

Spring

A Developer's Notebook™

Bruce A. Tate
Justin Gethland

- user interfaces
- Hibernate
- testing
- transactions
- security

Using Spring with JDO

JDO is the non-EJB standard for persistence in Java applications. In this section, we'll use our favorite JDO implementation, Kodo, to implement transparent persistence for our application. In this section, I'm not going to try to teach you JDO, but I will show you how to use it to provide persistence to our application.

If you've followed JDO for any length of time, it probably brings to mind all sorts of feverish battles with the intensity of the Crusades. Until recently, most people avoided JDO. With the pending release of JDO 2.0 and several solid commercial and open source JDO implementations, this persistence standard seems to be gathering momentum as a stronger player in the ORM space. In fact, my favorite ORM implementation is Solar Metric's Kodo, possibly the leading JDO implementation. It seems to be more robust than the alternatives when it comes to commercial implementations. It's got more flexible mapping support, it's easier to manage, and it's got far richer support for hardcore enterprise persistence. Consider these advantages:

- You can choose between a variety of open source JDO implementations if you're looking for something that's free, or better control of the source code.
- You can choose a commercial implementation, and get support and maintenance for a reasonable price if you don't.
- With the top commercial vendors, you get incredible power and performance, from better management to flexible mapping.
- You get all of this with the protection and security that an open standard provides.

How do I do that?

You'll use JDO to create a persistent model, and then use that model through a façade layer. The application already has a business domain model created. It's not yet persistent, though. You've also got the interface for the façade layer. You need only do the following steps to enable our application for JDO:

1. Make the domain model persistent. Do that through the byte code enhancer.
2. Configure Spring to use Kodo by making a few simple changes in the configuration file.

3. Build a façade that uses a persistent model through the JDO templates.

That's it. Spring will manage the core JDO resources, consisting of the `PersistenceManagerFactory` and the `PersistenceManager`. Think of these special options as the data source and connection for JDO. You can let Spring manage transactions. The three steps above are all that you need.

First, you need to download and install Kodo. Start with the trial version, which you can find at <http://www.solarmetric.com>. You can use Version 3.2.1 for this book. Once again, you have to add the libraries found in `/kodo-jdo-3.2.1/lib` to our `/lib` folder.

To make the model persistent, modify the Ant task to add JDO's byte code enhancement step: add an Ant task to do so, like in Example 5-8.

Example 5-8. `build.xml`

```
<taskdef name="jdoc" classname="kodo.ant.JDOEnhancerTask"/>

<target name="jdo.enhance">
  <jdoc>
    <fileset dir="${src.dir}">
      <include name="**/*.jdo" />
    </fileset>
  </jdoc>
</target>
```

You'll also add path elements to your Ant build file for `kodo-jdo.jar`, `jdo-1.0.1.jar`, and `jakarta-commons-lang-1.0.1.jar`.

Next, build the persistence mapping. The easiest way is through Kodo's wizard. Launch the Workbench (in the `\bin` directory of your Kodo install) and choose `MetaData` → `Create MetaData` from the menu. Conversely, you can use the `metadatatool` and `mappingtool` scripts that can be found in `/kodo-jdo-3.2.1/bin`, which are just launchers for `kodo.jdbc.meta.MappingTool` and `kodo.meta.JDOMetaDataTool`, respectively.

To keep things consistent with other JDO versions, though, you're going to build a mapping from scratch, with XML. Generate a `.jdo` file with our class metadata as well as a `.mapping` file. Both files reside in the `/war/WEB-INF/classes/com/springbook` folder.

Example 5-9 shows the metadata file.

Example 5-9. `package.jdo`

```
<?xml version="1.0" encoding="UTF-8"?>
<jdo>
  <package name="com.springbook">
    <class name="Bike">
```

Example 5-9. package.jdo (continued)

```
<extension vendor-name="kodo" key="detachable" value="true"/>
<field name="reservations">
  <collection element-type="Reservation"/>
  <extension vendor-name="kodo" key="inverse-owner" value="bike"/>
  <extension vendor-name="kodo" key="element-dependent"
    value="true"/>
</field>
</class>
<class name="Customer">
  <extension vendor-name="kodo" key="detachable" value="true"/>
  <field name="reservations">
    <collection element-type="com.springbook.Reservation"/>
    <extension vendor-name="kodo" key="inverse-owner"
      value="customer"/>
    <extension vendor-name="kodo" key="element-dependent"
      value="true"/>
  </field>
</class>
<class name="Reservation">
  <extension vendor-name="kodo" key="detachable" value="true"/>
</class>
</package>
</jdo>
```

Example 5-10 shows the mapping file.

