Object-relation mapping without the container

Develop a transactional persistence layer using Hibernate and Spring

Level: Intermediate

Richard Hightower (rhightower@arc-mind.com)
Developer, ArcMind Inc.
13 Apr 2004

Just when you think you've got your developer tools all sorted out, a fresh crop is sure to emerge. In this article, regular *developerWorks* contributor Rick Hightower uses a real-world example to introduce you to two of the most exciting new technologies for the enterprise. Hibernate is an object-relation mapping tool and Spring is an AOP framework and IOC container. Follow along as Rick shows you how to combine the two to build a transactional persistence tier for your enterprise applications.

Related content:
CMP-CMR tutorial series
AOP banishes the tight-coupling blues
Incremental development with Ant and JUnit

Subscriptions:
dW newsletters
dW Subscription (CDs and downloads)

If you follow the latest developer buzz then you've likely heard of IOC (Inversion of Control) containers and AOP (aspect-oriented programming). Like many developers, however, you may not see where these technologies fit into your development efforts. In this article, I'll begin to remedy that, with a hands-on introduction to using Hibernate and Spring to build a transactional persistence tier for your enterprise applications.

Hibernate is a popular, easy-to-use, open source object-relation (OR) mapping framework for the Java platform. Spring is an AOP framework and IOC container. Together, these two technologies will provide the foundation of your development efforts in this article. You'll use Hibernate to map some persistent objects to a relational database and Spring to make Hibernate easier to use and provide declarative transaction support. As an added bonus, I'll throw in a little TDD (test-driven development), as DbUnit was used to write the test code for the example classes.

Note that this article assumes that you are familiar with enterprise development on the Java platform, including JDBC, OR mapping issues, J2EE design patterns like DAO, and declarative transaction support such as that provided by Enterprise JavaBeans (EJB) technology. You are not expected to be an expert in any of these technologies in order to follow the discussion, nor do you need to be familiar with AOP, IOC, or TDD, as all three will be introduced in the article.

I'll start with an introduction to the two development technologies and then launch into the example.

Introducing Hibernate

Hibernate is a full-featured, open source OR mapping framework for the Java platform. In many ways Hibernate is similar to EJB CMP CMR (container-managed-persistence/container-managed-relationships), and JDO (Java Data Objects). Unlike JDO, Hibernate focuses entirely on OR mapping for relational databases, and includes more features than most commercial products. Most EJB CMP CMR solutions use code generation to implement persistence code, while JDO uses bytecode decoration. Conversely, Hibernate uses reflection and runtime bytecode generation, making it nearly transparent to end users. (Earlier implementations of Hibernate used reflection only, which aids in debugging, and current versions retain this option.)

Hibernate allows you to model inheritance (several ways); association (one-to-one or one-to-many, containment, and aggregation); and composition. I'll cover several examples of each type of relationship in this article.

Hibernate provides a *query language* called Hibernate Query Language (HQL), which is similar to JDO's JDOQL and EJB's EJB QL; although it is closer to the former. But Hibernate doesn't stop there: it also allows you to perform direct SQL queries and/or use *object criteria* to compose criteria easily at runtime. I'll use only HQL in the examples for this article.

Unlike EJB CMP CMR and like JDO, Hibernate can work inside of or outside of a J2EE container, which is a boon for those of us doing TDD and agile development.

Introducing Spring

The first time AOP expert Nicholas Lesiecki explained AOP to me, I didn't understand a word he was saying; and I felt much the same the first time I considered the possibility of using IOC containers. The conceptual basis of each technology alone is a lot to digest, and the myriad of new acronyms applied to each one doesn't help -- particularly given that many of them are variations on stuff we already use.

Like many technologies, these two are much easier to understand in practice than in theory. Having done my own research on AOP and IOC container implementations (namely, XWork, PicoContainer, and Spring), I've found that these technologies help me gain functionality without adding code-based dependencies on multiple frameworks. They'll both be a part of my development projects going forward.

Porting Hibernate-based apps
If your application must run on many RDBMS systems, Hibernate-based applications port almost effortlessly from IBM DB2, MySQL, PostgreSQL, Sybase, Oracle, HypersonicSQL, and many more. I even recently worked on an application port from MySQL to Firebird, which isn't all that well supported by Hibernate, and the port was effortless. See Resources for a case study of a switch between Postgres and MySQL.

