



Spring LIVE



Matt Raible



Spring Live

by Matt Raible

Copyright © 2004 by SourceBeat, LLC.

Cover Copyright © 2004 by SourceBeat, LLC.

All rights reserved.

Published by SourceBeat, LLC, Highlands Ranch, Colorado.

Managing Editor: James Goodwill

Technical Editor: Dion Almaer

Copy Editor: Amy Kesic

Layout Designer: Amy Kesic

Cover Designer: Max Hayes

ISBN: 0974884340

Many designations used by organizations to distinguish their products are claimed as trademarks. These trademarked names may appear in this book. We use the names in an editorial fashion only with no intention of infringing on the trademark; therefore you will not see the use of a trademark symbol with every occurrence of the trademarked name.

As every precaution has been taken in writing this book, the author and publisher will in no way be held liable for any loss or damages resulting from the use of information contained in this book.

Table of Contents

Introduction	vi
--------------------	----

Chapter 1: Introducing Spring 1

The History of Spring	2
About Spring	3
Why Everyone Loves It	3
Common Criticisms of Spring	4
How Spring Works	4
How Spring Makes J2EE Easier	7
Coding to Interfaces	7
Easy Testability	7
Reducing Coupling: Factory Pattern vs. Spring	8
Configuring and Binding Class Dependencies	10
Object/Relational Mapping Tools	11
Summary	12

Chapter 2: Spring Quick Start Tutorial 13

Overview	14
Download Struts and Spring	16
Create Project Directories and an Ant Build File	17
Tomcat and Ant	18
Create Unit Test for Persistence Layer	22
Configure Hibernate and Spring	25
How Spring Is Configured in Equinox	27
Implement UserDAO with Hibernate	30
Run Unit Test and Verify CRUD with DAO	32
Create Manager and Declare Transactions	33
Create Unit Test for Struts Action	39
Create Action and Model (DynaActionForm) for Web Layer	40
Run Unit Test and Verify CRUD with Action	47
Complete JSPs to Allow CRUD through a Web Browser	48
Verify JSP's Functionality through Your Browser	50
Adding Validation Using Commons Validator	53
Add the Validator Plug-in to struts-config.xml	53
Edit the validation.xml File to Specify That lastName Is a Required Field	54
Change the DynaActionForm to DynaValidatorForm	54
Configure Validation for save() Method, But Not for Others	55
Summary	57

Chapter 3: The BeanFactory and How It Works 58

About the BeanFactory	59
A Bean's Lifecycle in the BeanFactory	60
Inversion of Control	61
The Bean Definition Exposed	63
Configuring Properties and Dependencies	66
Specifying Dependencies with <ref>	67
Pre-Initializing Your Beans	68
Autowiring	68
Dependency Checks	68
setXXX()	69
setBeanFactory()	71
afterPropertiesSet()	72
init-method	72
Ready State	73
Destroying Beans	73
The ApplicationContext: Talking to Your Beans	74
Get That Context!	74
Tips for Unit Testing and Loading Contexts	75
Internationalization and MessageSource	76
Event Publishing and Subscribing	77
A Closer Look at MyUser's applicationContext.xml	77
Summary	78

Chapter 4: Spring's MVC Framework 79

Overview	81
Unit Testing Spring Controllers	83
Converting Struts to Spring MVC	84
Modify web.xml to Use Spring's DispatcherServlet	85
Create Unit Test for UserController	86
Create UserController and Configure action-servlet.xml	88
Modify userList.jsp to Work with Spring	91
Create Unit Test for UserFormController	94
Create UserFormController and Configure it in action-servlet.xml	97
Modify userForm.jsp to Use Spring's JSP Tags	101
Configure Commons Validator for Spring	104
SimpleFormController: Method Lifecycle Review	108
Spring's JSP Tags	111
Summary	112

Chapter 5: Advanced MVC: Templates, Validation, Exceptions and Uploading Files	113
Templating with SiteMesh	115
Installation and Configuration	116
Step 1: Configure SiteMesh in web.xml	117
Step 2: Create Configuration Files	118
Step 3: Create a Decorator	119
Templating with Tiles	122
Installation and Configuration	122
Step 1: Configure Spring to Recognize Tiles	123
Step 2: Create a Base Layout	124
Step 3: Create Page Definitions	126
Validating the Spring Way	130
Using Commons Validator	132
XDoclet	133
Chaining Validators	134
Validating in Business Delegates	134
Spring's Future Declarative Validation Framework	136
Exception Handling in Controllers	137
Uploading Files	141
Intercepting the Request	147
Sending E-Mail	150
Summary	153

This book is written for Java developers familiar with web frameworks. Its main purpose is for Java developers to learn Spring and evaluate it against other frameworks. One of my hopes is to compare Spring to other web frameworks, or at least show how it can be integrated with other frameworks (i.e. Struts, WebWork, maybe even Tapestry down the road). This book will contain a usable sample application that includes Spring code to wire DAOs and Services together. The book does have a bit of a Struts perspective to it as I have been a Struts developer for almost three years and Struts is the most popular web framework today. It is only natural that I use my experience in my writing.