Example 5-10. package.mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
  <package name="com.springbook">
    <class name="Bike">
      <jdbc-class-map type="base" pk-column="BIKEID" table="BIKES"/>
      <field name="bikeId">
        <jdbc-field-map type="value" column="BIKEID"/>
      </field>
      <field name="frame">
        <jdbc-field-map type="value" column="FRAME"/>
      </field>
      <field name="manufacturer">
        <jdbc-field-map type="value" column="MANUFACTURER"/>
      </field>
      <field name="model">
        <jdbc-field-map type="value" column="MODEL"/>
      </field>
      <field name="reservations" default-fetch-group="true">
        <jdbc-field-map type="one-many" ref-column.BIKEID="BIKEID"
          table="RESERVATIONS"/>
      </field>
      <field name="serialNo">
        <jdbc-field-map type="value" column="SERIALNO"/>
      </field>
    </class>
  </package>
</mapping>
```

Example 5-10. package.mapping (continued)

```
<field name="status">
  <jdbc-field-map type="value" column="STATUS"/>
</field>
<field name="weight">
  <jdbc-field-map type="value" column="WEIGHT"/>
</field>
</class>
<class name="Customer">
  <jdbc-class-map type="base" pk-column="CUSTID"
    table="CUSTOMERS"/>
  <field name="custId">
    <jdbc-field-map type="value" column="CUSTID"/>
  </field>
  <field name="firstName">
    <jdbc-field-map type="value" column="FIRSTNAME"/>
  </field>
  <field name="lastName">
    <jdbc-field-map type="value" column="LASTNAME"/>
  </field>
  <field name="reservations" default-fetch-group="true">
    <jdbc-field-map type="one-many" ref-column.CUSTID="CUSTID"
      table="RESERVATIONS"/>
  </field>
</class>
<class name="Reservation">
  <jdbc-class-map type="base" pk-column="RESID"
    table="RESERVATIONS"/>
  <field name="bike">
    <jdbc-field-map type="one-one" column.BIKEID="BIKEID"/>
  </field>
  <field name="customer">
    <jdbc-field-map type="one-one" column.CUSTID="CUSTID"/>
  </field>
  <field name="reservationDate">
    <jdbc-field-map type="value" column="RESDATE"/>
  </field>
  <field name="reservationId">
    <jdbc-field-map type="value" column="RESID"/>
  </field>
</class>
</package>
</mapping>
```

It's almost too easy. There's no persistence in the model itself, and that's why you use OR technologies. Still, you'll need a layer of code to use that persistence model for your application. That's the façade layer. You'll see a series of calls to the template. Specify the JDO query language statements for the finders, and the objects to persist for deletes, updates, and inserts. You've already got an interface, but we still need to implement the façade (Example 5-11).

Example 5-11. KodoRentABike.java

```
public class KodoRentABike extends JdoDaoSupport implements RentABike {
    private String storeName;

    public List getBikes() {
        return (List) getJdoTemplate().find(Bike.class);
    }

    public Bike getBike(String serialNo) {
        Collection c = getJdoTemplate().find(Bike.class,
            "serialNo == '" + serialNo + "'");
        Bike b = null;
        if(c.size() > 0) {
            b = (Bike)c.iterator().next();
        }
        return b;
    }

    public Bike getBike(int bikeId) {
        return (Bike) getJdoTemplate().
            getObjectById(Bike.class, new Long(bikeId));
    }

    public void saveBike(Bike bike) {
        getJdoTemplate().makePersistent(bike);
    }

    public void deleteBike(Bike bike) {
        getJdoTemplate().deletePersistent(bike);
    }

    //etc.
}
```

This is not a full JDO query language query; it's simply a filter. JDO 2.0 will add a convenience query strings, so that you can add a full JDO query as a single string without needing to build a full query.

Finally, you need to set up some configuration to wire it all together. First, Example 5-12 gives the JDO configuration.

Example 5-12. kodo.properties