In a nutshell, AOP allows developers to create non-behavioral concerns, called crosscutting concerns, and insert them in their application code. With AOP, common services like logging, persistence, transactions, and the like can be factored into aspects and applied to domain objects without complicating the object model of the domain objects.

IOC allows me to create an application context where I can construct objects, and then pass to those objects their collaborating objects. As the word *inversion* implies, IOC is like JNDI turned inside out. Instead of using a tangle of abstract factories, service locators, singletons, and straight construction, each object is constructed with its collaborating objects. Thus, the container manages the collaborators.

Spring is both an AOP framework and an IOC container. I believe it was Grady Booch who said the great thing about objects is that they can be replaced; and the great thing about Spring is that it helps you replace them. With Spring, you simply inject dependencies (collaborating objects) using JavaBeans properties and configuration files. Then it's easy enough to switch out collaborating objects with a similar interface when you need to.

Spring provides an excellent on-ramp to both IOC containers and AOP. As such, you don't need to be familiar with AOP in order to follow the examples in this article. All you need to know is that you'll be using AOP to declaratively add transactional support to your example application, much the same way that you would use EJB technology. See Resources to learn more about IOC containers, AOP, and Spring.

About DbUnit
Developing with a new framework without unit testing is like walking on a new trapeze wire without a net: sure you could do it, but you're going to bleed. I prefer to develop with a net, and for me that net is TDD. Before DbUnit came along, testing code that was dependent on a database could be a little tough. DbUnit is an extension of JUnit that provides a framework for unit tests dependent on a database. I used DbUnit to write the test code for the example classes in this article. While not present in the article, the DbUnit code is part of the article source code (see Resources). Or for an introduction to DbUnit, see "Control your test-environment with DbUnit and Anthill" (*developerWorks*, April 2004) by Philippe Girolami.

Down to business

For the remainder of the article, all of the discussion will be based on a working example. The starting point is an enterprise application for which you are implementing a transactional persistence layer. Your persistence layer, an object-relational database, includes familiar abstractions like `User`, `User Group`, `Roles`, and `ContactInfo`.

Before I can delve into the essentials of the database -- queries and transaction management -- I need to lay its foundation: object-relation mapping. I'll set this up using Hibernate, and just a touch of Spring.

OR mapping with Hibernate

Hibernate uses XML (**.hbm.xml*) files to map Java classes to tables and JavaBeans properties to database tables. Fortunately, a set of `XDoclet` tags support Hibernate development, which makes it easier to create the **.hbm.xml* files you need. The code in Listing 1 maps a Java class to a database table. See Resources to learn more about `XDoclet` tags.

Listing 1. Mapping a Java class to a DB table

```
[User.java]
/**
 * @hibernate.class table="TBL_USER"
 * ..
 * ..
 * ...
 */
public class User {

        private Long id = new Long(-1);
        private String email;
        private String password;

        .
        .
        .

        /**
         * @return
         * @hibernate.id column="PK_USER_ID"
         *               unsaved-value="-1"
         *               generator-class="native"
         */
        public Long getId() {
                return id;
        }

        ...


        /**
         * @hibernate.property column="VC_EMAIL"
         *                     type="string"
         *                     update="false"
         *                     insert="true"
         *                     unique="true"
         *                     not-null="true"
         *                     length="82"
         * @return
         */
        public String getEmail() {
                return email;
        }

        /**
         * @hibernate.property column="VC_PASSWORD"
         *                     type="string"
         *                     update="false"
         *                     insert="true"
         *                     unique="true"
         *                     not-null="true"
         *                     length="20"
         * @return
         */
        public String getPassword() {
                return password;
        }

        ...
        ...
```

```
        ...
}
```