Chapter 1: Introducing Spring covers the basics of Spring, how it came to be and why it's getting so much press and rave reviews. It compares the traditional way of resolving dependencies (binding interfaces to implementations using a Factory Pattern) and how Spring does it all in XML. It also briefly covers how it simplifies the Hibernate API.

Chapter 2: Spring Quick Start Tutorial 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.

Chapter 3: The BeanFactory and How It Works. The BeanFactory represents the heart of Spring, so it's important to know how it works. This chapter explains how bean definitions are written, their properties, dependencies, and autowiring. It also explains the logic behind making singleton beans versus prototypes. Then it delves into Inversion of Control, how it works, and the simplicity it brings. This chapter dissects the Lifecycle of a bean in the BeanFactory to show how it works. This chapter also inspects the applicationContext.xml file for the MyUsers application created in Chapter 2.

Chapter 4: Spring's MVC Framework describes the many features of Spring's MVC framework. It shows you how to replace the Struts layer in MyUsers with Spring. It covers the DispatcherServlet, various Controllers, Handler Mappings, View Resolvers, Validation and Internationalization. It also briefly covers Spring's JSP Tags.

Chapter 5: Advanced MVC Framework: Templates, Validation, Exceptions and Uploading Files covers advanced topics in web frameworks, particularly validation and page decoration. It shows the user how to use Tiles or SiteMesh to decorate a web application. It also explains how the Spring framework handles validation, and shows examples of using it in the web business layers. Finally, it explains a strategy for handling exceptions in the controllers, how to upload files and how to send e-mail.

Chapter 6: View Options covers the different options for views in Spring's MVC architecture. At the time of this writing, the options are: JSP, Velocity, XSLT, XMLC, and PDF. This chapter aims to become a reference for configuring all Spring-supported views. It will also contain a brief overview how each view works and a comparison will be given for constructing the User List of MyUsers with each option. In for each view option will be focused on as well.

Chapter 7: Persistence Strategies: Hibernate, JDBC, iBatis and JDO. Hibernate is quickly becoming a popular choice for persistence in Java applications, but sometimes it doesn't fit. If you have an existing database schema, or even pre-written SQL, sometimes it's better to use JDBC or iBATIS (which supports externalized SQL in XML files). In this chapter, I will refactor the MyUsers application to support both JDBC and iBATIS as persistence framework options. Lastly, I will briefly touch on JDO, give some code samples and compare it to Hibernate.

Chapter 8: Unit Testing with Spring (Hibernate and MVC) covers using JUnit to test the non-web layers and strategies for obtaining the applicationContext. For the Controllers, it will demonstrate how to use Cactus for testing in-container, as well as JUnit for out-of-container testing. For the view options (including PDF), it will demonstrate using Canoo's WebTest.

Chapter 9: AOP. Aspect Oriented Programming has received a lot of hype in the Java community in the last year. What is AOP and how can it help you in your applications? This chapter will cover the basics of AOP and give some useful examples of where/how AOP might help you. It will also cover how security can be a concern for AOP-enabled applications.

Chapter 10: Transactions. Transactions are an important part of J2EE, allowing several database calls to be viewed as one, and rolled back if they don't all succeed. One of the most highlighted features of EJBs is declarative transactions. This chapter will demonstrate how declarative transactions are possible in Spring and explain different transaction strategies.

Chapter 11: AppFuse: a Quick Way to Start Your Spring Apps. AppFuse is an application I wrote that allows a developer to get a webapp project started quickly. It leverages Ant, XDoclet and Hibernate heavily to create a database, configure Tomcat and deploy your application. It has unit tests for every layer (persistence, business and web) and uses Ant to automate running tests. This chapter will cover how Spring is used to bind the app's layers together. I will also create a version of AppFuse using Spring's MVC framework and I'll analyze my thoughts and findings from that process.

