Chapter 2

Spring Quick Start Tutorial

Developing Your First Spring Web Application

The following chapter is a sample from SourceBeat's *Spring Live* book by Matt Raible. Please visit www.sourcebeat.com to access the book.

This chapter is a tutorial on how to write a simple Spring web application using the Struts MVC framework for the front end, Spring for the middle-tier glue, and Hibernate for the back end. In Chapter 4, this application will be refactored to use the Spring MVC framework.

This chapter covers the following topics:

- Writing tests to verify functionality.
- Configuring Hibernate and Transactions.
- Loading Spring's *applicationContext.xml* file.
- Setting up dependencies between business delegates and DAOs.
- Wiring Spring into the Struts application.

Overview

You will create a simple application for user management that does basic CRUD (Create, Retrieve, Update and Delete). This application is called MyUsers, which will be the sample application throughout the book. It's a 3-tiered webapp, with an Action that calls a business delegate, which in turn calls a Data Access Object (DAO). The diagram below shows a brief overview of how the MyUsers application will work when you finish this tutorial. The numbers below indicate the order of flow – from the web (`UserAction`) to the middle tier, (`UserManager`), to the data layer (`UserDAO`) – and back again.

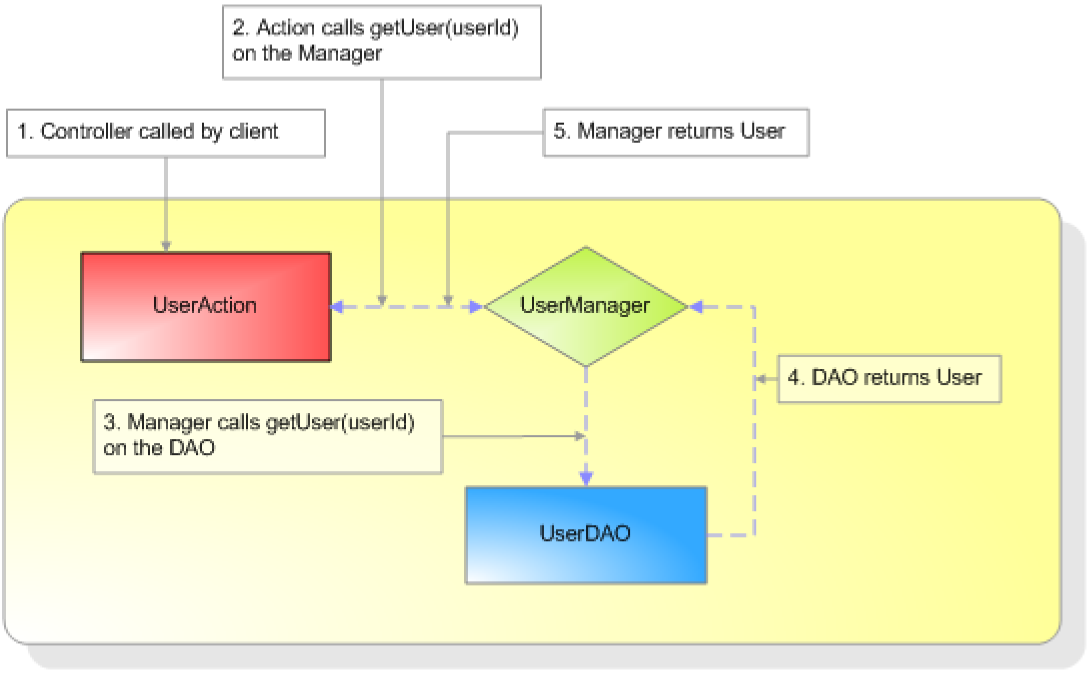

2. Action calls getUser(userId) on the Manager
1. Controller called by client
5. Manager returns User
UserAction
UserManager
4. DAO returns User
3. Manager calls getUser(userId) on the DAO
UserDAO

Figure 2.1: MyUsers application flow

This application uses Struts as the MVC framework because most readers are familiar with Struts. The real power of Spring lies in its declarative transactions, dependency binding and persistence support (for example Hibernate and iBATIS). *Chapter 4* refactors this application to use Spring's MVC framework.

Below are the ordered steps you will perform:

1. Download Struts and Spring.
2. Create project directories and an Ant build file.
3. Create a unit test for the persistence layer.
4. Configure Hibernate and Spring.
5. Create Hibernate DAO implementation.
6. Run the unit test and verify CRUD with DAO.
7. Create a Manager and Declare Transactions.
8. Create a unit test for the Struts Action.
9. Create an Action and model (`DynaActionForm`) for the web layer.
10. Run the unit test and verify CRUD with Action.
11. Create JSPs to allow CRUD through a web browser.
12. Verify the JSPs' functionality through your browser.
13. Replace the JSPs with Velocity templates.
14. Add Validation using Commons Validator.

Download Struts and Spring[1]

1. Download and install the following components:
   - JDK 1.4.2 (or above)
   - Tomcat 5.0+
   - Ant 1.6.1+
2. Set up the following environment variables:
   - JAVA_HOME
   - ANT_HOME
   - CATALINA_HOME
3. Add the following to your PATH environment variable:
   - JAVA_HOME/bin
   - ANT_HOME/bin
   - CATALINA_HOME/bin

To develop a Java-based web application, developers download JARs, create a directory structure, and create an Ant build file. For a Struts-only application, simplify this by using the *struts-blank.war*, which is part of the standard Struts distribution. For a webapp using Spring's MVC framework, use the *webapp-minimal* application that ships with Spring. Both of these are nice starting points, but neither simplifies the Struts-Spring integration nor takes into account unit testing. Therefore, I have made available to my readers **Equinox**.

Equinox is a bare-bones starter application for creating a Struts-Spring web application. It has a pre-defined directory structure, an Ant build file (for compiling, deploying and testing), and all the JARs you will need for a Struts, Spring and Hibernate-based webapp. Much of the directory structure and build file in Equinox is taken from my open-source AppFuse application. Therefore, Equinox is really just an "AppFuse Light" that allows rapid webapp development with minimal setup. Because it is derived from AppFuse, you will see many references to it in package names, database names and other areas. This is done purposefully so you can migrate from an Equinox-based application to a more robust AppFuse-based application.

In order to start MyUsers, download Equinox from http://sourcebeat.com/downloads and extract it to an appropriate location.

1. You can learn more about how I set up my development environment on Windows at http://raibledesigns.com/wiki/Wiki.jsp?page=DevelopmentEnvironment.

Create Project Directories and an Ant Build File

To set up your initial directory structure and Ant build file, extract the Equinox download onto your hard drive. I recommend putting projects in `C:\Source` on Windows and `~/dev` on Unix or Linux. For Windows users, now is a good time set your HOME environment variable to `C:\Source`. The easiest way to get started with Equinox is to extract it to your preferred "source" location, cd into the `equinox` directory and run **`ant new -Dapp.name=myusers`** from the command line.

Tip: I use Cygwin (www.cygwin.org) on Windows, which allows me to type forward-slashes, just like Unix/Linux. Because of this, all the paths I present in this book will have forward slashes. Please adjust for your environment accordingly (that is, use backslashes (\) for Windows' command prompt).

At this point, you should have the following directory structure for the MyUsers webapp:

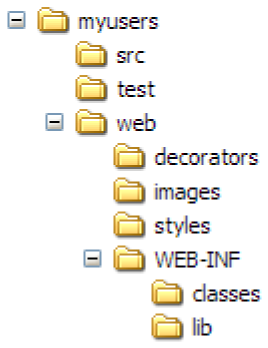


Figure 2.2: MyUsers application directory structure

Equinox contains a simple but powerful *build.xml* file to deploy, compile and test using Ant. For all the ant targets available, type “ant” in the MyUsers directory. The return should look like the following:

```
[echo] Available targets are:

[echo] compile   --> Compile all Java files
[echo] war       --> Package as WAR file
[echo] deploy    --> Deploy application as directory
[echo] deploywar --> Deploy application as a WAR file

[echo] install   --> Install application in Tomcat
[echo] remove    --> Remove application from Tomcat
[echo] reload    --> Reload application in Tomcat
[echo] start     --> Start Tomcat application
[echo] stop      --> Stop Tomcat application
[echo] list      --> List Tomcat applications

[echo] clean     --> Deletes compiled classes and WAR
[echo] new       --> Creates a new project
```

Equinox supports Tomcat’s Ant tasks. These tasks are already integrated into Equinox, but showing you *how* they were integrated will help you understand how they work.