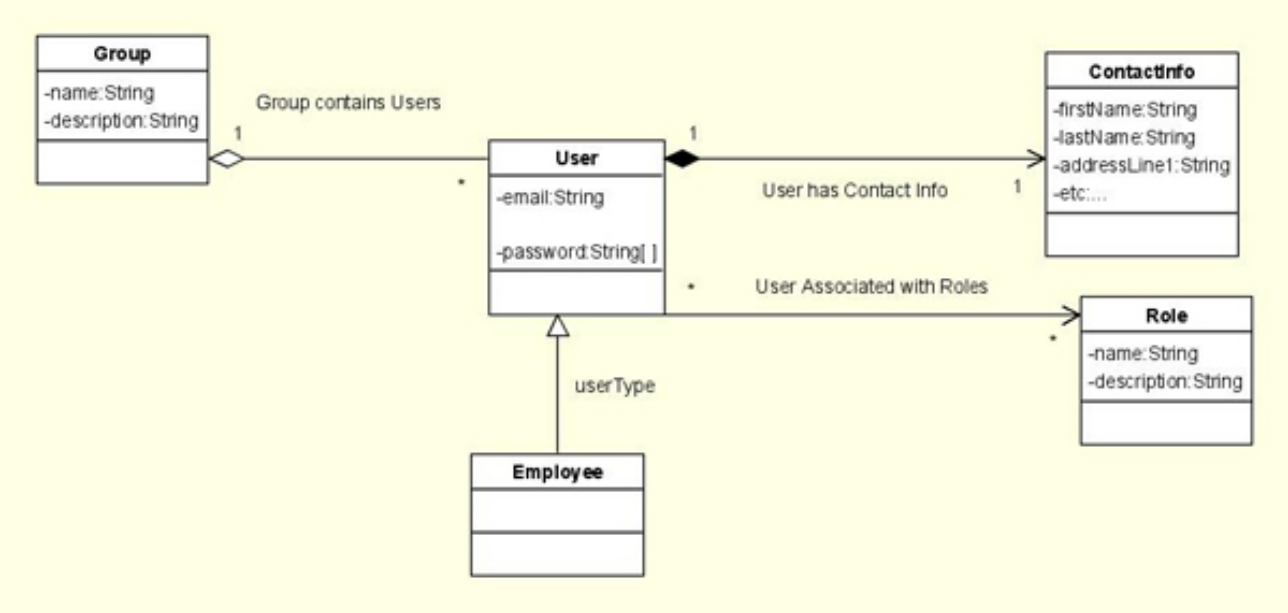
As you can see, the `@hibernate.class table="TBL_USER"` tag maps `User` to the `TBL_USER` table. The `@hibernate.property column="VC_PASSWORD"` maps the *password* JavaBeans property to the `VC_PASSWORD` column. The `@hibernate.id column="PK_USER_ID"` tag states that the *id* property is the primary key, and it will uses the native (`generator-class="native"`) database mechanism for generating keys (for examplei, Oracle sequences and SQL Server Identity keys). Hibernate allows you to specify every conceivable strategy for getting primary keys other than `generator-class="native"`, although I prefer to go native. The *type* and *length* attributes are for generating tables from the Hibernate **.hbm.xml* OR mapping files. These final attributes are optional, since you might not be using a green-field database. In the case of this example, the database already existed; so you don't need the extra attributes. (A *green-field application* is a new application and a *green-field database* is a new database for a new application. It isn't often that you get to work on a fresh application; although it's nice once in a while, isn't it?)

Now that you've seen how tables are mapped to classes and columns to JavaBeans properties, you'll use Hibernate to set up some relationships in your OR database.

Setting up object relations

In this section I'll just scratch the surface of the options Hibernate provides for setting up relationships between objects. Let's start by setting up relationships between classes such as `User`, `User Group`, `Roles`, and `ContactInfo`. Some of these relationships are shown in Figure 1, an authentication object model for your database.

Figure 1. Graphical representation of relationships



As you can see, a full gamut of relationships exists between the above abstractions. `User` has a one-to-one relationship with `ContactInfo`. The lifecycle of `ContactInfo` is the same as the `User` (composition in UML *aka* cascade delete in database speak). If the `User` were deleted so would be the corresponding `ContactInfo`. A many-to-many relationship exists between `Users` and `Roles` (that is, association with independent lifecycles). A a one-to-many relationship exists between `Groups` and `Users`, since a group has many users. Users can exists outside of a group; that is, aggregation not composition (in database speak no cascade delete relationship exists between `Groups` and `Users`). In addition, `User` and `Employee` have a subclass relationship; that is, an `Employee` is a type of `User`. Table 1 shows how to create several different types of object relationships using `XDoclet` tags.