```
# To evaluate or purchase a license key, visit http://www.solarmetric.com
kodo.LicenseKey: YOUR_LICENSE_KEY_HERE

javax.jdo.PersistenceManagerFactoryClass: kodo.jdbc.runtime.
JDBCPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName: bikestore
javax.jdo.option.ConnectionPassword:
javax.jdo.option.ConnectionURL: jdbc:mysql://localhost/bikestore
javax.jdo.option.Optimistic: true
javax.jdo.option.RetainValues: true
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.RestoreValues: true
```

Notice the detachOnClose. This makes sure that JDO loads anything that's lazy before the connection goes away, so that other parts of your application, like the view, can only access beans that have already been loaded.

Example 5-12. kodo.properties (continued)

```
kodo.Log: DefaultLevel=WARN, Runtime=INFO, Tool=INFO
kodo.PersistenceManagerImpl: DetachOnClose=true
```

The Spring context will need to wire together the JDO persistence manager, the persistence manager factory, the façade, and any services on the façade. That's done in the context (Example 5-13).

Example 5-13. RentABikeApp-servlet.xml

```
<bean id="jdofactory" class="org.springframework.orm.jdo.
LocalPersistenceManagerFactoryBean">
  <property name="configLocation">
    <value>E:\RentABikeApp\war\WEB-INF\kodo.properties</value>
  </property>
</bean>

<bean id="transactionManager" class="org.springframework.orm.jdo.
JdoTransactionManager">
  <property name="persistenceManagerFactory">
    <ref local="jdofactory"/>
  </property>
</bean>

<bean id="rentaBike" class="com.springbook.KodoRentABike">
  <property name="storeName"><value>Bruce's Bikes</value></property>
  <property name="persistenceManagerFactory">
    <ref local="jdofactory"/>
  </property>
</bean>
```

Recall that you've already got a test case that uses the façade, so you can build it and let it rip.

What just happened?

This is a perfect example of the power of Spring. You've radically changed the implementation of your persistence layer, but you haven't affected the rest of the application *at all*. Here's how it works.

Spring first uses dependency injection to resolve all of the dependencies. Loading the context configures JDO with the data source that you provided, and then sets the persistence manager factory in the façade JDO implementation. Then, when you call a method on the façade, Spring gets you a persistence manager and uses it to process the query that you supply. You can look at it this way: Spring provides a generic JDO façade method, called the template. You plug in the details, and give control to Spring.

When you're biking or when you're coding, one of the most important metrics is efficiency. How much work can you do with each turn of the pedals, or each line of code? Consider the JDO version of the application. The most compelling thing about the Spring programming model is the efficiency. To see what I mean, think about what you don't see here:

- You don't have exception management cluttering the lower levels of your application. With Spring's unchecked exceptions, you can get the exception where it's appropriate to do something with it.
- You don't have resource management. Where JDBC has connections, JDO has the persistence manager. Spring configures the persistence manager factory, and manages the persistence manager within the template for you.
- You aren't forced to manage transactions and security in the façade. Spring lets you configure these things easily so that you can strip all of the ugly details out of your façade layer and let it concentrate on using the persistent model.

All of this is done for you in the Spring template, in code that comes with the Spring framework, which you can read for better understanding or debug in a pinch. In short, you can get more leverage with each line of code, much like running your bike in a higher gear. That's the bottom line for all of the most successful frameworks and programming languages.

Using Hibernate with Spring

Hibernate has long been the persistence framework of choice for Spring developers. Although the Spring team continues to improve the integration with other persistence frameworks, Hibernate remains the most widely used persistence framework with Spring. It's fitting that these two lightweight solutions helped each other succeed. They go quite well together. In this example, we'll show how to integrate Spring and Hibernate.

Hibernate is an outstanding persistence framework. It's popular, it's relatively fast, and it's free. It has rich mapping support and a convenient usage model that's made it popular with developers across the world. Hibernate holds up very well in small- and intermediate-size projects. In fact, though it's nonstandard, you could say that behind EJB, Hibernate is the most popular persistence framework in the world.

How do I do that?

Now that you've configured Spring with an ORM, you know the basic flow, with a persistent domain model and a façade. Since Spring ships with Hibernate dependencies, you can just copy them from Spring's */dist* folder to your */lib* folder: *hibernate2.jar*, *aopalliance.jar*, *cglib-full-2.0.2.jar*, *dom4j.jar*, *ehcache-1.1.jar*, and *odmg.jar*.

Since Hibernate uses reflection, there's no byte code enhancement step. All you've got to do to make the model persistent is to create the mappings, and refer to them in the context. Examples 5-14, 5-15, and 5-16 show the mappings.

Example 5-14. Bike.hbm.xml