Introducing Spring

The Basics of Spring and Its History

This chapter covers the basics of Spring, how it came to be and why it's getting so much press and rave reviews. It compares the traditional way of resolving dependencies (binding interfaces to implementations using a Factory Pattern) and how Spring does it all in XML. It also briefly covers how it simplifies the Hibernate API.

The History of Spring

Rod Johnson is the ingenious inventor of Spring. It started from infrastructure code in his book, [Expert One-on-One J2EE Design and Development](#), in late 2002. I highly recommend this book. In it, Rod explains his experiences with J2EE and how Enterprise JavaBeans (EJBs) are often overkill for projects. He believes a lightweight, JavaBeans-based framework can suite most developers' needs. The framework described eventually became known as The Spring Framework when it was open-sourced on [SourceForge](#) in February 2003. At this point, Rod was joined by Juergen Hoeller as Lead Developer and right-hand man of Spring. Rod and Juergen have added many other developers over the last several months. At the time of this writing, sixteen developers are on Spring's committee list. Rod and Juergen have recently written a book titled *Expert One-on-One J2EE Development without EJB* that describes how Spring solves many of the problems with J2EE.

The architectural foundations of Spring have been developed by Rod since early 2000 (before Struts or any other frameworks I know of). These foundations were built from Rod's experiences building infrastructure on a number of successful commercial projects. Spring's foundation is constantly being enhanced and re-enforced by hundreds (possibly thousands) of developers. All are bringing their experience to the table, and you can literally watch Spring become stronger day-by-day. Its community is thriving, its developers are enthusiastic and dedicated and it's quite possibly the best thing that has ever happened to J2EE.

About Spring

According to Spring's [web site](#), "Spring is a layered J2EE application framework based on code published in [Expert One-on-One J2EE Design and Development](#) by Rod Johnson." At its core, it provides a means to manage your business objects and their dependencies. For example, using Inversion of Control (IoC), it allows you to specify that a Data Access Object (DAO) depends on a `DataSource`. It also allows a developer to code to interfaces and simply define the implementation using an XML file. Spring contains many classes that support other frameworks (such as Hibernate and Struts) to make integration easier.

Following J2EE Design Patterns can be cumbersome and unnecessary at times (and in fact often become anti-patterns). Spring is like following design patterns, but everything is simplified. For example, rather than writing a `ServiceLocator` to look up Hibernate Sessions, you can configure a `SessionFactory` in Spring. This allows you to follow the best practices of J2EE field experts rather than trying to figure out the latest pattern.

Why Everyone Loves It

If you follow online forums like [TheServerSide.com](#) or [JavaLobby.org](#), you've likely seen Spring mentioned. It has even more traction in the Java blogging community (such as [JavaBlogs.com](#) and [JRoller.com](#)). Many developers are describing their experiences with Spring and praising its ease of use.

Not only does Spring solve developers' problems, it also enforces good programming practices like coding to interfaces, reducing coupling and allowing for easy testability. In the modern era of programming, particularly in Java, good developers are practicing Test Driven Development (TDD). TDD is a way of letting your tests, or clients of your classes, drive the design of those classes. Rather than building a class, then trying to retrofit the client, you're building the client first. This way, you know exactly what you want from the class you're developing. Spring has a rich test suite of its own that allows for easy testing of your classes.

Compare this to "best practices" from J2EE, where the blueprints recommend that you use EJBs to handle business logic. EJBs require an EJB container to run, so you have to startup your container to test them. When's the last time you started up an EJB server like WebLogic, WebSphere or JBoss? It can test your patience if you have to do it over and over to test your classes.

Note: Recently, an article titled "Testing EJBs with OpenEJB" showed a faster way to test EJBs, but it's only a workaround for the testability problems inherent in EJBs.

Common Criticisms of Spring

With success, there's always some criticism. The most compelling argument I've seen against Spring is that it's not a "standard," meaning it's not part of the J2EE specification and it hasn't been developed through the Java Community Process. The same folks who argue against Spring advocate EJBs, which are standard. However, the main reason for standards is to ensure portability across appservers. The code you develop for one server should run on another, but porting EJBs from one EJB container to another is not as simple as it should be. Different vendors require different deployment descriptors and there's no common way of configuring data sources or other container dependencies. However coding business logic with Spring is highly portable across containers – with no changes to your code or deployment descriptors!

While Spring "makes things easier," some developers complain that it's "too heavyweight." However, Spring is really an *a la carte* framework where you can pick and choose what you want to use. The development team has segmented the distribution so you can use only the Java ARchives (JARs) you need.