Table 1. Creating object relationships using XDoclets

Relationship	Java/XDoclet	SQL DDL (MySQL generated by Hibernate Schema Export)

<table>
<tr>
<td>Group contains User

One-to-many

Aggregation

Bidirectional (Group<-->Users)</td>
<td>[Group.java]
/**
*
* @return
*
* @hibernate.bag name="users"
* cascade="save-update"
* lazy="true"
* inverse="true"
*
* @hibernate.collection-key
* column="FK_GROUP_ID"
*
* @hibernate.collection-one-to-many
* class="net.sf.hibernateExamples.User"
*/
public List getUsers() {
return users;
}

[User.java]
/**
* @hibernate.many-to-one
* column="FK_GROUP_ID"
* class="net.sf.hibernateExamples.Group"
*/
public Group getGroup() {
return group;
}</td>
<td>create table TBL_USER (
PK_USER_ID BIGINT NOT NULL
AUTO_INCREMENT,
USER_TYPE VARCHAR(255) not null,
FK_GROUP_ID BIGINT,
VC_EMAIL VARCHAR(82) not null
unique,
primary key (PK_USER_ID)
)

create table TBL_GROUP (
PK_GROUP_ID BIGINT NOT NULL
AUTO_INCREMENT,
VC_DESCRIPTION VARCHAR(255),
VC_NAME VARCHAR(40) unique,
primary key (PK_GROUP_ID)
)

alter table TBL_USER add index
(FK_GROUP_ID),
add constraint FK_111 foreign
key (FK_GROUP_ID)
references TBL_GROUP
(PK_GROUP_ID)</td>
</tr>
<tr>
<td>User has contact info

One-to-one

Composition

Unidirectional (User-->ContactInfo)</td>
<td>[User.java]
/**
* @return
*
* @hibernate.one-to-one cascade="all"
*
*/
public ContactInfo getContactInfo() {
return contactInfo;
}

[ContactInfo.java]
(Nothing to see here. Unidirectional!)</td>
<td>create table TBL_USER (
PK_USER_ID BIGINT NOT NULL
AUTO_INCREMENT,
USER_TYPE VARCHAR(255) not null,
FK_GROUP_ID BIGINT,
VC_EMAIL VARCHAR(82) not null
unique,
primary key (PK_USER_ID)
)

create table TBL_CONTACT_INFO (
PK_CONTACT_INFO_ID BIGINT not
null,
...
...
...
primary key (PK_CONTACT_INFO_ID)
)</td>
</tr>
</table>

User associated with roles Many-to-many Association Unidirectional (Users-->Roles)	**[User.java]** /** * @return * @hibernate.bag * table="TBL_JOIN_USER_ROLE" * cascade="all" * inverse="true" * * @hibernate.collection-key * column="FK_USER_ID" * * @hibernate.collection-many-to-many * class="net.sf.hibernateExamples.Role" * column="FK_ROLE_ID" * */ public List getRoles() { return roles; } **[Role.java]** Nothing to see here. Unidirectional!	create table TBL_ROLE (PK_ROLE_ID BIGINT NOT NULL AUTO_INCREMENT, VC_DESCRIPTION VARCHAR(200), VC_NAME VARCHAR(20), primary key (PK_ROLE_ID)) create table TBL_USER (PK_USER_ID BIGINT NOT NULL AUTO_INCREMENT, USER_TYPE VARCHAR(255) not null, FK_GROUP_ID BIGINT, VC_EMAIL VARCHAR(82) not null unique, primary key (PK_USER_ID)) create table TBL_JOIN_USER_ROLE (FK_USER_ID BIGINT not null, FK_ROLE_ID BIGINT not null)
Employee is a User Inheritance User ↑ Employee	**[User.java]** /** * @hibernate.class table="TBL_USER" * discriminator-value="2" * @hibernate.discriminator column="USER_TYPE" * ... ... */ public class User { **[Employee.java]** /** * @hibernate.subclass discriminator-value = "1" */ public class Employee extends User{	create table TBL_USER (PK_USER_ID BIGINT NOT NULL AUTO_INCREMENT, USER_TYPE VARCHAR(255) not null, FK_GROUP_ID BIGINT, VC_EMAIL VARCHAR(82) not null unique, primary key (PK_USER_ID))

See Resources to learn more about setting up object relationships in Hibernate.

Queries in Hibernate

Hibernate has three types of queries:

- Criteria, object composition
- SQL
- HQL

You'll work entirely with HQL in the examples that follow. You'll also begin working with Spring in this section, using its AOP-driven HibernateTemplate to simplify working with Hibernate sessions. In this section you will develop a DAO (Data Access Object). See Resources to learn more about DAOs.