Tomcat and Ant

Tomcat ships with a number of Ant tasks that allow you to install, remove and reload webapps using its Manager application. The easiest way to declare and use these tasks is to create a properties file that contains all the definitions. In Equinox, a *tomcatTasks.properties* file is in the base directory with the following contents:

```
deploy=org.apache.catalina.ant.DeployTask
undeploy=org.apache.catalina.ant.UndeployTask
remove=org.apache.catalina.ant.RemoveTask
reload=org.apache.catalina.ant.ReloadTask
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
list=org.apache.catalina.ant.ListTask
```

A number of targets are in *build.xml* for installing, removing and reloading the application:

```
<!-- Tomcat Ant Tasks -->
<taskdef file="tomcatTasks.properties">
    <classpath>
        <pathelement
            path="${tomcat.home}/server/lib/catalina-ant.jar"/>
    </classpath>
</taskdef>

<target name="install" description="Install application in Tomcat"
    depends="war">
    <deploy url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"
        war="file:${dist.dir}/${webapp.name}.war"/>
</target>

<target name="remove" description="Remove application from Tomcat">
    <undeploy url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>

<target name="reload" description="Reload application in Tomcat">
    <reload url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>

<target name="start" description="Start Tomcat application">
    <start url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>

<target name="stop" description="Stop Tomcat application">
    <stop url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>

<target name="list" description="List Tomcat applications">
    <list url="${tomcat.manager.url}"
```

```
            username="${tomcat.manager.username}"
            password="${tomcat.manager.password}"/>
    </target>
```

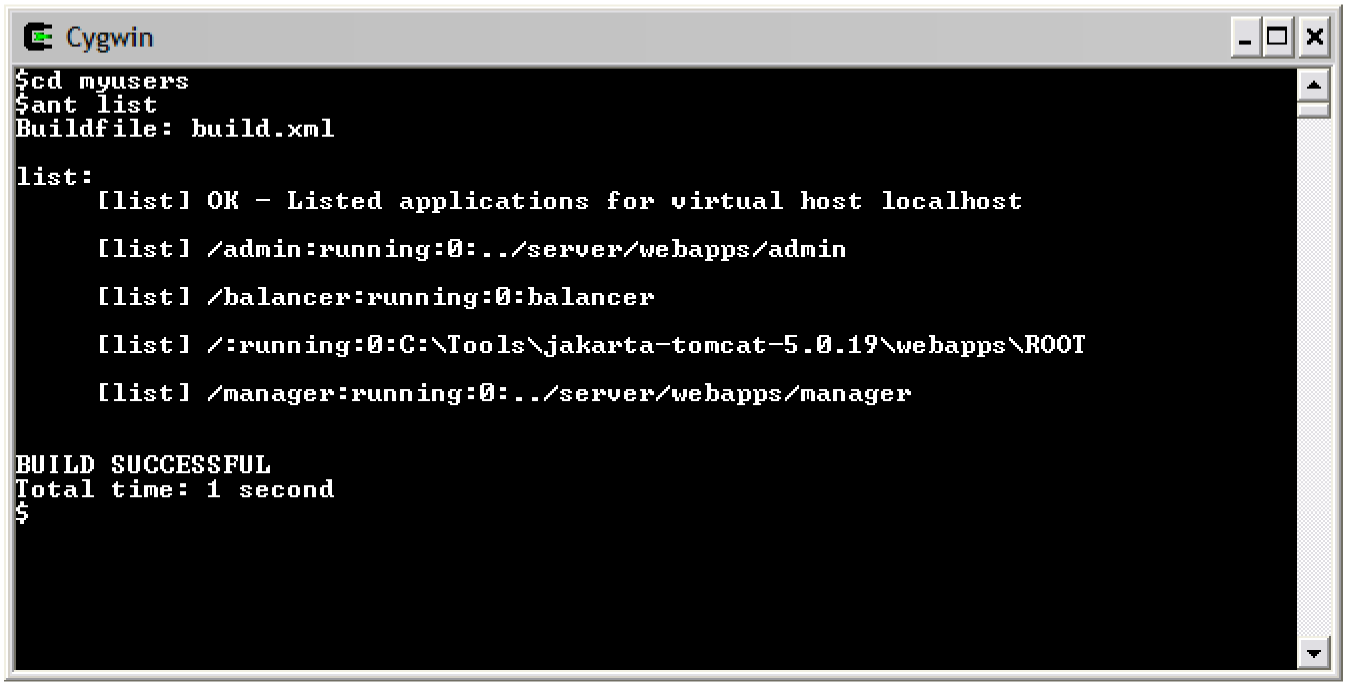
In the targets listed above, several `${tomcat.*}` variables need to be defined. These are in the *build.properties* file in the base directory. By default, they are defined as follows:

```
    # Properties for Tomcat Server
    tomcat.manager.url=http://localhost:8080/manager
    tomcat.manager.username=admin
    tomcat.manager.password=admin
```

To make sure the "admin" user is able to access the Manager application, open the *$CATALINA_HOME/conf/ tomcat-users.xml* file and verify that the following line exists. If it does not exist, you must create it. Note that the "roles" attribute may contain a comma-delimited list of roles.

```
    <user username="admin" password="admin" roles="manager"/>
```

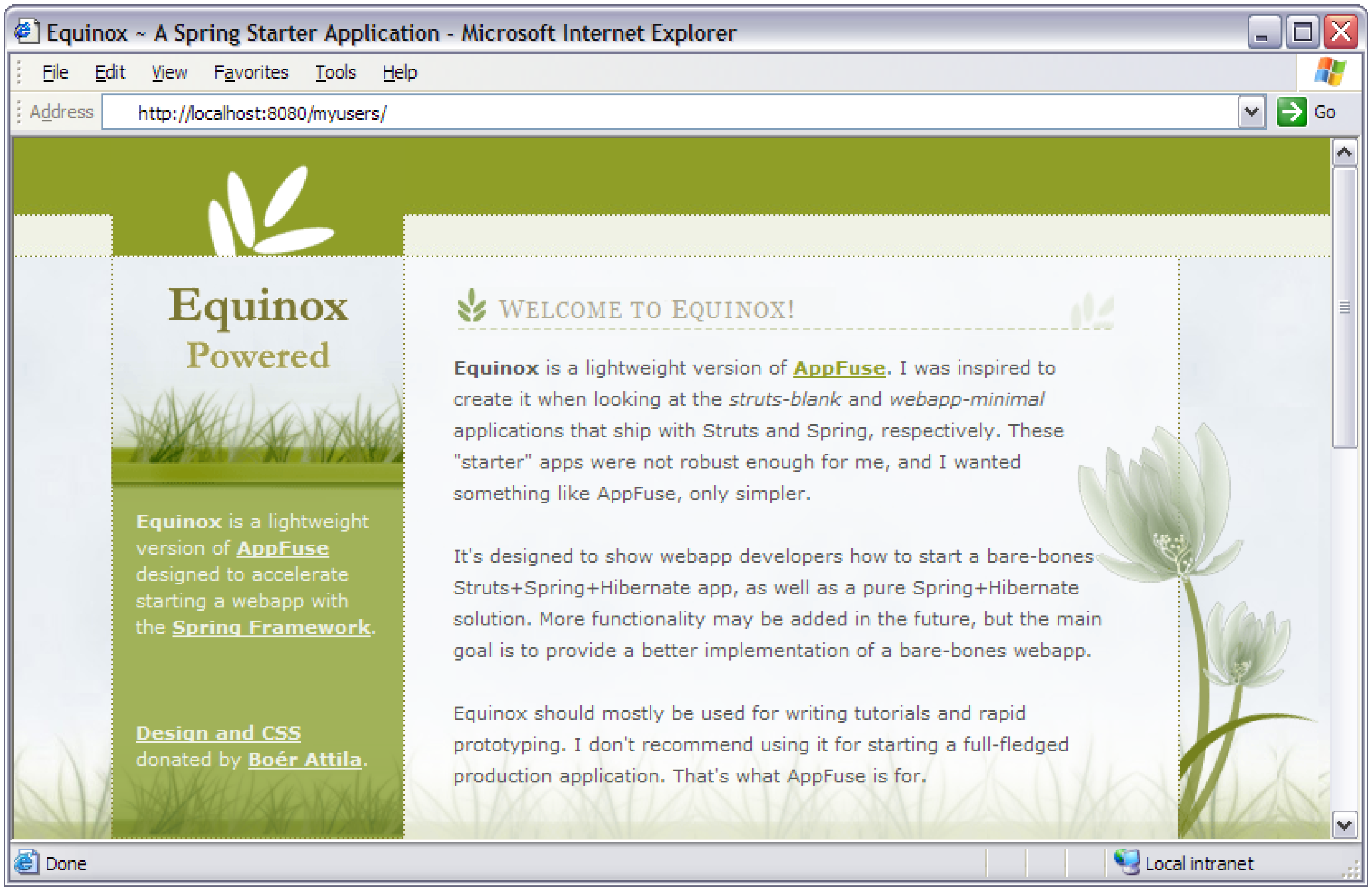
To test these changes, save all your files and start Tomcat. Then navigate to the "myusers" directory from the command line and try running "ant list." You should see a list of currently running applications on your Tomcat server.



```
$cd myusers
$ant list
Buildfile: build.xml

list:
     [list] OK - Listed applications for virtual host localhost

     [list] /admin:running:0:../server/webapps/admin

     [list] /balancer:running:0:balancer

     [list] /:running:0:C:\Tools\jakarta-tomcat-5.0.19\webapps\ROOT

     [list] /manager:running:0:../server/webapps/manager


BUILD SUCCESSFUL
Total time: 1 second
$
```

Figure 2.3: Results of the `ant list` command

Now you can install MyUsers by running **`ant deploy`**. Open your browser and go to http://localhost:8080/myusers. The “Welcome to Equinox” screen displays, as shown in Figure 2.4:


Equinox ~ A Spring Starter Application - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8080/myusers/ Go
Equinox Powered
Done
Local intranet

WELCOME TO EQUINOX!

Equinox is a lightweight version of **AppFuse**. I was inspired to create it when looking at the *struts-blank* and *webapp-minimal* applications that ship with Struts and Spring, respectively. These "starter" apps were not robust enough for me, and I wanted something like AppFuse, only simpler.

It's designed to show webapp developers how to start a bare-bones Struts+Spring+Hibernate app, as well as a pure Spring+Hibernate solution. More functionality may be added in the future, but the main goal is to provide a better implementation of a bare-bones webapp.

Equinox should mostly be used for writing tutorials and rapid prototyping. I don't recommend using it for starting a full-fledged production application. That's what AppFuse is for.

Equinox is a lightweight version of **AppFuse** designed to accelerate starting a webapp with the **Spring Framework**.

Design and CSS donated by **Boér Attila**.

Figure 2.4: Equinox Welcome page

In the next few sections, you will develop a User object and a Hibernate DAO to persist that object. You will use Spring to manage the DAO and its dependencies. Lastly, you will write a business delegate to use AOP and declarative transactions.