How Spring Works

The *J2EE Design and Development* book illustrates Spring and how it works. Spring is a way to configure applications using JavaBeans. When I say *JavaBeans*, I mean Java classes with *getters* and *setters* (also called *accessors* and *mutators*) for its class variables. Specifically, if a class exposes *setters*, Spring configures that class. Using Spring, you can expose any of your class dependencies (that is, a database connection) with a setter, and then configure Spring to set that dependency. Even better, you don't have to write a class to establish the database connection; you can configure that in Spring too! This dependency resolution has a name: *Inversion of Control* or *Dependency Injection*. Basically, it's a technical term for wiring dependent objects together through some sort of *container*.

Spring has seven individual modules, each having its own JAR file. Below is a diagram¹ of the seven modules:

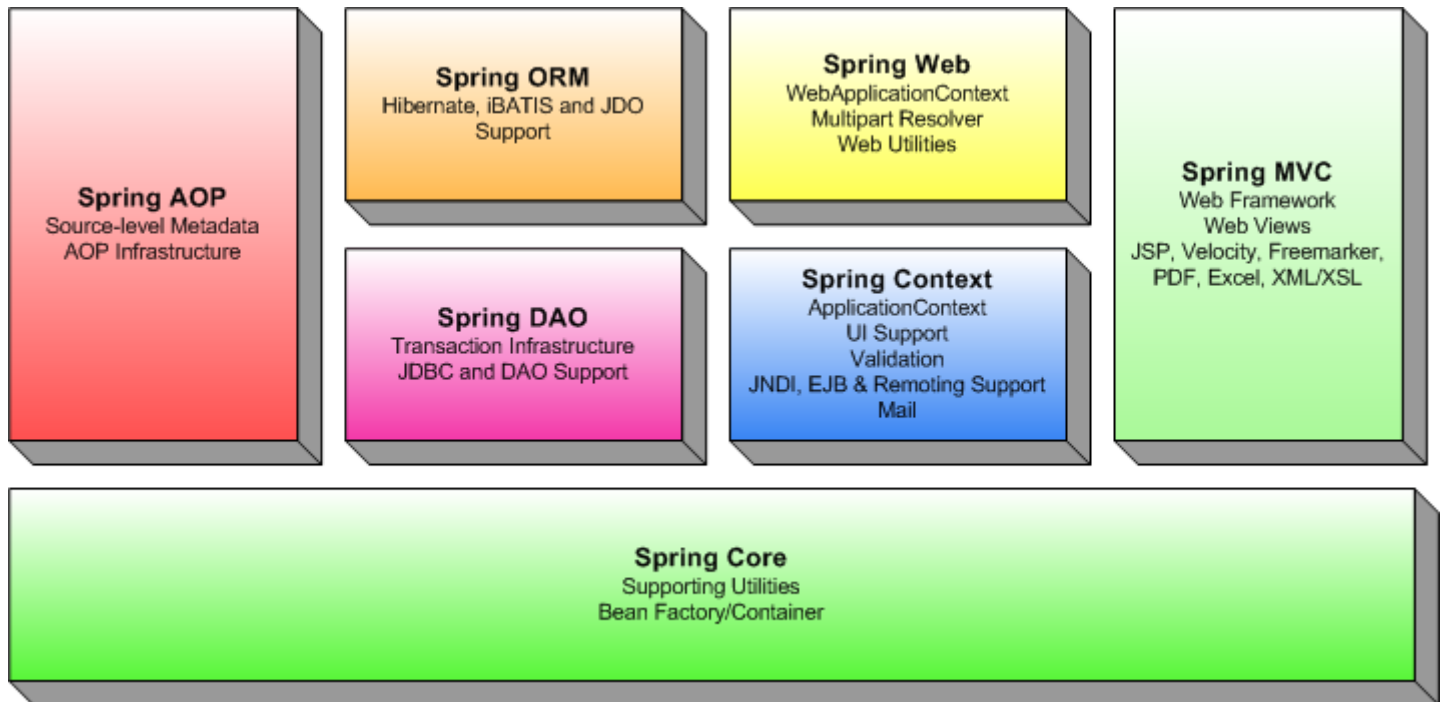


Figure 1.1: Spring's seven modules

In the MyUsers application that you will develop in *Chapter 2*, you will use several of the modules above, but not all of them. Furthermore, you'll only be using a fraction of the functionality in each module.

1. This diagram is based on [one from Spring's Reference Documentation](#).

The diagram below shows the Spring modules that MyUsers will be using.