Listing 2 demonstrates two methods: a group lookup using an HQL query and a group lookup followed by an action. Note how the Spring HibernateTemplate simplifies session management in the second method.

Listing 2. Using queries

```
import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.Query;
import org.springframework.orm.hibernate.HibernateCallback;
import org.springframework.orm.hibernate.support.HibernateDaoSupport;



/**
 * @author Richard Hightower
 * ArcMind Inc. http://www.arc-mind.com
 */
public class UserDAO extends HibernateDaoSupport{

  .
  .
  .

  /**
        * Demonstrates looking up a group with a HQL query
        * @param email
        * @return
        */
        public Group findGroupByName(String name) {
           return (Group) getHibernateTemplate().find("from Group g where g.name=?",name).get
(0);
        }

        /**
         * Demonstrates looking up a group and forcing it to populate users (relationship was
lazy)
         * @param email
         * @return
         */
        public Group findPopulatedGroupByName(final String name) {
           HibernateCallback callback = new HibernateCallback(){

            public Object doInHibernate(Session session) throws HibernateException,
SQLException {
                        Group group =null;
                        String query = "from Group g where g.name=?";
                        Query queryObject = getHibernateTemplate().createQuery(session, query);
                        queryObject.setParameter(0, name);
                        group = (Group) queryObject.list().get(0);
                        group.getUsers().size();//force load
                        return group;
                        }

                };

                return (Group) getHibernateTemplate().execute(callback);
        }
  .
  .
  .
}
```

You likely noticed that the second method is quite a bit more involved than the first, because it forces the `users` collection to load. Because the relationship between `Group->Users` was set to lazy initialize (that is, `lazy="true"` in Table 2), the group object required an active session to lookup the users. The second method wouldn't have been required had you set the attribute to `lazy="false"` when you were defining the relationship between `Group` and `Users`. In this case, you would probably have used the first method (`findGroupByName`) to do a listing of groups, and the second method (`findPopulatedGroupByName`) to view the group details.

Spring IOC and Hibernate

When you use Spring it is just as easy to work inside or outside of a J2EE container. On a recent project, for example, I ran my persistence unit tests inside of Eclipse using HSQL and local datasources against a Hypersonic SQL database using Hibernate Transaction manager. Then, when I deployed to my J2EE server I switched my persistence layer to use the J2EE datasources (through JNDI), and the JTA transactions, and to use FireBird (an open source version of Interbase). This was accomplished using Spring as the IOC container.

As you'll see in Listing 3, Spring allows dependency injection. Note how the application context file in the listing allows me to configure a `dataSource`. The `dataSource` is passed to a `sessionFactory`, and the `sessionFactory` is passed to your `UserDAO`.

Listing 3. Spring IOC and Hibernate

```
<beans>

        <!-- Datasource that works in any application server
                You could easily use J2EE data source instead if this were
                running inside of a J2EE container.
         -->
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
                <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
                <property name="url"><value>jdbc:mysql://localhost:3306/mysql</value></property>
                <property name="username"><value>root</value></property>
                <property name="password"><value></value></property>
        </bean>

        <!-- Hibernate SessionFactory -->
        <bean id="sessionFactory" class="org.springframework.orm.hibernate.
LocalSessionFactoryBean">
                <property name="dataSource"><ref local="dataSource"/></property>

                <!-- Must references all OR mapping files. -->
                <property name="mappingResources">
                        <list>
                        <value>net/sf/hibernateExamples/User.hbm.xml</value>
                        <value>net/sf/hibernateExamples/Group.hbm.xml</value>
                        <value>net/sf/hibernateExamples/Role.hbm.xml</value>
                        <value>net/sf/hibernateExamples/ContactInfo.hbm.xml</value>
                        </list>
                </property>

                <!-- Set the type of database; changing this one property will port this to
Oracle,
                         MS SQL etc. -->
                <property name="hibernateProperties">
                        <props>
                          <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</
prop>
                        </props>
                </property>
        </bean>

        <!-- Pass the session factory to our UserDAO -->
        <bean id="userDAO" class="net.sf.hibernateExamples.UserDAO">
                <property name="sessionFactory"><ref local="sessionFactory"/></property>
```

```
        </bean>

</beans>
```

With your `UserDAO` set up, your next step is to define and use some more queries to show what is possible. Hibernate allows you to store queries outside of the source code using named queries, as shown in Listing 4.