Create Unit Test for Persistence Layer

In the MyUsers app, you will use Hibernate for your persistence layer. Hibernate is an Object/Relational (O/R) framework that relates Java Objects to database tables. It allows you to very easily perform CRUD (Create, Retrieve, Update, Delete) on these objects. Spring makes working with Hibernate even easier. Switching from Hibernate to Spring+Hibernate reduces code by about 75%. This code reduction is sponsored by the removal of the `ServiceLocator` class, a couple of `DAOFactory` classes, and using Spring's runtime exceptions instead of Hibernate's checked exceptions.

Writing a unit test will help you formulate your `UserDAO` interface. To create a JUnit test for your `UserDAO`, complete the steps below:

1. Create a `UserDAOTest.java` class in the **test/org/appfuse/dao** directory. This class should extend `BaseDAOTestCase`, which already exists in this package. This parent class initializes Spring's `ApplicationContext` from the *web/WEB-INF/applicationContext.xml* file. Below is the code you will need for a minimal JUnit test:

```
package org.appfuse.dao;

// use your IDE to handle imports

public class UserDAOTest extends BaseDAOTestCase {
    private User user = null;
    private UserDAO dao = null;

    protected void setUp() throws Exception {
        log = LogFactory.getLog(UserDAOTest.class);
        dao = (UserDAO) ctx.getBean("userDAO");
    }

    protected void tearDown() throws Exception {
        dao = null;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(UserDAOTest.class);
    }
}
```

   This class won't compile yet because you haven't created your `UserDAO` interface. Before you do that, write a couple of tests to verify CRUD works on the `User` object.

2. Add the `testSave` and `testAddAndRemove` methods to the `UserDAOTest` class, as shown below:

```java
public void testSaveUser() throws Exception {
    user = new User();
    user.setFirstName("Rod");
    user.setLastName("Johnson");

    dao.saveUser(user);
    assertTrue("primary key assigned", user.getId() != null);
    log.info(user);
    assertTrue(user.getFirstName() != null);
}

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Bill");
    user.setLastName("Joy");

    dao.saveUser(user);

    assertTrue(user.getId() != null);
    assertTrue(user.getFirstName().equals("Bill"));

    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }

    dao.removeUser(user.getId());

    assertNull(dao.getUser(user.getId()));
}
```

From these test methods, you can see that you need to create a `UserDAO` with the following methods:

- `saveUser(User)`
- `removeUser(Long)`
- `getUser(Long)`
- `getUsers()` (to return all the users in the database)

3. Create a *UserDAO.java* file in the **src/org/appfuse/dao** directory and populate it with the code below:

   Tip: If you are using an IDE like Eclipse or IDEA, a “lightbulb” icon will appear to the left of a non-existent class and allow you to create it on-the-fly.

```
package org.appfuse.dao;

// use your IDE to handle imports

public interface UserDAO extends DAO {
    public List getUsers();
    public User getUser(Long userId);
    public void saveUser(User user);
    public void removeUser(Long userId);
}
```

   Finally, in order for the `UserDAOTest` and `UserDAO` to compile, create a `User` object to persists.

4. Create a *User.java* class in the **src/org/appfuse/model** directory and add “id,” “firstName” and “lastName” as member variables, as shown below:

```
package org.appfuse.model;
public class User extends BaseObject {
  private Long id;
  private String firstName;
  private String lastName;

    /*
     Generate your getters and setters using your favorite IDE:
     In Eclipse:
     Right-click -> Source -> Generate Getters and Setters
    */
}
```

   Notice that you’re extending a `BaseObject` class. It has the following useful methods: `toString()`, `equals()` and `hashCode()`. The latter two are required by Hibernate. After creating the User object, open the `UserDAO` anda `UserDAOTest` classes and organize imports with your IDE.

Configure Hibernate and Spring

Now that you have the Plain Old Java Object (POJO), create a mapping file so Hibernate can persist it.

1. In the **src/org/appfuse/model** directory, create a file named *User.hbm.xml* with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class name="org.appfuse.model.User" table="app_user">

        <id name="id" column="id" unsaved-value="0">
            <generator class="increment" />
        </id>
        <property name="firstName" column="first_name"
            not-null="true"/>
        <property name="lastName" column="last_name" not-null="true"/>

    </class>
</hibernate-mapping>
```

2. Add this mapping to Spring's *applicationContext.xml* file in the **web/WEB-INF** directory. Open this file and look for `<property name="mappingResources">` and change it to the following:

```
<property name="mappingResources">
    <list>
        <value>org/appfuse/model/User.hbm.xml</value>
    </list>
</property>
```

In the *applicationContext.xml* file, you can see how the database is set up and Hibernate is configured to work with Spring. Equinox is designed to work with an HSQL database named "db/appfuse." It will be created in your Ant "db" directory. Details of this configuration will be covered in the "How Spring Is Configured in Equinox" section.

3. Run **`ant deploy reload`** (with Tomcat running) and see the database tables being creating as part of Tomcat's console log:

```
INFO - SchemaExport.execute(98) | Running hbm2ddl schema export
INFO - SchemaExport.execute(117) | exporting generated schema to database
INFO - ConnectionProviderFactory.newConnectionProvider(53) | Initializing
connection provider:
org.springframework.orm.hibernate.LocalDataSourceConnectionProvider
INFO - DriverManagerDataSource.getConnectionFromDriverManager(140) |
Creating new JDBC connection to [jdbc:hsqldb:db/appfuse]
INFO - SchemaExport.execute(160) | schema export complete
```

Tip: If you'd like to see more (or less) logging, change the log4j settings in the *web/WEB-INF/classes/log4j.xml* file

4. To verify that the "app_user" table was actually created in the database, run **`ant browse`** to bring up a HSQL console. You should see the HSQL Database Manager as shown below:

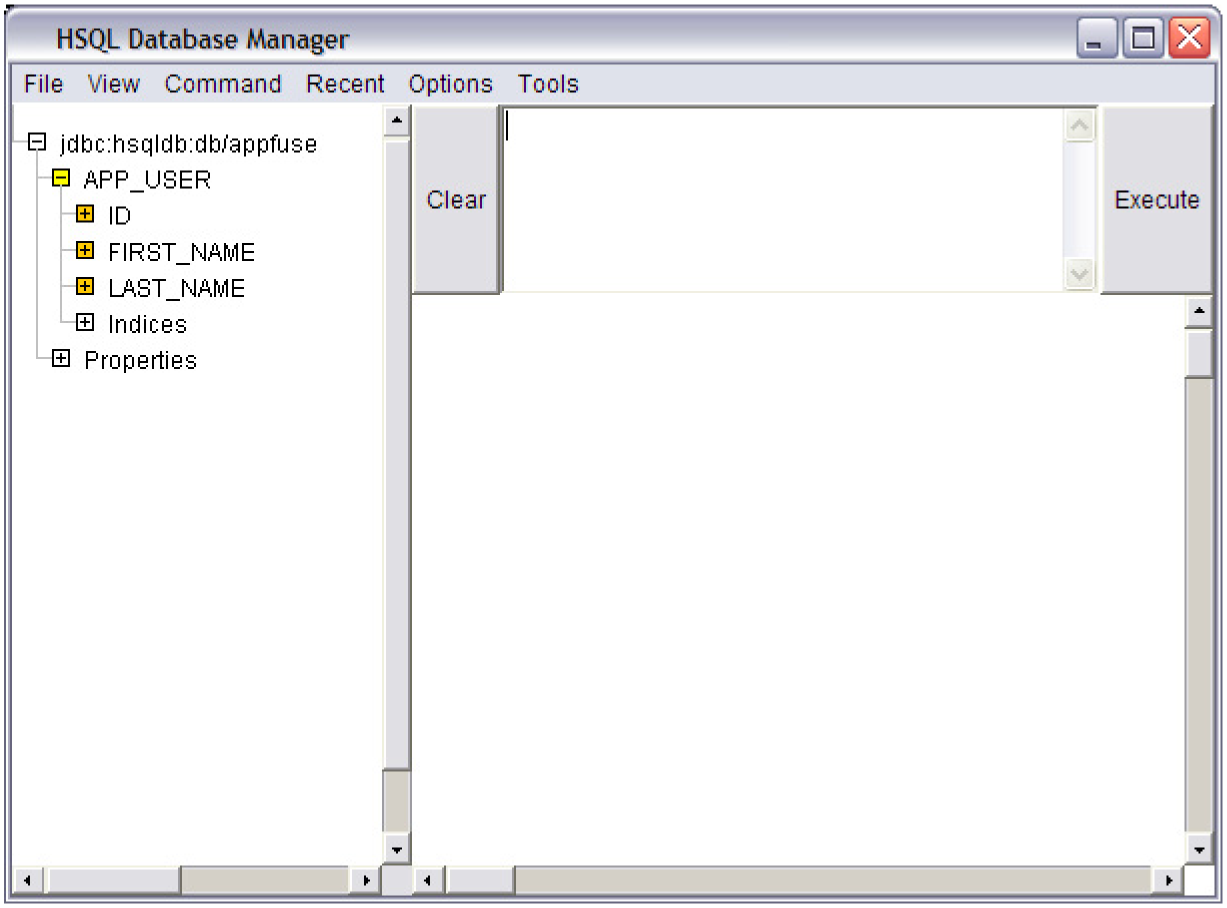


Figure 2.5: HSQL Database Manager

How Spring Is Configured in Equinox

It is very easy to configure any J2EE-based web application to use Spring. At the very least, you can simply add Spring's `ContextLoaderListener` to your *web.xml* file:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

This is a `ServletContextListener` that initializes when your webapp starts up. By default, it looks for Spring's configuration file at *WEB-INF/applicationContext.xml*. You can change this default value by specifying a `<context-param>` element named "contextConfigLocation." An example is provided below:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/sampleContext.xml</param-value>
</context-param>
```

The `<param-value>` element can contain a space or comma-delimited set of paths. In Equinox, Spring is configured using this Listener and its default "contextConfigLocation."