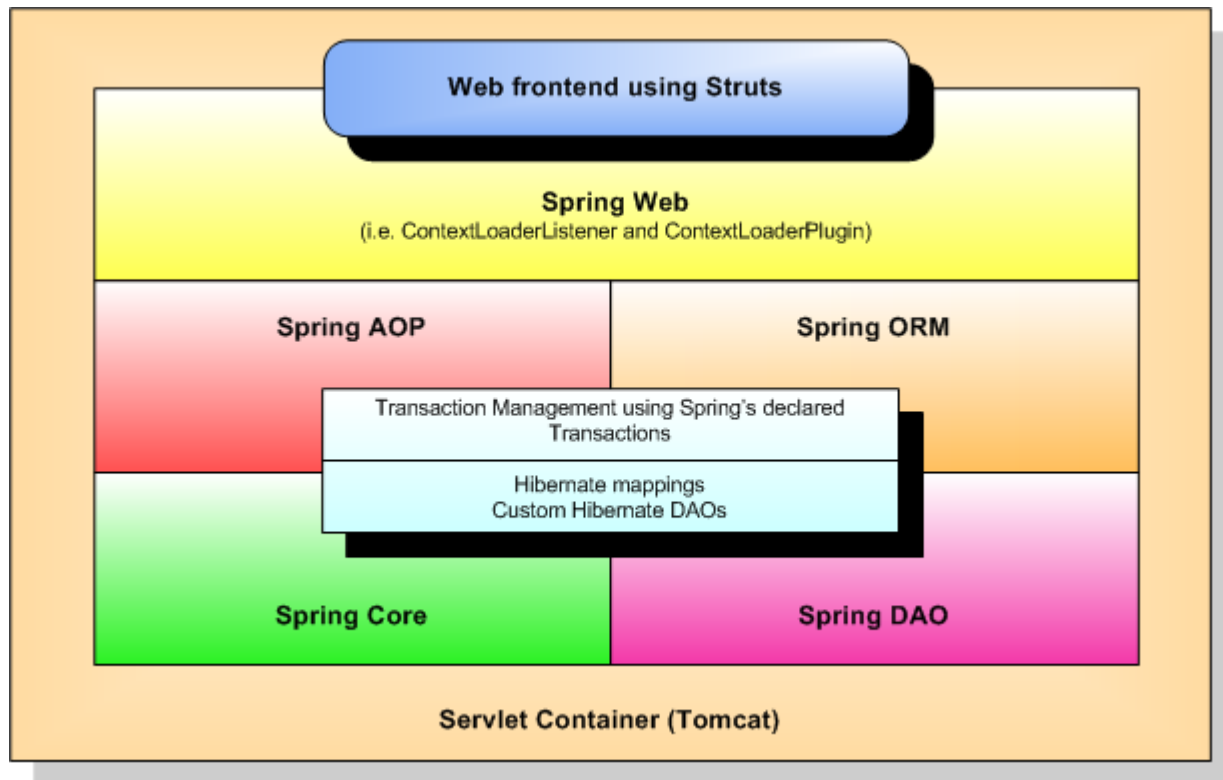


Figure 1.2: MyUsers application architecture using Spring

How Spring Makes J2EE Easier

In Figure 1.2, you can see that Spring provides a lot of pieces in the application you'll be building. At first glance, it looks intrusive and that you'll have to know a lot about Spring. Not true, my friend. In most cases, you won't even see Spring's API being used. For instance, in the middle tier, you will set up declarative transactions and set DAOs on the business delegates. Your code will not see a single import from Spring, nor any *Factory Patterns* to figure out which DAO implementation to use. You simply configure it all in an XML file - and *voila!* - clean design is yours.

The following sections talk about some of the processes that Spring simplifies.

Coding to Interfaces

Coding to interfaces allows developers to advertise the methods by which their objects will be used. It's helpful to design your application using interfaces because you gain a lot of flexibility in your implementation. Furthermore, communicating between the different tiers using interfaces promotes loose coupling of your code.

Easy Testability

Using Test-Driven Development (TDD) is the best way to rapidly produce high-quality code. It allows you to drive your design by coding your clients (the tests) before you code interfaces or implementations. In fact, modern IDEs like Eclipse and IDEA will allow you to create your classes and methods on-the-fly as you're writing your tests. Spring-enabled projects are easy to test for two reasons:

- You can easily load and use all your Spring-managed beans in a JUnit test. This allows you to interact with them as you normally would from any client.
- Your classes won't bind their own dependencies. This allows you to ignore Spring in your tests and set mock objects as dependencies.