Listing 4. Named queries

```
[User.java]
/**
 * @author Richard Hightower
 * ArcMind Inc. http://www.arc-mind.com
 * @hibernate.class table="TBL_USER" discriminator-value="2"
 * @hibernate.discriminator column="USER_TYPE"
 *
 * @hibernate.query name="AllUsers" query="from User user order by user.email asc"
 *
 * @hibernate.query name="OverheadStaff"
 * query="from Employee employee join employee.group g where g.name not in ('ENGINEERING','IT')"
 *
 * @hibernate.query name="CriticalStaff"
 * query="from Employee employee join employee.group g where g.name in ('ENGINEERING','IT')"
 *
 * @hibernate.query name="GetUsersInAGroup"
 * query="select user from Group g join g.users user"
 *
 * @hibernate.query name="GetUsersNotInAGroup"
 * query="select user from User user where user.group is null"
 *
 * @hibernate.query name="UsersBySalaryGreaterThan"
 * query="from User user inner join user.contactInfo info where info.salary > ?1"
 *
 * @hibernate.query name="UsersBySalaryBetween"
 * query="from User user join user.contactInfo info where info.salary between ?1 AND ?2"
 *
 * @hibernate.query name="UsersByLastNameLike"
 * query="from User user join user.contactInfo info where info.lastName like ?1"
 *
 * @hibernate.query name="GetEmailsOfUsers"
 * query="select user.email from Group g join g.users as user where g.name = ?1"
 *
 */
public class User {
                        .
                        .
                        .
```

The above code defines several named queries. *Named queries* are queries that are stored in the **.hbm.xml* file. In Listing 5, you can see how to execute a named query.

Listing 5. Using named queries

```
[UserDAO.java]
        /**
         * Demonstrates a query that returns a String.
         */
        public String[] getUserEmailsInGroup(String groupName){
                List emailList =
                getHibernateTemplate().findByNamedQuery("GetEmailsOfUsers");
                return (String [])
                        emailList.toArray(new String[emailList.size()]);
        }


        /**
         * Demonstrates a query that returns a list of Users
         *
         * @return A list of emails of all of the users in the authentication system.
         *
         */
        public List getUsers(){
                return getHibernateTemplate().findByNamedQuery("AllUsers");
        }

      /**
        * Demonstrates passing a single argument to a query.
        *
        * @return A list of UserValue objects.
        *
        */
        public List getUsersBySalary(float salary){
           return getHibernateTemplate()
               .findByNamedQuery("UsersBySalaryGreaterThan",
                    new Float(salary));
        }


        /**
         * Demonstrates passing multiple arguments to a query
         *
         * @return A list of UserValue objects.
         *
         */
        public List getUsersBySalaryRange(float start, float stop){
                return getHibernateTemplate()
                 .findByNamedQuery("UsersBySalaryBetween",
                        new Object[] {new Float(start), new Float(stop)});
        }
```

With queries underway, you can add the final layer to your persistence tier: transaction management using Spring.

Managing transactions with Spring

Spring allows you to manage transactions declaratively. For example the `UserDAO.addUser` method does not currently execute in a single transaction. Thus, every `User` in a group would be inserted in its own transaction, as shown in Listing 6.

Listing 6. Adding a group of users

```
[UserDAO.java]
/**
 * @param group
 */
public void addGroup(Group group) {
        getHibernateTemplate().save(group);

}

[UserDAOTest.java]

public void testAddGroupOfUsers(){
        Group group = new Group();

        for (int index=0; index < 10; index++){
                User user = new User();
                user.setEmail("rick"+index+"@foobar.com" );
                user.setPassword("foobar");
                group.addUser(user);
        }

        group.setName("testGroup");

        userDAO.addGroup(group);
        assertNotNull(group.getId());

        Group group2 = userDAO.findPopulatedGroupByName("testGroup");

        assertEquals("testGroup",group2.getName());
        assertEquals(10, group2.getUsers().size());
        String email = ((User)group2.getUsers().get(0)).getEmail();
        assertEquals("rick0@foobar.com", email);


}
```

The above solution isn't desirable because each `User` would be inserted into the database in its own transaction. If there were a problem, only a subset of the users could be added. If you wanted to preserve ACID properties (that is, make sure it all happens or nothing happens), you could do transaction management programmatically; but it gets ugly pretty quick. Instead, you'll use Spring's AOP support for declarative transactions, as shown in Listing 7.

