on Java reflection and runtime class enhancement (through CGLIB). So, without any dependency of the POJO class on Hibernate, we can map it to a database table.

1.3. 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 and the mapping of properties (to columns and foreign key relationships to other entities) to database tables.

```
<?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>

    <class name="org.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" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>
```

```
    </class>

</hibernate-mapping>
```

Every persistent class should have an identifer attribute (actually, only classes representing entities, not dependent value-typed classes, which are mapped as components of an entity). 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, hi/lo identifier tables, and application assigned identifiers). We use the UUID generator (only recommended for testing, as integer surrogate keys generated by the database should be prefered) and also specify the column `CAT_ID` of the table `CAT` for the Hibernate 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 is automatically generated (as SQL DDL statements) from the mapping declaration with Hibernate's *SchemaExport* tool. All other properties are mapped using Hibernate's default settings, which is what you need most of the time. 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)
```

You should now create this table in your database manually, and later read Chapter 21, *Toolset Guide* if you want to automate this step with the `hbm2ddl` tool. This tool can create a full SQL DDL, including table definition, custom column type constraints, unique constraints and indexes.

1.4. Playing with cats

We're now ready to start Hibernate's `Session`. It is the *persistence manager*, we use it to store and retrieve `Cat`s to and from the database. But first, we've to get a `Session` (Hibernate's unit-of-work) from the `SessionFactory`:

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

The call to `configure()` loads the `hibernate.cfg.xml` configuration file and initializes the `Configuration` instance. You can set other properties (and even change the mapping metadata) by accessing the `Configuration` *before* you build the `SessionFactory` (it is immutable). Where do we create the `SessionFactory` and how can we access it in our application?

A `SessionFactory` is usually only build once, e.g. at startup with a *load-on-startup* servlet. This also means you should not keep it in an instance variable in your servlets, but in some other location. Furthermore, we need some kind of *Singleton*, so we can access the `SessionFactory` easily in application code. The approach shown next solves both problems: startup configuration and easy access to a `SessionFactory`.

We implement a `HibernateUtil` helper class:

```
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() {
        Session s = (Session) session.get();
        if (s != null)
            s.close();
        session.set(null);
    }
}
```

This class does not only take care of the `SessionFactory` with its static initializer, but also has a `ThreadLocal` variable which holds the `Session` for the current thread. Make sure you understand the Java concept of a thread-local variable before you try to use this helper. A more complex and powerful `HibernateUtil` class can be found in `CaveatEmptor`, http://caveatemptor.hibernate.org/

A `SessionFactory` is threadsafe, many threads can access it concurrently and request `Session`s. A `Session` is a non-threadsafe object that represents a single unit-of-work with the database. `Session`s are opened from a `SessionFactory` and are closed when all work is completed. An example in your servlet's `process()` method might look like this (sans exception handling):

```
Session session = HibernateUtil.currentSession();
Transaction tx = session.beginTransaction();

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

session.save(princess);

tx.commit();
HibernateUtil.closeSession();
```

In a `Session`, every database operation occurs inside a transaction that isolates the database operations (even read-only operations). We use Hibernates `Transaction` API to abstract from the underlying transaction strategy (in our case, JDBC transactions). This allows our code to be deployed with container-managed transactions (using JTA) without any changes.

Note that you may call `HibernateUtil.currentSession();` as many times as you like, you will always get the current `Session` of this thread. You have to make sure the `Session` is closed after your unit-of-work completes, either in your servlet code or in a servlet filter before the HTTP response is send. The nice side effect of the second option is easy lazy initialization: the `Session` is still open when the view is rendered, so Hibernate can load unitialized objects while you navigate the current object graph.

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:

```
Transaction tx = session.beginTransaction();

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

tx.commit();
```

Hibernate also offers an object-oriented *query by criteria* API that can be used to formulate type-safe queries. Hibernate of course uses `PreparedStatement`s and parameter binding for all SQL communication with the database. You may also use Hibernate's direct SQL query feature or get a plain JDBC connection from a `Session` in rare cases.

1.5. Finally

We only scratched the surface of Hibernate in this small tutorial. Please note that we don't include any servlet specific code in our examples. You have to create a servlet yourself and insert the Hibernate code as you see fit.

Keep in mind that Hibernate, as a data access layer, is tightly integrated into your application. Usually, all other layers depent on the persistence mechanism. Make sure you understand the implications of this design.

For a more complex application example, see http://caveatemptor.hibernate.org/ and have a look at other tutorials linked on http://www.hibernate.org/Documentation