Reducing Coupling: Factory Pattern vs. Spring

In order to have an easily maintainable and extendable application, it's not desirable to tightly couple your code to a specific resource (for example, you may have code that uses SQL functions that are specific to a database type). Of course, it's often easier to code to a specific database if the proprietary functionality helps you get your job done quicker. When you do end up coding proprietary functionality into a class, the J2EE Patterns recommend that you use a Factory Pattern to de-couple your application from the implementing class.

In general, it's a good idea to create interfaces for the different layers of your application. This allows the individual layers to be ignorant of the implementations that exist. A typical J2EE application has three layers:

- **Data Layer:** classes that talk to a database or other data storage system.
- **Business Logic:** classes that hold business logic provide a bridge between the GUI layer and data layer.
- **User Interface:** classes and view files used to compose a web or desktop interface to a user.

Figure 1.3 is a graphical representation of a typical J2EE application.

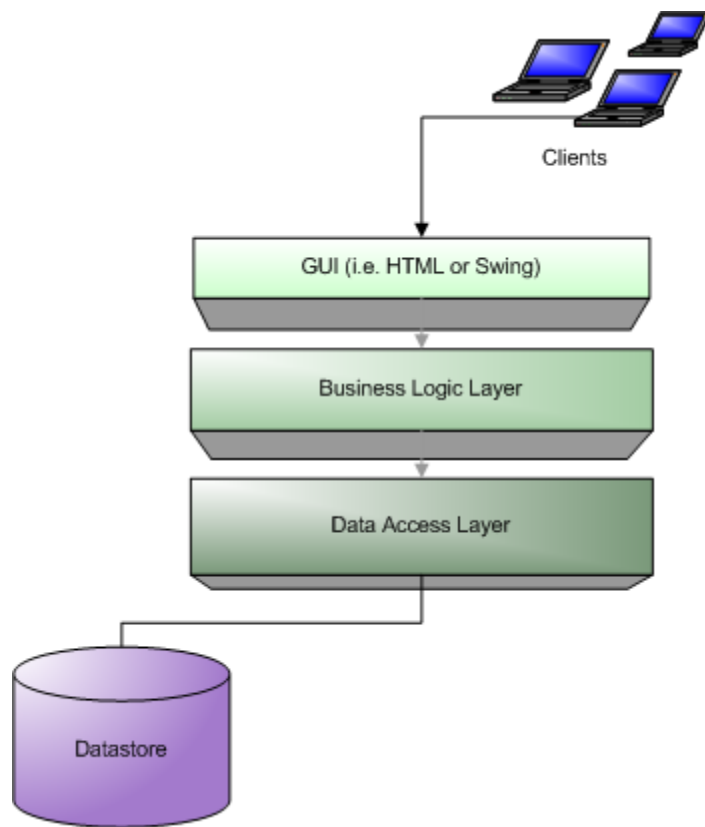


Figure 1.3: A typical J2EE application

A Factory Pattern (also known as the Abstract Factory and Factory Method from the Gang of Four Patterns) allows you to easily switch from one implementation to another by using a couple of factory classes. Typically, you'll create an abstract `DAOFactory` class and a factory class for the specific implementation (such as `DAOFactoryMySQL`). For more information, see [Core J2EE Patterns – Data Access Object](#) from the J2EE Patterns Catalog.

Configuring and Binding Class Dependencies

The Factory Pattern is a complicated J2EE pattern. Not only does it require two classes to setup, but it also introduces issues with managing dependencies of those “factored” objects. For instance, if you’re getting a DAO from the factory, how do you pass in a connection (rather than opening one for each method)? You can pass it in as part of the constructor, but what if you’re using a DAO implementation that requires a Hibernate Session? You could make the constructor parameter a `java.lang.Object` and then cast it to the required type, but it just seems ugly.