Listing 7. Managing transactions declaratively

```
[applicationContext.xml]
<!-- Pass the session factory to our UserDAO -->
   <bean id="userDAOTarget" class="net.sf.hibernateExamples.UserDAOImpl">
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
   </bean>

   <bean id="transactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
      <property name="sessionFactory"><ref bean="sessionFactory"/></property>
   </bean>

   <bean id="userDAO"
     class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref local="transactionManager"/></property>
        <property name="target"><ref local="userDAOTarget"/></property>
```

```
        <property name="transactionAttributes">
                <props>
                        <prop key="addGroup">PROPAGATION_REQUIRED</prop>
                </props>
        </property>
    </bean>
```

Notice that in preparation for the code in Listing 7 I refactored `UserDAO` and extracted its interface. The interface is now `UserDAO` and its implementation class is `UserDAOImpl`. The transaction code in Listing 7 then uses the `UserDAO.addGroup()` method with a transaction attribute of `(PROPAGATION_REQUIRED)`. All the users can now be added in a single transaction, assuming the support of the underlying database.

Conclusion

In this article you've learned how to implement a transactional persistence layer using Hibernate and Spring. Hibernate is a leading OR mapping tool and Spring is an AOP framework and IOC container. Together, the two technologies allow developers to write code that is database-vendor agnostic, and that can run inside of a J2EE container or run standalone. While not central to the discussion, DbUnit (an extension to JUnit) was used to build and test all of the code for the examples in this article.

See Resources to learn more about AOP, IOC containers, and test-driven development.

Resources

- Hibernate's creator Gavin King set the bar high when it came to developing the documentation (as well as the framework!) for Hibernate. See the Hibernate User Guide, on the Hibernate home page, to learn more about this exciting OR mapping tool for the Java platform.
- You can learn more about Spring directly from its inventor, with Rod Johnson's "Introducing the Spring framework" (ServerSide.com, October 2003).
- Matt Raible's AppFuse handily demonstrates using Hibernate and Spring on several databases on several J2EE application servers.
- Juergen Hoeller set down the law about Hibernate, AOP, Spring, and transaction management with his tutorial, "Hibernate -- Data Access with a Spring Framework" (July 2003).
- Still hungry for Spring? Visit the Spring home page.
- The DbUnit home page is your first source for learning more about this handy extension to JUnit.
- For an introduction to DbUnit, see "Control your test-environment with DbUnit and Anthill" (*developerWorks*, April 2004) by Philippe Girolami.
- Of course, Andrew Glover's " Effective unit testing with DbUnit" (OnJava.com, January 2004) is another great source.
- Learn more about the conceptual background of DbUnit and its predecessor, JUnit, with Malcolm Davis's " Incremental development with Ant and JUnit" (*developerWorks*, November 2000).
- When it comes to AOP, Nicholas Lesiecki's the one who started it all (at least for me). Read his "Improve modularity with aspect-oriented programming" (*developerWorks*, January 2002) to learn more about this powerful complement to object-oriented programming.
- In "AOP banishes the tight-coupling blues" (*developerWorks*, February 2004), Andrew Glover is back; this time showing you how to use static crosscutting to decouple your enterprise applications.
- Sean Sullivan's "Advanced DAO programming" (*developerWorks*, October 2003) provides a short introduction to the DAO pattern and then cuts directly to some of its more powerful uses in J2EE programming.
- For those just getting started with CMP CMR, Richard Hightower wrote a complete tutorial series on the subject, starting with "Introduction to container-managed persistence and relationships" (*developerWorks*, March 2003).
- For a short tutorial on XDoclets, see "Enhance J2EE component reuse with XDoclet" (*developerWorks*, May 2003) also by Richard Hightower.
- Of course, you might also want to check out the Hibernate XDoclet tags on SourceForge.

- You'll find articles about every aspect of Java programming in the *developerWorks* Java technology zone.
- Visit the Developer Bookstore for a comprehensive listing of technical books, including hundreds of Java-related titles.
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from *developerWorks*.

About the author

Rick works at Arc-Mind Inc., where he focuses on Struts and J2EE mentoring and consulting. Rick is a software developer at heart with 14 years software development experience (seven of them on the Java platform). Rick has contributed to several Java technology books and written articles for the *Java Developer's Journal* and IBM *developerWorks*. Rick recently completed a book on Struts at SourceBeat.