So, how does Spring know about Hibernate? This is the beauty of Spring: it makes it very simple to bind dependencies together. Look at the full contents of your *applicationContext.xml* file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
            <value>org.hsqldb.jdbcDriver</value>
        </property>
        <property name="url">
            <value>jdbc:hsqldb:db/appfuse</value>
        </property>
        <property name="username"><value>sa</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>
        <property name="mappingResources">
            <list>
                <value>org/appfuse/model/User.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                net.sf.hibernate.dialect.HSQLDialect
            </prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
        </props>
        </property>
    </bean>

    <!-- Transaction manager for a single Hibernate SessionFactory
(alternative to JTA) -->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref local="sessionFactory"/>
        </property>
    </bean>
</beans>
```

The first bean (dataSource) represents an HSQL database, and the second bean (sessionFactory) has a dependency on that bean. Spring just calls `setDataSource(DataSource)` on the `LocalSessionFactoryBean` to make this work. If you wanted to use a JNDI `DataSource` instead, you could easily change this bean's definition to something similar to the following:

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:comp/env/jdbc/appfuse</value>
     </property>
</bean>
```

Also note the "hibernate.hbm2ddl.auto" property in the "sessionFactory" definition. This property creates the database tables automatically when the application starts. Other possible values are *update* and *create-drop*.

The last bean configured is the "transactionManager" (and nothing is stopping you from using a JTA transaction manager), which is necessary to perform distributed transactions across two databases. If you want to use a JTA transaction manager, simply change this bean's "class" attribute to `org.springframework.transaction.jta.JtaTransactionManager`.

Now you can implement the `UserDAO` with Hibernate.

Implement UserDAO with Hibernate

To create a Hibernate implementation of the UserDAO, complete the following steps:

1. Create a *UserDAOHibernate.java* class in **src/org/appfuse/dao/hibernate** (you will need to create this directory/package). This file extends Spring's `HibernateDaoSupport` and implements `UserDAO`.

```
package org.appfuse.dao.hibernate;

// use your IDE to handle imports

public class UserDAOHibernate extends HibernateDaoSupport implements
UserDAO {
    private Log log = LogFactory.getLog(UserDAOHibernate.class);

    public List getUsers() {
        return getHibernateTemplate().find("from User");
    }

    public User getUser(Long id) {
        return (User) getHibernateTemplate().get(User.class, id);
    }

    public void saveUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);

        if (log.isDebugEnabled()) {
            log.debug(“userId set to: “ + user.getID());
        }
    }

    public void removeUser(Long id) {
      Object user = getHibernateTemplate().load(User.class, id);
        getHibernateTemplate().delete(user);
    }
}
```

   Spring’s `HibernateDaoSupport` class is a convenient super class for Hibernate DAOs. It has handy methods you can call to get a Hibernate `Session`, or a `SessionFactory`. The most convenient method is `getHibernateTemplate()`, which returns a `HibernateTemplate`. This template wraps Hibernate checked exceptions with runtime exceptions, allowing your DAO interfaces to be Hibernate exception-free.

Nothing is in your application to bind `UserDAO` to `UserDAOHibernate`, so you must create that relationship.

2. With Spring, add the following lines to the *web/WEB-INF/applicationContext.xml* file.

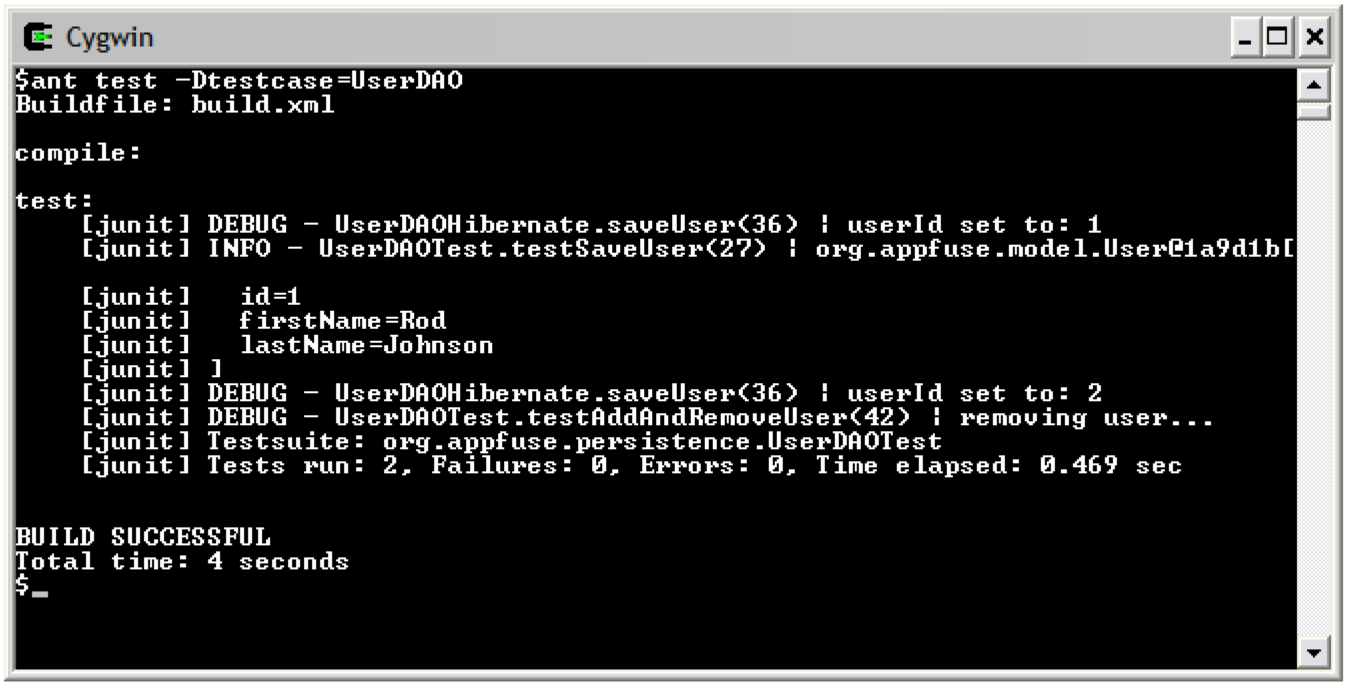
```
<bean id="userDAO"
    class="org.appfuse.dao.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

This sets a Hibernate `SessionFactory` on your `UserDAOHibernate` (which inherits `setSessionFactory()` from `HibernateDaoSupport`). Spring detects if a `Session` already exists (that is, it was opened in the web tier), and it uses that one instead of creating a new one. This allows you to use Hibernate's popular "Open Session in View" pattern for lazy loading collections.

Run Unit Test and Verify CRUD with DAO

Before you run this first test, tune down your default logging from informational messages to warnings.

1. Change `<level value="INFO"/>` to `<level value="WARN"/>` in the *log4j.xml* file (in **web/WEB-INF/classes**).
2. Run `UserDAOTest` using **`ant test`**. If this wasn't your only test, you could use **`ant test -Dtestcase=UserDAO`** to isolate which tests are run. After running this, your console should have a couple of log messages from your tests, as shown below:



```
$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

test:
    [junit] DEBUG - UserDAOHibernate.saveUser(36) | userId set to: 1
    [junit] INFO - UserDAOTest.testSaveUser(27) | org.appfuse.model.User@1a9d1b[

    [junit]   id=1
    [junit]   firstName=Rod
    [junit]   lastName=Johnson
    [junit] ]
    [junit] DEBUG - UserDAOHibernate.saveUser(36) | userId set to: 2
    [junit] DEBUG - UserDAOTest.testAddAndRemoveUser(42) | removing user...
    [junit] Testsuite: org.appfuse.persistence.UserDAOTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.469 sec


BUILD SUCCESSFUL
Total time: 4 seconds
$_
```

Figure 2.6: Results of the `ant test -Dtestcase=UserDAO` command

Create Manager and Declare Transactions

A recommended practice in J2EE development is to keep your layers separated. That is to say, the data layer (DAOs) shouldn't be bound to the web layer (servlets). Using Spring, it's easy to separate them, but it's useful to further separate these tiers using the business delegate[2] pattern.

The main reasons for using the business delegate pattern are:

- Most presentation tier components execute a unit of business logic. It's best to put this logic in a non-web class so a web-service or rich platform client can use the same API as a servlet.
- Most business logic can take place in one method, possibly using more than one DAO. Using a business delegate allows you to use Spring's declarative transactions feature at a higher "business logic" level.

The `UserManager` interface in the MyUsers application has the same methods as the `UserDAO`. The main difference is the Manager is more web-friendly; it accepts Strings where the `UserDAO` accepts Longs, and it returns a `User` object in the `saveUser()` method. This is convenient after inserting a new user (for example, to get its primary key). The Manager (or business delegate) is also a good place to put any business logic that your application requires.

2. Read more about this Core J2EE Pattern at http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html.

1. Start the "services" layer by first creating a `UserManagerTest` class in **test/org/appfuse/service** (you have to create this directory). This class extends JUnit's TestCase and contains the following code:

```
package org.appfuse.service;