The better way is to use Spring to bind interfaces to implementations. Everything is configured in an XML file and you can easily switch out your implementation by modifying that file. Even better, you can write your unit tests so no one knows which implementation you’re using – and you can run them for numerous implementations. It works great with iBATIS and Hibernate DAO implementations. Since the test is in fact a client, it’s a great way to ensure the business logic layer will work with an alternative DAO implementation. Here’s an example of getting a `UserDAO` implementation using Spring:

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("/WEB-INF/applicationContext.xml");
UserDAO dao = (UserDAO) ctx.getBean("userDAO");
You'll have to configure the "userDAO" bean in the /WEB-INF/
applicationContext.xml file. Below is a code snippet from Chapter 2:
<bean id="userDAO"
    class="org.appfuse.persistence.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

If you want to change the implementation of `UserDAO`, all you need to change is the “class” attribute in the above XML block. It’s a cleaner pattern that you can use throughout your application. All you need to do is add a few lines to your beans definition file. Furthermore, Spring manages the Hibernate Session for this DAO for you via the “sessionFactory” property. You don’t even need to worry about closing it anymore!

Note: Spring is often referred to as a “lightweight container” because you talk to its “ApplicationContext” in order to get instantiated objects. The objects are defined in a *context file* (also called *beans definition file*). This file is simply an XML file with a number of `<bean>` elements. In a sense, it’s a “bean container” or an “object library,” where everything has been set up and is ready to be used. It’s not really a container in the traditional sense (such as Tomcat or WebLogic), but more of a “configured beans provider.”

Object/Relational Mapping Tools

Another example of Spring's usability is its first-class support for Object/Relational Mapping (ORM) tools. The first advantage of using the ORM support classes is you don't need to try/catch many of the checked exceptions that these APIs throw. Spring wraps the checked exceptions with runtime exceptions, allowing the developer to decide if he wants to catch exceptions. Below is an example of a `getUsers()` method from a `UserDAOHibernate` class without Spring:

```
public List getUsers() throws DAOException {
    List list = null;

    try {
        list = ses.find("from User u order by upper(u.username)");
    } catch (HibernateException he) {
        he.printStackTrace();
        throw new DAOException(he);
    }

    return list;
}
```

Here is an example using Spring's Hibernate support (by extending `HibernateDaoSupport`), which is much shorter and simpler:

```
public List getUsers() {
    return getHibernateTemplate()
        .find("from User u order by upper(u.username)");
}
```

From these examples, you can see how Spring makes it easier to de-couple your application layers, your dependencies *and* it handles configuring and binding a class's dependencies. It also greatly simplifies the API to use ORM tools, such as Hibernate.

Summary

This chapter discussed the history of Spring, why everyone loves it and how it makes J2EE development easier. Examples were provided comparing the traditional Factory Pattern to Spring's `ApplicationContext`, as well as a before and after view of developing with Hibernate.

Spring's `ApplicationContext` can be thought of as a “bean provider” that handles instantiating objects, binding their dependencies and providing them to you pre-configured.

Chapter 2 is a tutorial for developing a web application that uses Spring, Hibernate and Struts to manage users in a database. This application will demonstrate how Spring makes J2EE and Test-Driven Development easier. *Chapter 3* examines the Spring Core module, as well as the lifecycle of the beans it manages. This is the heart and brain of Spring – controlling how objects work together and providing them with the support they need to survive.

Spring Quick Start Tutorial

Developing Your First Spring Web Application

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.