```
<hibernate-mapping>
  <class name="com.springbook.Bike" table="bikes">
    <id name="bikeId" column="bikeid" type="java.lang.Integer"
      unsaved-value="-1">
      <generator class="native"></generator>
    </id>
    <property name="manufacturer" column="manufacturer" type="string"/>
    <property name="model" column="model" type="string"/>
    <property name="frame" column="frame" type="int"/>
    <property name="serialNo" column="serialno" type="string"/>
    <property name="weight" column="weight" type="java.lang.Double"/>
    <property name="status" column="status" type="string"/>
    <set name="reservations">
      <key column="bikeId"/>
      <one-to-many class="com.springbook.Reservation"/>
    </set>
  </class>
</hibernate-mapping>
```

The Hibernate version of this mapping is a bit less awkward than the JDO counterpart. It has a richer identification generation library.

Example 5-15. Customer.hbm.xml

```
<hibernate-mapping>
  <class name="com.springbook.Customer" table="customers">
    <id name="custId" column="custid" type="java.lang.Integer"
      unsaved-value="-1">
      <generator class="native"></generator>
    </id>
    <property name="firstName" column="firstname" type="string"/>
    <property name="lastName" column="lastname" type="string"/>
    <set name="reservations">
      <key column="custId"/>
      <one-to-many class="com.springbook.Reservation"/>
    </set>
  </class>
</hibernate-mapping>
```

Example 5-16. Reservation.hbm.xml

```
<hibernate-mapping>
  <class name="com.springbook.Reservation" table="reservations">
    <id name="reservationId" column="resid" type="java.lang.Integer"
      unsaved-value="-1">
      <generator class="native"></generator>
    </id>
    <property name="reservationDate" column="resdate" type="date"/>
    <many-to-one name="bike" column="bikeId" class="com.springbook.Bike"
      cascade="none"/>
    <many-to-one name="customer" column="custId"
      class="com.springbook.Customer"
      cascade="none"/>
  </class>
</hibernate-mapping>
```

In the context, you need to configure Hibernate properties, configure the session factory, and plug the session factory into the façade. The transaction strategies with the JDO context, the iBATIS context and the Hibernate context are the same, as they should be. That's part of what dependency injection is doing for you. Example 5-17 shows the changes to the context.

Example 5-17. RentABikeApp-servlet.xml

```
<bean name="rentaBike" class="com.springbook.HibRentABike">
  <property name="storeName"><value>Bruce's Bikes</value></property>
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>

<bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource"><ref local="dataSource"/></property>
  <property name="mappingResources">
    <list>
      <value>com/springbook/Bike.hbm.xml</value>
      <value>com/springbook/Customer.hbm.xml</value>
      <value>com/springbook/Reservation.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.MySQLDialect
      </prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>
```

Hibernate users often put all of the mapping classes into separate files, unlike JDO. You don't have to; it's just the way that Hibernate users typically prefer to manage things.

Since you've already got a façade interface, you need only a Hibernate implementation. You can use templates, just as with JDO. For finders, specify a Hibernate query language (HQL) statement. For updates, all that you need to specify is the new object to be stored. Example 5-18 is the remarkably thin façade.

Example 5-18. HibRentABike.java

```
package com.springbook;

import org.springframework.orm.hibernate.support.HibernateDaoSupport;

import java.util.List;
import java.util.Date;
import java.util.Set;

import net.sf.hibernate.Query;

public class HibRentABike extends HibernateDaoSupport implements RentABike {
    private String name;

    public List getBikes() {
        return getHibernateTemplate().find("from Bike");
    }

    public Bike getBike(String serialNo) {
        Bike b = null;
        List bikes = getHibernateTemplate().
            find("from Bike where serialNo = ?", serialNo);
        if(bikes.size() > 0) {
            b = (Bike)bikes.get(0);
        }
        return b;
    }

    public Bike getBike(int bikeId) {
        return (Bike)getHibernateTemplate().
            load(Bike.class, new Integer(bikeId));
    }

    public void saveBike(Bike bike) {
        getHibernateTemplate().saveOrUpdate(bike);
    }

    public void deleteBike(Bike bike) {
        getHibernateTemplate().delete(bike);
    }