// use your IDE to handle imports

public class UserManagerTest extends TestCase {
    private static Log log = LogFactory.getLog(UserManagerTest.class);
    private ApplicationContext ctx;
    private User user;
    private UserManager mgr;

protected void setUp() throws Exception {
    String[] paths = {"/WEB-INF/applicationContext.xml"};
    ctx = new ClassPathXmlApplicationContext(paths);
        mgr = (UserManager) ctx.getBean("userManager");
    }

    protected void tearDown() throws Exception {
        user = null;
        mgr = null;
    }
    // add testXXX methods here

    public static void main(String[] args) {
        junit.textui.TestRunner.run(UserDAOTest.class);
    }
}
```

In the `setUp()` method above, you are loading your *applicationContext.xml* file into the `ApplicationContext` variable using `ClassPathXmlApplicationContext`. Several methods are available for loading the `ApplicationContext`: from the classpath, the file system or within a web application. These methods will be covered in the *Chapter 3: The BeanFactory and How It Works*.

2. Code the first test method to verify that adding and removing a `User` object with the `UserManager` completes successfully:

```
public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");

    user = mgr.saveUser(user);

    assertTrue(user.getId() != null);

    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }

    String userId = user.getId().toString();
    mgr.removeUser(userId);

    user = mgr.getUser(userId);
    if (user != null) {
        fail("User object found in database!");
    }
}
```

This test is really an *integration test* rather than a *unit test* because it uses all the real components it depends on. To be more like a *unit test*, you would use EasyMock or a similar tool to "fake" the DAO. Using this, you could even get away from loading Spring's `ApplicationContext` and depending on any of Spring's APIs. I recommend the test we created because it tests all the internals that our project depends on (Spring, Hibernate, our classes), including the database. *Chapter 9* discusses refactoring the UserManagerTest to use mocks for its DAO dependency.

3. To compile the UserManagerTest, create the UserManager interface in the **src/org/appfuse/service** directory. Use the code below to create this class in the `org.appfuse.service` package:

```
package org.appfuse.service;

// use your IDE to handle imports

public interface UserManager {
    public List getUsers();
    public User getUser(String userId);
    public User saveUser(User user);
    public void removeUser(String userId);
}
```

4. Now create a new sub-package called `org.appfuse.service.impl` and create an implementation class of the `UserManager` interface.

```
package org.appfuse.service.impl;

// use your IDE to handle imports

public class UserManagerImpl implements UserManager {
    private static Log log = LogFactory.getLog(UserManagerImpl.class);
    private UserDAO dao;

    public void setUserDAO(UserDAO dao) {
        this.dao = dao;
    }

    public List getUsers() {
        return dao.getUsers();
    }

    public User getUser(String userId) {
        User user = dao.getUser(Long.valueOf(userId));

        if (user == null) {
            log.warn("UserId '" + userId + "' not found in database.");
        }

        return user;
    }

    public User saveUser(User user) {
        dao.saveUser(user);

        return user;
    }

    public void removeUser(String userId) {
        dao.removeUser(Long.valueOf(userId));
    }
}
```

This class has no indication that you're using Hibernate. This is important if you ever want to switch your persistence layer to use a different technology.

This class has a private `dao` member variable, as well as a `setUserDAO()` method. This allows Spring to perform its "dependency binding" magic and wire the objects together. Later, when you refactor this class to use a mock for its DAO, you'll need to add the `setUserDAO()` method to the `UserManager` interface.

5. Before running this test, configure Spring so `getBean("userManager")` returns the `UserManagerImpl` class. In *web/WEB-INF/applicationContext.xml*, add the following lines:

```
<bean id="userManager"
    class="org.appfuse.service.UserManagerImpl">
    <property name="userDAO"><ref local="userDAO"/></property>
</bean>
```

   The only problem with this is you're not leveraging Spring's AOP and, specifically, declarative transactions.

6. To do this, change the "userManager" bean to use a *ProxyFactoryBean*. A *ProxyFactoryBean* creates different implementations of a class, so that AOP can intercept and override method calls. For transactions, use `TransactionProxyFactoryBean` in place of the `UserManagerImpl` class. Add the following bean definition to the context file:

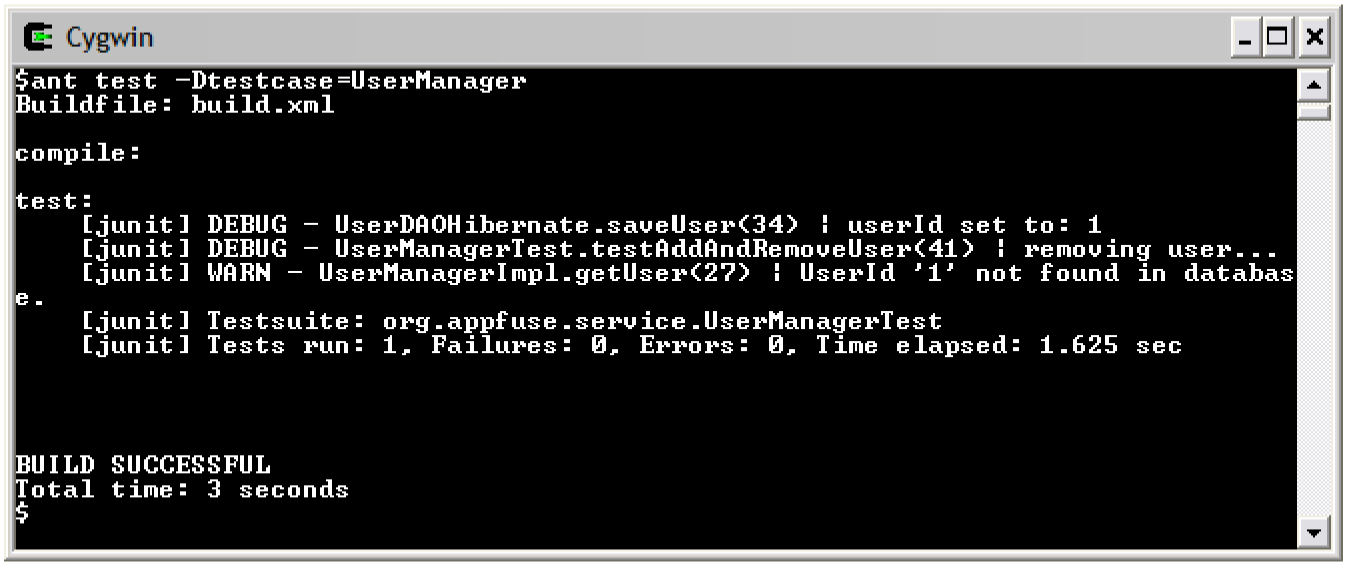
```
<bean id="userManager"
    class="org.springframework.transaction.interceptor.TransactionProxy
        FactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager"/>
    </property>
    <property name="target">
        <ref local="userManagerTarget"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

   You can see from this XML fragment that the `TransactionProxyFactoryBean` must have a "transactionManager" property set, and "transactionAttributes" defined.

7. Tell this Transaction Proxy the object you're mimicking: *userManagerTarget*. As part of this new bean, change the old "userManager" bean to have an id of "userManagerTarget."

```
<bean id="userManagerTarget"
    class="org.appfuse.service.impl.UserManagerImpl">
    <property name="userDAO"><ref local="userDAO"/></property>
</bean>
```

After editing *applicationContext.xml* to add definitions for "userManager" and "userManagerTarget," run **`ant test -Dtestcase=UserManager`** to see the following console output:



```
$ant test -Dtestcase=UserManager
Buildfile: build.xml

compile:

test:
    [junit] DEBUG - UserDAOHibernate.saveUser(34) | userId set to: 1
    [junit] DEBUG - UserManagerTest.testAddAndRemoveUser(41) | removing user...
    [junit] WARN - UserManagerImpl.getUser(27) | UserId '1' not found in databas
e.
    [junit] Testsuite: org.appfuse.service.UserManagerTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.625 sec



BUILD SUCCESSFUL
Total time: 3 seconds
$
```

Figure 2.7: Results of the `ant test -Dtestcase=UserManager` command

8. If you'd like to see the transactions execute and commit, add the XML below to the *log4j.xml* file:

```
<logger name="org.springframework.transaction">
    <level value="DEBUG"/> <!-- INFO does nothing -->
</logger>
```

Running the test again will give you a plethora of Spring log messages as it binds objects, creates transactions, and then commits them. You'll probably want to remove the above logger after running the test.

Congratulations! You've just implemented a Spring/Hibernate solution for the backend of a web application. You've also configured a business delegate to use AOP and declarative transactions. This is no small feat; give yourself a pat on the back!

Create Unit Test for Struts Action

The business delegate and DAO are now functional, so let's slap an MVC framework on top of this sucker! Whoa, there – not just yet. You can do the C (Controller), but not the V (View). Continue your Test-Driven Development path by creating a Struts Action for managing users.

The Equinox application is configured for Struts. Configuring Struts requires putting some settings in *web.xml* and defining a *struts-config.xml* file in the **web/WEB-INF** directory. Since there is a large audience of Struts developers, this chapter deals with *Struts way* first. *Chapter 4* deals with the *Spring way*. If you'd prefer to skip this section and learn the Spring MVC way, please refer to *Chapter 4: Spring's MVC Framework*.

To develop your first Struts Action unit test, create a `UserActionTest.java` class in **test/org/appfuse/web**. This file should have the following contents:

```
package org.appfuse.web;

// use your IDE to handle imports

public class UserActionTest extends MockStrutsTestCase {

    public UserActionTest(String testName) {
        super(testName);
    }

    public void testExecute() {
        setRequestPathInfo("/user");
        addRequestParameter("id", "1");
        actionPerform();
        verifyForward("success");
        verifyNoActionErrors();
    }
}
```

Create Action and Model (DynaActionForm) for Web Layer

1. Create a *UserAction.java* class in **src/org/appfuse/web**. This class extends `DispatchAction`, which you will use in a few minutes to *dispatch* to the different CRUD methods of this class.

