

Selected Topics from

Java Persistence with Hibernate



Ken Kousen

Version 3.2

118. Java Persistence with Hibernate (Selected Topics)

Version 3.2

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

© 2006 Ken Kousen. All rights reserved.
Published in the United States.

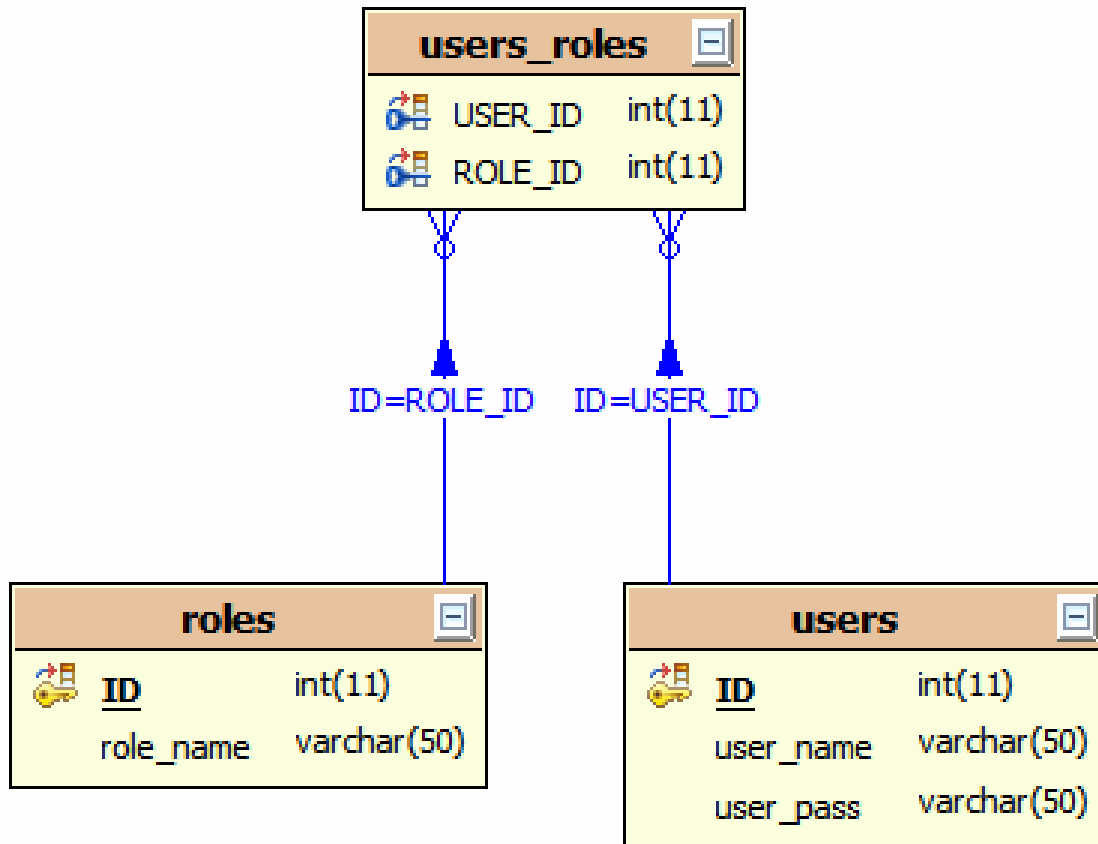
This book is printed on 100% recycled paper.

The Users-Roles Schema

- As a simple example, consider a typical authentication system for a web site.
 - The site uses a simple username/password combination to authenticate users.
- Security is applied in terms of standard roles.
 - Users are assigned to roles.
 - Roles determine which pages a user may access.
- A typical database schema modeling users and roles is given below.
- The schema consists of three tables:
 - **ROLES**, which has an **ID** column of type **int** and a **role_name** column of type **varchar**.
 - **USERS**, which has an **ID** column of type **int**, a **user_name** column of type **varchar** and a **user_pass** column of type **varchar**.
 - **USERS_ROLES**, a linking table with two columns, **user_id** and **role_id**, which are foreign keys to the other two tables.

The Users-Roles Schema

- An entity/relationship diagram for this model is given below.