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

Example 5-18. HibRentABike.java (continued)

```
public String getStoreName() {
    return this.name;
}

public List getCustomers() {
    return getHibernateTemplate().find("from Customer");
}

public Customer getCustomer(int custId) {
    return (Customer)getHibernateTemplate().
        load(Customer.class, new Integer(custId));
}

public List getReservations() {
    return getHibernateTemplate().find("from Reservation");
}

public List getReservations(Customer customer) {
    return getHibernateTemplate().
        find("from Reservation where custId = ?", customer.getCustId());
}

public List getReservations(Bike bike) {
    return getHibernateTemplate().
        find("from Reservation where bikeId = ?", bike.getBikeId());
}

public List getReservations(Date date) {
    return getHibernateTemplate().
        find("from Reservation where resdate = ?", date);
}

public Reservation getReservation(int resId) {
    return (Reservation)getHibernateTemplate().
        load(Reservation.class, new Integer(resId));
}
}
```

What just happened?

The Hibernate flow is pretty much the same as the JDO flow. The Spring JDO template represents a set of default DAO methods. All that you need to do is customize them, usually with a HQL query statement, and possibly the values for parameters to any HQL parameterized queries. Then Spring takes control, getting a session, firing the query, and managing any exceptions.

Once again, you see exceptional leverage. Example 5-19 gives the typical Hibernate method that you'd need to write without Spring.

Example 5-19. HibRentABike.java

```
public List getBikesOldWay() throws Exception {
    // Relies on other static code for configuration
    // and generation of SessionFactory. Might look like:

    // Configuration config = new Configuration();
    // config.addClass(Bike.class).addClass(Customer.class).
    //     addClass(Reservation.class);
    // SessionFactory mySessionFactory = Configuration.
    //     buildSessionFactory();

    List bikes = null;
    Session s = null;
    try {
        s = mySessionFactory.openSession();
        bikes = s.find("from Bike");
    }catch (Exception ex) {
        //handle exception gracefully
    }finally {
        s.close();
    }
    return bikes;
}
```

Example 5-20 shows, again, the Spring counterpart.

Example 5-20. HibRentABike.java

```
public List getBikes() {
    return getHibernateTemplate().find("from Bike");
}
```

Strong typing is for weak programmers. Each building block of an application should strive do one thing well, and only once.

What about...

...alternatives to Hibernate? Hibernate is indeed free, fast, effective, and popular. It's been proven under fire, and has excellent performance and flexibility. Most tools have good Hibernate support, and Hibernate supports every database that matters. I still recommend Hibernate for small and intermediate applications.

So far, as an open source framework, the persistence community can't be quick enough to praise Hibernate or condemn the alternatives. Such blind worship leads to cultish religious decision making. There are some things that competitors do better. If your application has some of these characteristics, you might be best served to look elsewhere:

Standards

JDO and JDBC solutions conform to a standard. Although it's open source code, Hibernate does not conform to a standard. You have to trust the motivations of JBoss group and the founders to do the right thing for decades, and for any framework, that's proven to be a dicey proposition so far.

Management

Other solutions are easier to manage. For example, Kodo JDO and TopLink have management consoles that make it easier to manage caching scenarios, and eager or lazy-loading scenarios.

Mapping

Other frameworks have more powerful and flexible mapping support. If you do not control your schema, you may be best served with another solution. Also, if you prefer to map your schema with GUI tools, then something like JDO Genie or Cayenne may work best for you.

In general, using mainstream frameworks may be the right choice in the end, but often you can find a much better fit with a little digging. Hibernate certainly deserves consideration, but there are other good choices out there, too.

Running a Test Case

The test case is easy to run. You've already got it. It's the one that you ran with the façade.

How do I do that?

Since the test case exists, you can run the existing façade test. You'll just have to make sure that you set up your test data correctly, and you can use the application context unchanged. That's the power, and testability, of Spring.

What just happened?

You used the existing test case. That's nice, because you only need to manage the incremental details for the database. In the next chapter, you'll start to dig into services that you can add to your application with Spring's AOP.