   ```
   package org.appfuse.web;

   // use your IDE to handle imports

   public class UserAction extends DispatchAction {
       private static Log log = LogFactory.getLog(UserAction.class);

       public ActionForward execute(ActionMapping mapping,
                                    ActionForm form,
                                    HttpServletRequest request,
                                    HttpServletResponse response)
       throws Exception {
           request.getSession().setAttribute("test", "succeeded!");

           log.debug("looking up userId: " + request.getParameter("id"));

           return mapping.findForward("success");
       }
   }
   ```

2. To configure Struts so that the "/user" request path means something, add an *action-mapping* to *web/WEB-INF/struts-config.xml*. Open this file and add the following as an action-mapping:

   ```
   <action path="/user" type="org.appfuse.web.UserAction">
       <forward name="success" path="/index.jsp"/>
   </action>
   ```

3. Execute **`ant test -Dtestcase=UserAction`** and you should get the lovely "BUILD SUCCESSFUL" message.

4. Add a *form-bean* definition to the *struts-config.xml* file (in the `<form-beans>` section). For the Struts `ActionForm`, use a `DynaActionForm`, which is a JavaBean that gets created dynamically from an XML definition.

```
<form-bean name="userForm"
    type="org.apache.struts.action.DynaActionForm">
    <form-property name="user" type="org.appfuse.model.User"/>
</form-bean>
```

   You're using this instead of a concrete `ActionForm` because you only need a thin wrapper around the `User` object. Ideally, you could use the `User` object, but you'd lose the ability to validate properties and reset checkboxes in a Struts environment. Later, I'll show you how Spring makes this easier and allows you to use the `User` object in your web tier.

5. Modify your `<action>` definition to use this form and put it in the request:

```
<action path="/user" type="org.appfuse.web.UserAction"
    name="userForm" scope="request">
    <forward name="success" path="/index.jsp"/>
</action>
```

6. Modify your `UserActionTest` to test the different CRUD methods in your Action, as shown below:

```
public class UserActionTest extends MockStrutsTestCase {

    public UserActionTest(String testName) {
        super(testName);
    }

    // Adding a new user is required between tests because HSQL creates
    // an in-memory database that goes away during tests.
    public void addUser() {
        setRequestPathInfo("/user");
        addRequestParameter("method", "save");
        addRequestParameter("user.firstName", "Juergen");
        addRequestParameter("user.lastName", "Hoeller");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }

    public void testAddAndEdit() {
        addUser();

        // edit newly added user
        addRequestParameter("method", "edit");
        addRequestParameter("id", "1");
        actionPerform();
```

```
            verifyForward("edit");
            verifyNoActionErrors();
        }

        public void testAddAndDelete() {
            addUser();

            // delete new user
            setRequestPathInfo("/user");
            addRequestParameter("method", "delete");
            addRequestParameter("user.id", "1");
            actionPerform();
            verifyForward("list");
            verifyNoActionErrors();
        }

        public void testList() {
            addUser();
            setRequestPathInfo("/user");
            addRequestParameter("method", "list");
            actionPerform();
            verifyForward("list");
            verifyNoActionErrors();

            List users = (List) getRequest().getAttribute("users");
            assertNotNull(users);
            assertTrue(users.size() == 1);
        }
    }
```

7. Modify the UserAction so your tests will pass and it can handle CRUD requests. The easiest way to do this is to write edit, save and delete methods. Be sure to remove the existing “execute” method first. Below is the modified UserAction.java:

```
public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);
    private UserManager mgr = null;

    public void setUserManager(UserManager userManager) {
        this.mgr = userManager;
    }

    public ActionForward delete(ActionMapping mapping, ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
    throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        mgr.removeUser(request.getParameter("user.id"));

        ActionMessages messages = new ActionMessages();
        messages.add(ActionMessages.GLOBAL_MESSAGE,
                     new ActionMessage("user.deleted"));

        saveMessages(request, messages);

        return list(mapping, form, request, response);
    }

    public ActionForward edit(ActionMapping mapping, ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
    throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'edit' method...");
        }

        DynaActionForm userForm = (DynaActionForm) form;
        String userId = request.getParameter("id");

        // null userId indicates an add
        if (userId != null) {
            User user = mgr.getUser(userId);

            if (user == null) {
                ActionMessages errors = new ActionMessages();
                errors.add(ActionMessages.GLOBAL_MESSAGE,
                           new ActionMessage("user.missing"));
                saveErrors(request, errors);

                return mapping.findForward("list");
            }

            userForm.set("user", user);
        }

        return mapping.findForward("edit");
    }

    public ActionForward list(ActionMapping mapping, ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
    throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'list' method...");
        }
```

```
            request.setAttribute("users", mgr.getUsers());

            return mapping.findForward("list");
        }

        public ActionForward save(ActionMapping mapping, ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
        throws Exception {
            if (log.isDebugEnabled()) {
                log.debug("entering 'save' method...");
            }

            DynaActionForm userForm = (DynaActionForm) form;
            mgr.saveUser((User)userForm.get("user"));

            ActionMessages messages = new ActionMessages();
            messages.add(ActionMessages.GLOBAL_MESSAGE,
                         new ActionMessage("user.saved"));
            saveMessages(request, messages);

            return list(mapping, form, request, response);
        }
    }
```

Now that you’ve modified this class for CRUD, perform the following steps:

8. Modify *struts-config.xml* to use the `ContextLoaderPlugin` and configure Spring to set the `UserManager`. To configure the `ContextLoaderPlugin`, simply add the following to your *struts-config.xml* file:

```
<plug-in
    className=”org.springframework.web.struts.ContextLoaderPlugIn”>
    <set-property property=”contextConfigLocation”
        value=”/WEB-INF/applicationContext.xml,
               /WEB-INF/action-servlet.xml”/>
</plug-in>
```

   This plug-in will load the *action-servlet.xml* file by default. Since you want your Test Actions to know about your Managers, you must configure the plug-in to load *applicationContext.xml* as well.

9. For each action that uses Spring, define the action mapping to `type=”org.springframework.web.struts.DelegatingActionProxy”` and declare a matching Spring bean for the actual Struts action. Therefore, modify your action mapping to use this new class.
10. Modify your action mapping to work with `DispatchAction`.

   In order for the `DispatchAction` to work, add `parameter=”method”` to the mapping. This indicates (in a URL or hidden field) which method should be called. At the same time, add forwards for the “edit” and “list” forwards that are referenced in your CRUD-enabled `UserAction` class:

```
<action path="/user"
    type="org.springframework.web.struts.DelegatingActionProxy"
        name="userForm" scope="request" parameter="method">
        <forward name="list" path="/userList.jsp"/>
        <forward name="edit" path="/userForm.jsp"/>
    </action>
```

   Be sure to create the *userList.jsp* and *userForm.jsp* files in the “web” directory of MyUsers. You don’t need to put anything in them at this time.

11. As part of this plug-in, configure Spring to recognize the “/user” bean and to set the `UserManager` on it. Add the following bean definition to *web/WEB-INF/action-servlet.xml*:

```
<bean name="/user" class="org.appfuse.web.UserAction"
    singleton="false">
    <property name="userManager">
        <ref bean="userManager"/>
    </property>
</bean>
```

In this definition you're using `singleton="false"`. This creates new Actions for every request, alleviating the need for thread-safe Actions. Since neither your Manager nor your DAO contain member variables, this should work without this attribute (defaults to singleton="true").

12. Configure messages in the *messages.properties* ResourceBundle.

    In the `UserAction` class are a few references to success and error messages that will appear after operations are performed. These references are keys to messages that should exist in the ResourceBundle (or *messages.properties* file) for this application. Specifically, they are:

    - user.saved
    - user.missing
    - user.deleted

    Add these keys to the *messages.properties* file in **web/WEB-INF/classes,** as in the example below:

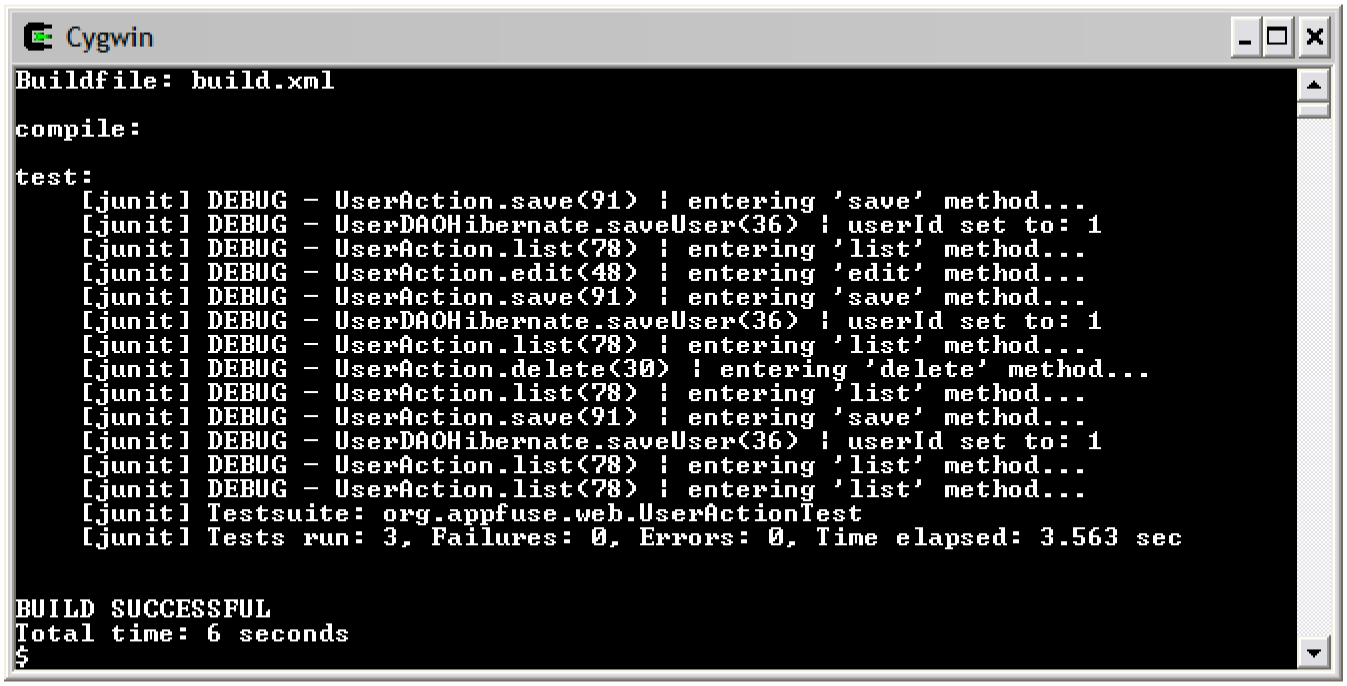
```
user.saved=User has been saved successfully.
user.missing=No user found with this id.
user.deleted=User successfully deleted.
```