- Contrast this with the likely object-oriented model of the same information:



A JDBC Query

EXAMPLE

- This example will show how to access all the rows in the ROLES table, convert them to instances of the Role class, and print them. The code can be found in **Examples/UsersRoles/JDBC**.
- Connecting to the database using JDBC involves
 1. Loading the appropriate database driver.
 2. Opening a connection using a JDBC URL.
 3. Creating a **java.sql.Statement**, or its subinterfaces **PreparedStatement** or **CallableStatement**.
 4. Using the **Statement** to pass SQL to the database.
 5. Processing the returned **java.sql.ResultSet** (if the SQL was a query) or checking the update count (if the SQL was an **INSERT**, **UPDATE**, or **DELETE**).
 6. Closing everything.
- Nearly everything in JDBC can throw a **SQLException**, so the code must be wrapped in all the needed **try/catch** blocks.

A JDBC Query

EXAMPLE

```
import java.sql.*;
import cc.db.beans.Role;

public class PrintRolesJDBC {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost/authdb",
                "admin",
                "password");
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
                "SELECT * FROM ROLES");
            while (rs.next()) {
                Role role = new Role();
                role.setId(rs.getInt("id"));
                role.setName(rs.getString("role_name"));
                System.out.println(role);
            }
            rs.close();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

A JDBC Query

EXAMPLE

- Test the application now with the following commands:

```
ant
run cc.db.jdbc.PrintRolesJDBC
(1,user)
(2,admin)
```

- Immediate issues related to this code:
 - The name of the driver class, the JDBC URL, the username and the password are all **hard-wired** into the code.
 - A **try/catch** block is required for loading the driver, in case it isn't found in the classpath of the application.
- By moving to a **DataSource**, the driver, URL, username and password can all be removed, but this requires an application server or some other mechanism that supports JNDI.
- Alternative, light-weight approaches exist (such as the Spring framework), but they, too, require properties files, datasources, and additional configuration.
- In the **while** loop that processes the **ResultSet**,
 - The value of each column in the **ResultSet** is extracted using the proper “get” method (here, **getInt** and **getString**)
 - The data for each row is then assigned to a property of the **Role** class using the corresponding “set” methods, **setId** and **setName**
 - Printing the object is done by putting the Role reference inside **System.out.println**. This invokes the **toString** method on the **Role** object, so this method should be overridden appropriately

A JDBC Query

EXAMPLE

- Since database connections are considered scarce resources, all connected elements must be closed as soon as possible.
 - **ResultSet** is closed in the **try** block here.
 - **Statement** and **Connection** are closed in the **finally** block to guarantee that they are closed whether an exception is thrown or not.
 - The calls to **close** in the **finally** block must be guarded by an **if** statement to ensure that a **NullPointerException** isn't thrown accidentally.
 - Since even the **close** method of both **Statement** and **Connection** can throw a **SQLException**, they too require **try/catch** blocks.
- Exception handling considerably increases the amount of code required.
 - Only the stack trace of any exceptions is being printed here.
 - In a real system, logging would likely be done and the SQL exceptions might be rethrown as application exceptions.
 - For debugging purposes it would probably be better to close the **Statement** and **Connection** in separate **try/catch** blocks.
- Note also that object/relational mapping is being done manually here by extracting data from the **ROLES** table and assigning it to properties of the **Role** class.

Using Hibernate

EXAMPLE

- The code in **Examples/UsersRoles/Hibernate** illustrates how Hibernate can be used to solve the same problem.
- Hibernate reduces the required Java coding considerably, but the trade-off is the number of XML files that must be defined.
 - An overall Hibernate configuration file is created for each database.
 - Individual mapping files need to be defined for each class.
- The Hibernate configuration file consists of a series of keys and values, which either form a properties file or are specified in XML. Some defined values include:
 - **Dialect** (specified for each database vendor)
 - **URL, driver class, username, password**
 - Other helpful properties, like **show_sql**
 - Location of mapping files for individual classes

Using Hibernate

EXAMPLE

- Here is the XML Hibernate configuration file, **hibernate.cfg.xml**:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration ... details below >

<hibernate-configuration>
  <session-factory>
    <property name="dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="connection.url">
      jdbc:mysql://localhost/authdb
    </property>
    <property name="connection.username">
      admin
    </property>
    <property name="connection.password">
      password
    </property>
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.HashtableCacheProvider
    </property>
    <property name="show_sql">true</property>
    <mapping resource =
      "com/capcourse/db/Role.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Using Hibernate

EXAMPLE

- The above file looks more complicated than it is. The Hibernate configuration consists of a series of properties, each of which is identified as an attribute of the `<property>` element.
- This configuration file holds all the connection details, such as username, password, database driver, and URL. It also specifies the Hibernate **dialect**, which is used to identify the particular flavor of SQL needed for this vendor.
 - A list of supported dialects is contained in the **hibernate.properties** file, which is discussed in the next chapter.
- The mapping file also allows other interesting properties to be specified, such as **show_sql**. This property tells Hibernate to output the generated SQL code to the console.
- Finally, the file includes a `<mapping>` tag which specifies the location of the specific XML mapping file for the Role class.
- The XML file requires a **DOCTYPE** declaration which is as follows:

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

- The DOCTYPE declares
 - The root element of the XML file, `<hibernate-configuration>`
 - Information about the owner of the language and its version
 - The URL where the DTD could be downloaded, if necessary

Using Hibernate

EXAMPLE

- The mapping file for the Role class specifies:
 - The Java class being mapped, as well as its package
 - The database table associated with this class
 - The primary key column for the table
 - Specific property mappings (the **String** data type here is inferred)
- The mapping file for the Role class is as follows:

```
<?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="cc.db.beans">
  <class name="Role" table="roles"
    catalog="authdb">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" column="role_name" />
  </class>
</hibernate-mapping>
```

Using Hibernate

EXAMPLE

- The mapping file has a different DOCTYPE declaration, meaning that the mapping files are from a different XML vocabulary than the configuration file.
 - Its presence is required by Hibernate.
- The mapping file tells Hibernate what it needs to know in order to manage persistence for the role type:
 - There is only one class, **cc.db.beans.Role**.
 - There is only one table, **ROLES**.
 - The **ROLES** table is part of the **authdb** database, here identified by the catalog attribute.
 - The **<id>** element states that the primary key value is generated by the database when rows are inserted. The various options for this are discussed in a later chapter.
 - The single **<property>** element maps the **name** attribute of the **Role** class to the **role_name** column of the **ROLES** table. Since no data type is identified, Hibernate assumes that the same data type is used for both the class and the table. By reflection, this is identified as a **java.lang.String**.

Using Hibernate

EXAMPLE

- With these files in the application classpath, the Java code becomes:

```
import java.util.*
import org.hibernate.*;
import org.hibernate.cfg.Configuration;

import cc.db.beans.Role;

public class PrintRoles {

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        SessionFactory factory = new Configuration()
            .configure().buildSessionFactory();
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        Criteria crit =
            session.createCriteria(Role.class);
        List<Role> roles = crit.list();
        tx.commit();
        for (Role r : roles) {
            System.out.println(r);
        }
        session.close();
        factory.close();
    }
}
```

Using Hibernate

EXAMPLE

- The application uses a Hibernate **SessionFactory** class to represent all the connection information.
 - The **SessionFactory** reads the configuration file, generates any required connections, and pools them.
 - Normally the **SessionFactory** instance is created only once in a given application and cached.
- The **Session** class represents a single interaction with the database (to be discussed in depth later).
- The **Transaction** class represents a database transaction.
- The **Criteria** interface is used as a simplified API for retrieving instances and will be discussed in its own chapter.
 - Its **createCriteria** method selects all rows mapped to the class.
 - The **list** method retrieves the populated instances of the **Role** class objects as a **java.util.List** collection.
- The contents of the list are already **Role** objects. Hibernate automatically populates **Role** instances for each row of the table and adds them to the list.
 - While Hibernate does not yet use **Java-5.0 generics**, the code supports their use.
 - The list method returns an unchecked **java.util.List**, which is stored in a **List<Role>**.
 - The Java-5 attribute **@SuppressWarnings** is then added to the method signature.

Using Hibernate

EXAMPLE

- Build and test; the output is the same as from the JDBC version:

```
ant
run cc.db.hibernate.PrintRoles
(1,user)
(2,admin)
```

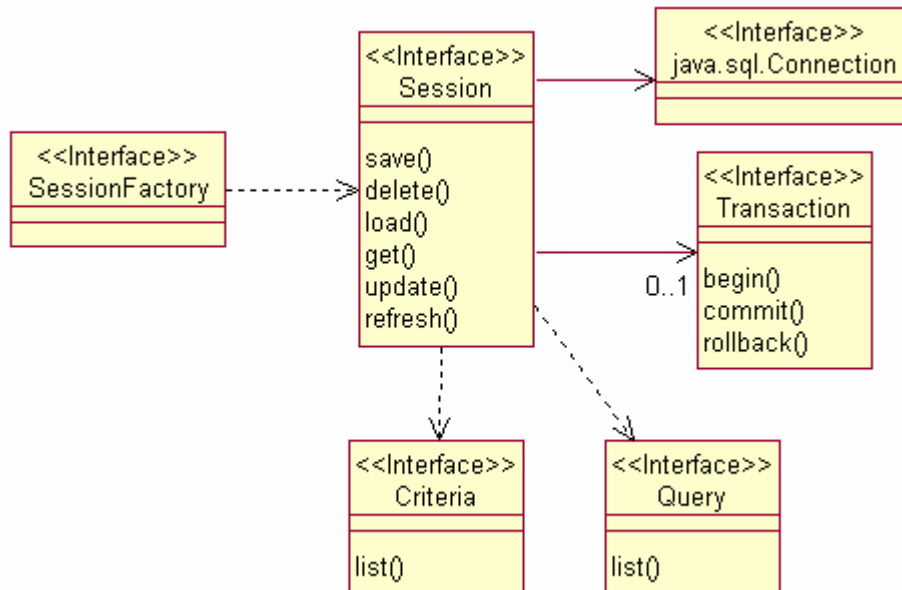
- Here, the application is only reading the database.
- If modifications are made and something goes wrong, the transaction would need to be rolled back.
- To handle that, the code would be wrapped in a **try/catch** block that catches **HibernateException**, which is a runtime (i.e., unchecked) exception.
 - The transaction would be committed at the end of the **try** block, and if necessary rolled back in the **catch** block.
 - The session would be closed in a **finally** block.
- Hibernate reduces the required coding compared to JDBC.
- The differences become more dramatic as the number of classes and tables increases.

The Hibernate Architecture

- The Hibernate API includes about 1000 classes and interfaces.
- According to the JavaDoc documentation, these classes are divided into the **Core API**, the **Extension API**, a **Miscellaneous API**, an **Internal Implementation**, and so-called **Other Packages**.
- Fortunately, a developer intending to use Hibernate only needs to learn a relatively small number of them in order to make progress; the following interfaces are all from **org.hibernate**:
 - **SessionFactory**
 - **Session**
 - **Transaction**
 - **Query**
 - **Criteria**
 - **ScrollableResults**
- For backward compatibility with versions prior to 3.0, the 3.2 distribution includes the **org.hibernate.classic** package, now deprecated.
- Finally, the **org.hibernate.usertype** package includes interfaces for user-defined types.

Interface Model

- The top-level interfaces used by most Hibernate code are shown in the following UML diagram:



- Every Hibernate client will use a **SessionFactory** to create one or more **Sessions**, which wrap JDBC **Connections** and can associate Hibernate’s own **Transaction** abstraction.
- Not shown is **org.hibernate.cfg.Configuration**, used in the previous chapter for programmatic configuration.
- We’ll see that there are a number of “second-level” interfaces involved with **Criteria** and HQL queries.

SessionFactory

- All Hibernate programs start by acquiring a **SessionFactory** instance.
- Earlier examples illustrated that the way to build a **SessionFactory** differs depending on whether one uses a **hibernate.properties** file or a **hibernate.cfg.xml** file.

– When using the **hibernate.properties** file, the code is:

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

– When using an XML configuration file, the process is slightly different:

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

- Building a **SessionFactory** is an expensive operation.
 - It involves processing all the input properties files – both the basic configuration file and the individual mapping files.
 - A **SessionFactory** should therefore be instantiated only once and cached for later access.
- The Hibernate tutorial built into the documentation in the distribution shows a typical idiom, which is to use a utility class to store the **SessionFactory** and acquire **Session** instances. We'll examine that idea next.

A Hibernate Utility Class

EXAMPLE

- Here is the class **HibernateUtil** from the tutorial. This file is in the Hibernate distribution in the following directory:

```
c:/Capstone/Tools/Hibernate3.2
    /doc/reference/tutorial/src/util
```

- The “util” token means that we’re working with the **util** package.
- The only difference between that file and the listing below is that the code here has been formatted for readability.

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

public class HibernateUtil {

    private static final SessionFactory factory;

    static {
        try {
            factory = new Configuration()
                .configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println(
                "Initial SessionFactory creation failed."
                + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory()
    {
        return factory;
    }
}
```

A Hibernate Utility Class

EXAMPLE

- The **SessionFactory** is created in a **static initialization block**, which ensures that it is available as soon as the utility class is loaded.
- The factory itself is stored in a **static** attribute of the class.
- This class assumes that the default XML configuration file **hibernate.cfg.xml** is being used; this is the most common option.
- If creation fails, the exception is printed and rethrown.
- This example assumes that the application is self-contained.
 - If an application server is available, the **SessionFactory** instance can be stored in a JNDI registry under a particular look-up string.
 - The utility class would then do the JNDI look-up, but would otherwise be the same.