    This file is loaded and made available to Struts via the `<message-resources>` element in *struts-config.xml*:

```
<message-resources parameter="messages"/>
```

Run Unit Test and Verify CRUD with Action

Run the **`ant test -Dtestcase=UserAction`**. It should result in the following output:



```
Buildfile: build.xml

compile:

test:
    [junit] DEBUG - UserAction.save(91) | entering 'save' method...
    [junit] DEBUG - UserDAOHibernate.saveUser(36) | userId set to: 1
    [junit] DEBUG - UserAction.list(78) | entering 'list' method...
    [junit] DEBUG - UserAction.edit(48) | entering 'edit' method...
    [junit] DEBUG - UserAction.save(91) | entering 'save' method...
    [junit] DEBUG - UserDAOHibernate.saveUser(36) | userId set to: 1
    [junit] DEBUG - UserAction.list(78) | entering 'list' method...
    [junit] DEBUG - UserAction.delete(30) | entering 'delete' method...
    [junit] DEBUG - UserAction.list(78) | entering 'list' method...
    [junit] DEBUG - UserAction.save(91) | entering 'save' method...
    [junit] DEBUG - UserDAOHibernate.saveUser(36) | userId set to: 1
    [junit] DEBUG - UserAction.list(78) | entering 'list' method...
    [junit] DEBUG - UserAction.list(78) | entering 'list' method...
    [junit] Testsuite: org.appfuse.web.UserActionTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.563 sec


BUILD SUCCESSFUL
Total time: 6 seconds
$
```

Figure 2.8: Results of the `ant test -Dtestcase=UserAction` command

Complete JSPs to Allow CRUD through a Web Browser

1. Add code to your JSPs (*userForm.jsp* and *userList.jsp*) so that they can render the results of your actions. If you haven't already done so, create a *userList.jsp* file in the **web** directory. Now add some code so you can see the all the users in the database. In the code below, the first line includes a *taglibs.jsp* file. This file contains all the JSP Tag Library declarations for this application, mostly for Struts Tags, JSTL and SiteMesh (which is used to "pretty up" the JSPs).

```
<%@ include file="/taglibs.jsp"%>

<title>MyUsers ~ User List</title>

<button onclick="location.href='user.do?method=edit'">Add User</button>

<table class="list">
<thead>
<tr>
    <th>User Id</th>
    <th>First Name</th>
    <th>Last Name</th>
</tr>
</thead>
<tbody>
<c:forEach var="user" items="${users}" varStatus="status">
<c:choose>
  <c:when test="${status.count % 2 == 0}"><tr class="even"></c:when>
  <c:otherwise><tr class="odd"></c:otherwise>
</c:choose>
    <td><a href="user.do?method=edit&amp;id=${user.id}">${user.id}</a></
td>
    <td>${user.firstName}</td>
    <td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```

You can see a row of headings (in the `<thead>`). JSTL's `<c:forEach>` tag iterates through the results and displays the users.

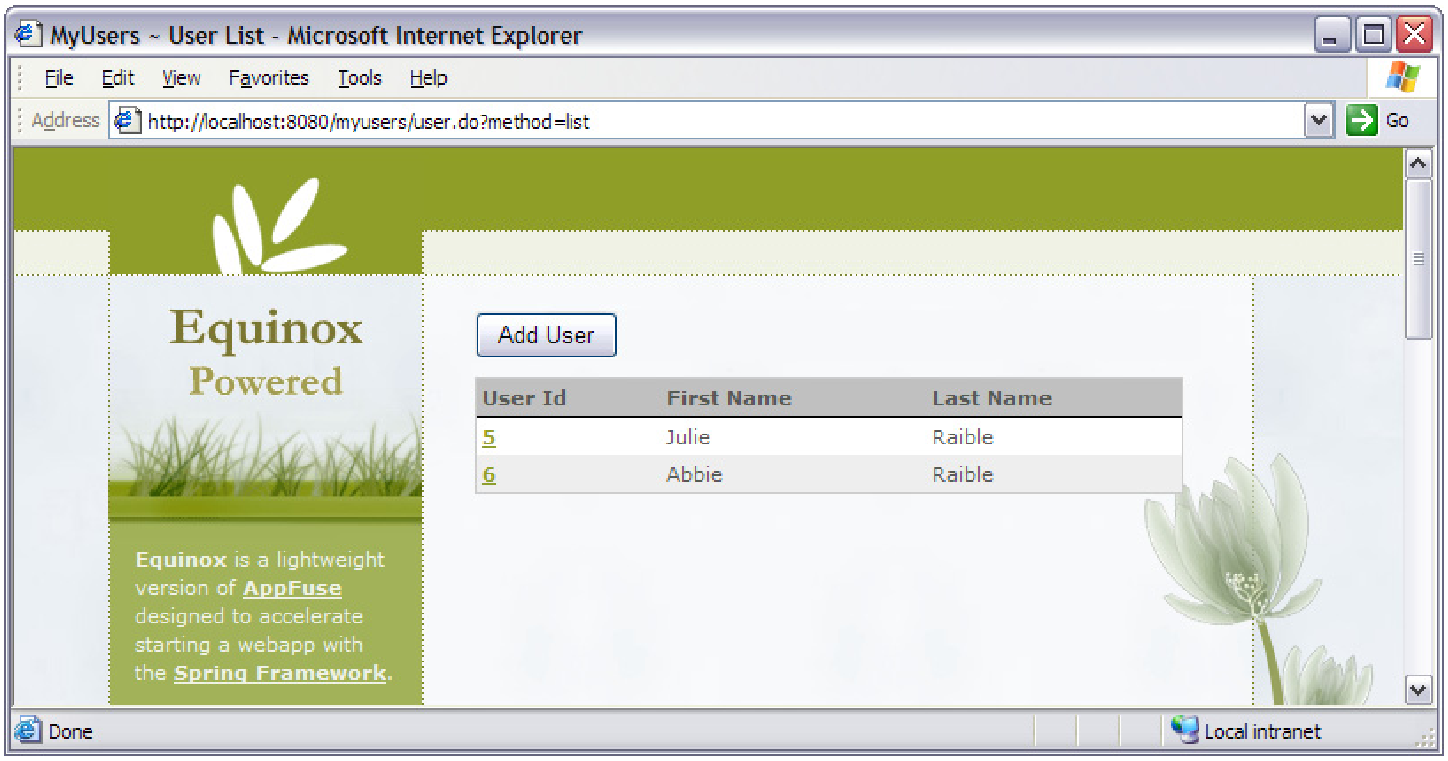
2. Populate the database so you can see some actual users. You have a choice: you can do it by hand, using **`ant browse`**, or you can add the following target to your *build.xml* file:

```
<target name="populate">
    <echo message="Loading sample data..."/>
    <sql driver="org.hsqldb.jdbcDriver"
         url="jdbc:hsqldb:db/appfuse"
         userid="sa" password="">
        <classpath refid="classpath"/>

        INSERT INTO app_user (id, first_name, last_name)
            values (5, 'Julie', 'Raible');
        INSERT INTO app_user (id, first_name, last_name)
            values (6, 'Abbie', 'Raible');

    </sql>
</target>
```

Warning! In order for the in-memory HSQLDB to work correctly with MyUsers, start Tomcat from the same directory from which you run Ant. Type “$CATALINA_HOME/bin/startup.sh” on Unix/Linux and “%CATALINA_HOME%\bin\startup.bat” on Windows.

Verify JSP's Functionality through Your Browser

1. With this JSP and sample data in place, view this JSP in your browser. Run **`ant deploy reload`**, then go to http://localhost:8080/myusers/user.do?method=list. The following screen displays:


MyUsers ~ User List - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8080/myusers/user.do?method=list
Go
Equinox Powered
Add User
Done
Local intranet

Equinox is a lightweight version of AppFuse designed to accelerate starting a webapp with the Spring Framework.

User Id	First Name	Last Name
5	Julie	Raible
6	Abbie	Raible

Figure 2.9: Results of `ant deploy reload` command

2. This example doesn't have an internationalized page title or column headings. Do this by adding some keys to the *messages.properties* file in **web/WEB-INF/classes**.