SessionFactory Methods

- The Hibernate API documentation shows 25 different methods in the **SessionFactory** class, counting all overloads.
- Nevertheless, the vast majority of programs use only the following methods:
 - The **openSession** method, which creates a database connection and instantiates an internal **Session** on it.

```
public Session openSession()  
    throws HibernateException
```

- The **getCurrentSession** method, which retrieves the current session, if some form of session context management has been enabled (to be discussed later).

```
public Session getCurrentSession()  
    throws HibernateException
```

- The **close** method, which destroys the **SessionFactory** and releases all resources associated with it.

```
public void close() throws HibernateException
```

- Recall that **HibernateException** is a runtime (i.e. unchecked) exception and therefore does not require a **try/catch** block.

Hibernate Sessions

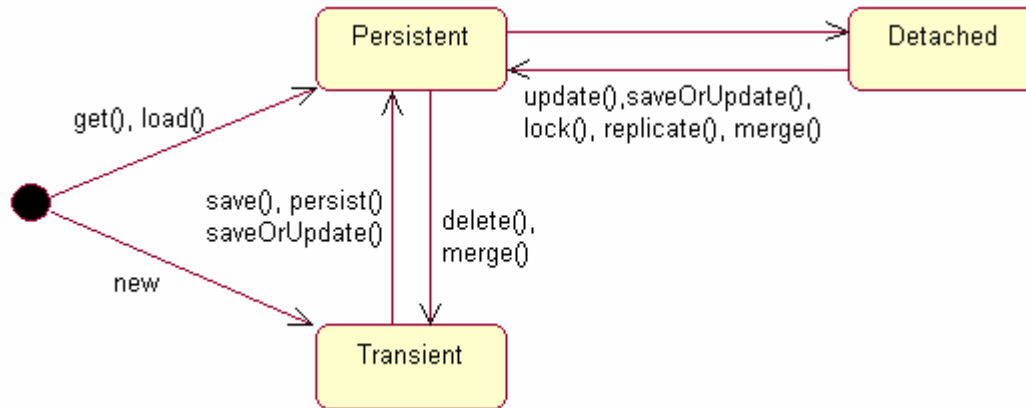
- A Hibernate **Session** is a single-threaded, non-shared object that represents a particular unit of work with the database.
 - It also has the API methods, like **save**, **delete**, **update** and **get**, used to interact with the database.
- Each interaction with the database takes place in a session.
- The main function of the **Session** is to manage persistence for **entity** classes.
 - An **entity** is an object associated with a unique row in the database, identified by a primary key.
 - This is in contrast to a **value object**, which is dependent on an entity.
 - The distinction will be discussed further in the next chapter.
- The Session class therefore contains the so-called **CRUD** methods: **Create**, **Retrieve**, **Update**, and **Delete**.
 - **Create** is done with **save**.
 - **Retrieve** is done with **get**, or the various other querying APIs.
 - **Update** is done with **update**.
 - **Delete** is done with **delete**.
- Later we'll consider the **Data Access Object** design pattern as a way to encapsulate all interactions with the persistent store.

Object States

- Java objects are populated in **Sessions**, but an object's lifecycle may extend beyond that of the session.
- Any Hibernate-mapped object may exist in one of three states:
 - **Transient**, implying the object is not associated with any **Session**
 - **Persistent**, meaning associated with a unique **Session**
 - **Detached**, which implies that the object was previously persistent but no longer associated with any **Session**
- Detached objects replace the so-called **Value Object** or **Data Transfer Object (DTO)** design pattern often used in J2EE web applications.
 - An instance is populated from the database but is no longer connected to it. The instance is then used wherever necessary in the middleware or view layer.
 - Since it is unconnected, there is a risk the data will become outdated (stale). In many cases, this is considered better than leaving open database connections longer than strictly necessary.

State Transitions

- An object may move from one of these three states to the other, as follows:



- From the JavaDoc API for the **Session** class:
 - “Transient instances may be made persistent by calling **save**, **persist**, or **saveOrUpdate**. Persistent instances may be made transient by calling **delete**. Any instance returned by a **get** or **load** method is persistent. Detached instances may be made persistent by calling **update**, **saveOrUpdate**, **lock**, or **replicate**. The state of a transient or detached instance may also be made persistent as a new persistent instance by calling **merge**.”