```
user.id=User Id
user.firstName=First Name
user.lastName=Last Name
```

The modified, i18n-ized header should now resemble the following:

```
<thead>
<tr>
    <th><bean:message key="user.id"/></th>
    <th><bean:message key="user.firstName"/></th>
    <th><bean:message key="user.lastName"/></th>
</tr>
</thead>
```

Note that JSTL's `<fmt:message key="...">` tag could also be used. If you wanted to add sorting and paging to this table, use the Display Tag (http://displaytag.sf.net). Below is an example of using this JSP tag:

```
<display:table name="users" pagesize="10" styleClass="list"
    requestURI="user.do?method=list">
    <display:column property="id" paramId="id" paramProperty="id"
        href="user.do?method=edit" sort="true"/>
    <display:column property="firstName" sort="true"/>
    <display:column property="lastName" sort="true"/>
</display:table>
```

Please refer to the display tag's documentation for internationalization of column headings.

3. Now that you've created your list, create the form where you can add/edit data. If you haven't already done so, create a *userForm.jsp* file in the **web** directory of MyUsers. Below is the code to add to this JSP to allow data entry:

```
<%@ include file="/taglibs.jsp"%>

<title>MyUsers ~ User Details</title>

<p>Please fill in user's information below:</p>

<html:form action="/user" focus="user.firstName">
<input type="hidden" name="method" value="save"/>
<html:hidden property="user.id"/>
<table>
<tr>
    <th><bean:message key="user.firstName"/>: </th>
    <td><html:text property="user.firstName"/></td>
</tr>
<tr>
    <th><bean:message key="user.lastName"/>: </th>
    <td><html:text property="user.lastName"/></td>
</tr>
<tr>
    <td></td>
    <td>
        <html:submit styleClass="button">Save</html:submit>
      <c:if test="${not empty param.id}">
        <html:submit styleClass="button"

            onclick="this.form.method.value='delete'">
            Delete</html:submit>
      </c:if>
    </td>
</table>
</html:form>
```

Note: If you're developing an application with internationalization (i18n), replace the informational message (at the top) and the button labels with `<bean:message>` or `<fmt:message>` tags. This is a good exercise for you. For informational messages, I recommend key names like *pageName.message* (such as, "userForm.message"), and button names like *button.name* (such as "button.save").

4. Run **`ant deploy`** and perform CRUD on a user from your browser.

The last thing that most webapps need is validation. In the next section, you'll configure Struts' Validator to make the user's last name a required field.

Adding Validation Using Commons Validator

In order to enable validation in Struts, perform the following steps:

1. Add the `ValidatorPlugIn` to *struts-config.xml*.
2. Create a *validation.xml* file that specifies that `lastName` is a required field.
3. Change the `DynaActionForm` to be a `DynaValidatorForm`.
4. Configure validation for the `save()` method, but not for others.
5. Add validation errors to *messages.properties*.

Add the Validator Plug-in to *struts-config.xml*

Configure the Validator plug-in by adding the following XML fragment to your *struts-config.xml* file (right after the Spring plug-in):

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
        property="pathnames" value="/WEB-INF/validator-rules.xml,
                                    /WEB-INF/validation.xml"/>
</plug-in>
```

From this you can see that the Validator is going to look for two files in the **WEB-INF** directory: *validator-rules.xml* and *validation.xml*. The first file, *validator-rules.xml*, is a standard file that's distributed as part of Struts. It defines all the available validators, as well as their client-side JavaScript functions. The second file, *validation.xml*, contains the validation rules for each form.

Edit the *validation.xml* File to Specify That `lastName` Is a Required Field

The *validation.xml* file has a number of standard elements to match its Document Type Definition (DTD), but you only need the `<form>` and `<field>` elements you see below. Please refer to the Validator's documentation for more information. Add the following `<formset>` between the `<form-validation>` tags in *web/WEB-INF/validation.xml*:

```
<formset>
    <form name="userForm">
        <field property="user.lastName" depends="required">
            <arg0 key="user.lastName"/>
        </field>
    </form>
</formset>
```

Change the `DynaActionForm` to `DynaValidatorForm`

Now change the `DynaActionForm` to a `DynaValidatorForm` in *struts-config.xml*.

```
<form-bean name="userForm"
    type="org.apache.struts.validator.DynaValidatorForm">
...
```

Configure Validation for `save()` Method, But Not for Others

One unfortunate side effect of using Struts' `DispatchAction` is that validation is turned on at the mapping level. In order to turn validation off for the list and edit screen, you could create a separate mapping with *validate="false"*. For example, AppFuse's `UserAction` has two mappings: "/editUser" and "/saveUser". However, there's an easier way that requires less XML, and only slightly more Java.

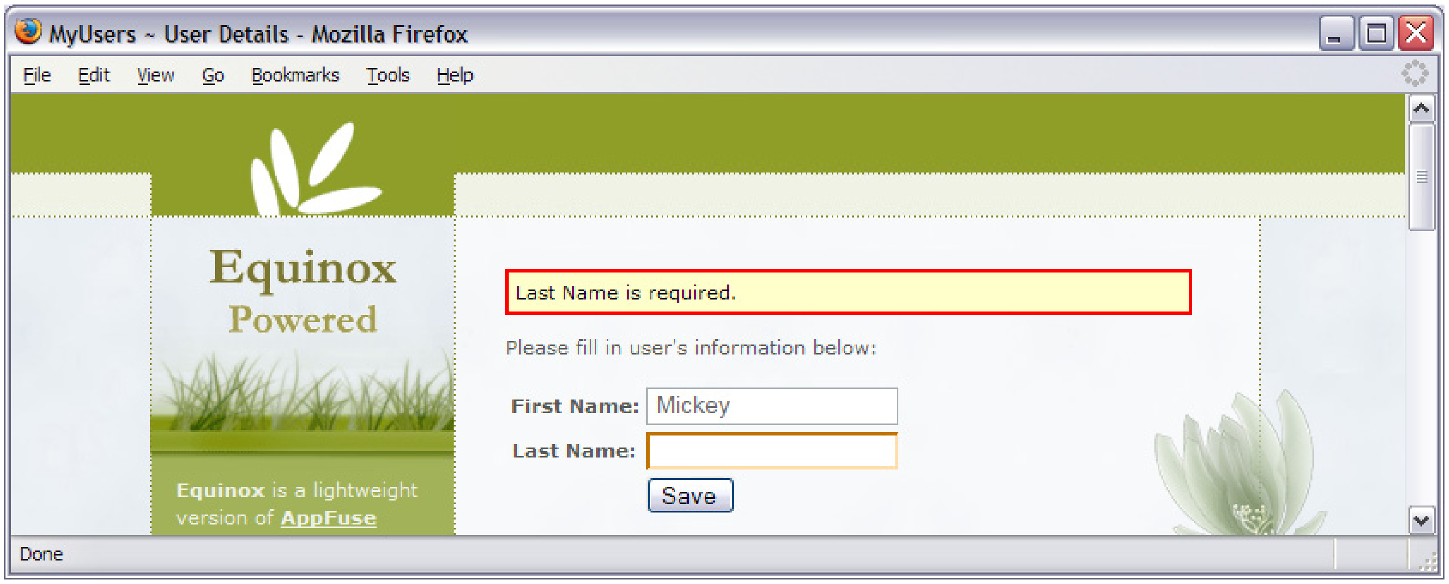
1. In the mapping for "/user", add *validate="false"*.
2. In `UserAction.java`, modify the `save()` method to call `form.validate()` and return to the edit screen if any errors are found.

```
if (log.isDebugEnabled()) {
    log.debug("entering 'save' method...");
}

// run validation rules on this form
ActionMessages errors = form.validate(mapping, request);
if (!errors.isEmpty()) {
    saveErrors(request, errors);
return mapping.findForward("edit");
}

DynaActionForm userForm = (DynaActionForm) form;
```

When working with `DispatchAction`, this is cleaner than having two mappings with one measly attribute changed. However, the *two mappings* approach has some advantages:

- It allows you to specify an "input" attribute that indicates where to go when validation fails.
- You can declare a "roles" attribute on your mapping to specify who can access that mapping. For instance, anyone can see the "edit" screen, but only administrators can save it.

3. Run **`ant deploy`** reload and try to add a new user without a last name. You will see a validation error indicating that last name is a required field, as in the example below:


MyUsers ~ User Details - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Equinox Powered
Equinox is a lightweight version of AppFuse
Save
Done

Last Name is required.

Please fill in user's information below:

First Name: Mickey

Last Name:

Figure 2.10: Result of the `ant deploy` command

Another nice feature of the Struts Validator is client-side validation.

4. To enable this quickly, add an "onsubmit" attribute to the `<form>` tag (in *web/userForm.jsp*), and a `<html:javascript>` tag at the bottom of the form.

```
<html:form action="/user" focus="user.firstName"
    onsubmit="return validateUserForm(this)">
...
</html:form>

<html:javascript formName="userForm"/>
```

Now if you run **`ant deploy`** and try to save a user with a blank last name, you will get a JavaScript alert stating that "Last Name is required." The one issue with the short-form of the `<html:javascript>` tag is that it puts all of the Validator's JavaScript functions into your page. There is a better way: include the JavaScript from an outside page (which is itself generated). How to do this will be covered in *Chapter 5*.

Congratulations! You've just developed a webapp that talks to a database, implements validation and even displays success and error messages. In Chapter 4, you will convert this application to use Spring's MVC framework. In Chapter 5, you will add exception handling, file uploading and e-mailing features. Chapter 6 will explore alternatives to JSP, and in Chapter 7 you'll add alternative DAO implementations using iBATIS, JDO and Spring's JDBC.

Summary

Spring is a great framework for reducing the amount of code you have to write. If you look at the number of steps in this tutorial, most of them involved setting up or writing code for Struts. Spring made the DAO and Manager implementations easy. It also reduced most Hibernate calls to one line and allowed you to remove any Exception handling that can sometimes be tedious. In fact, most of the time I spent writing this chapter (and the MyUsers app) involved configuring Struts.

I have two reasons for writing this chapter with Struts as the MVC Framework. The first is because I think that's the framework most folks are familiar with, and it's easier to explain a Struts-to-Spring migration than a JSP/Servlet-to-Spring migration. Secondly, I wanted to show you how writing your MVC layer with Struts can be a bit cumbersome. In *Chapter 4*, you'll refactor the web layer to use Spring's MVC Framework. I think you'll find it a bit refreshing to see how much easier and more intuitive it is.