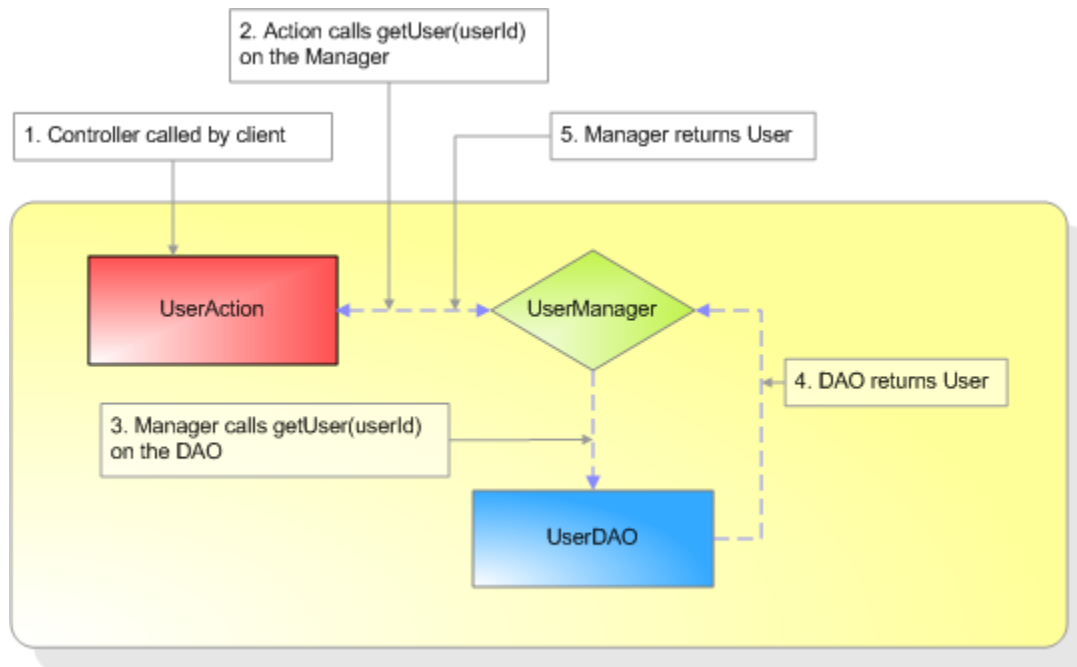


Figure 2.1: MyUsers application flow

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 Spring¹

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:

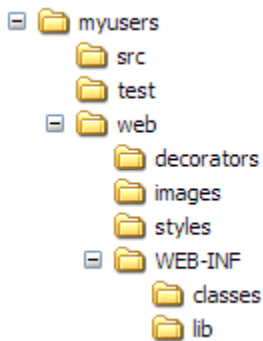


Figure 2.2: MyUsers application directory structure

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>
```

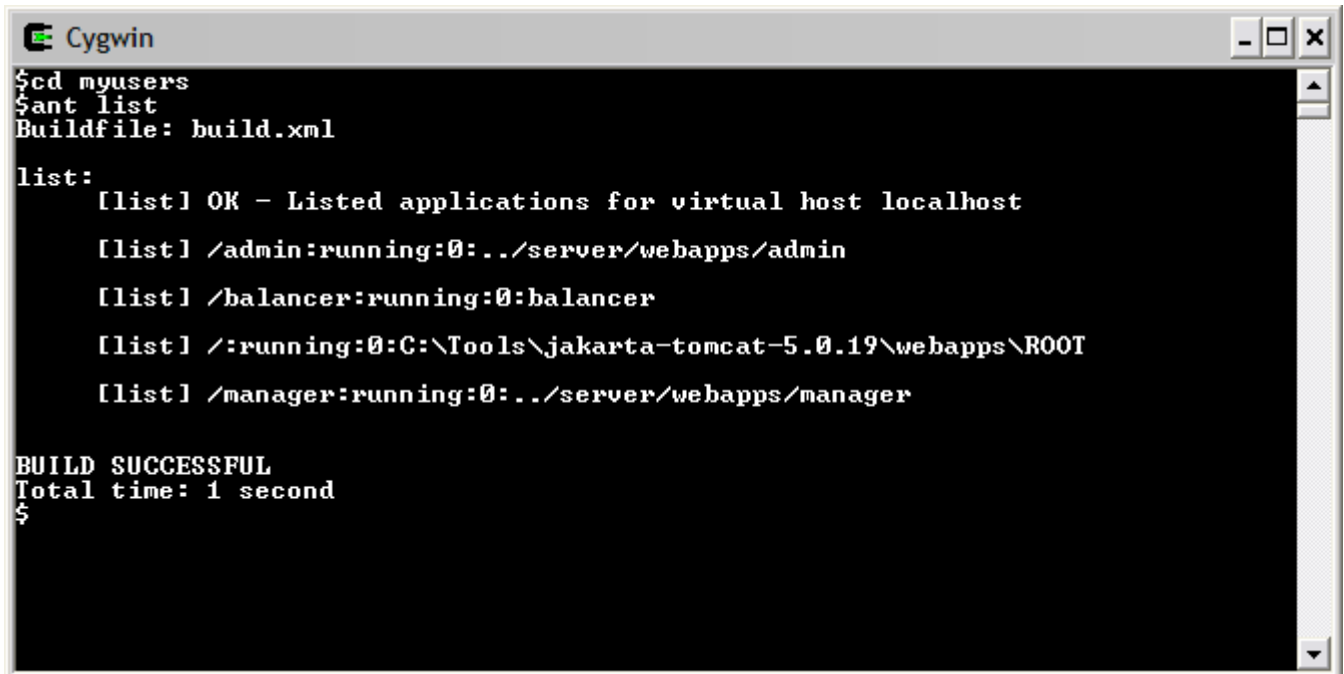
In the targets listed above, several `${tomcat.*}` variables need to be defined. These are in the *build.properties* file in the base directory. By default, they are defined as follows:

```
# 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.



```
Cygwin
$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:



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:

```
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* and *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:

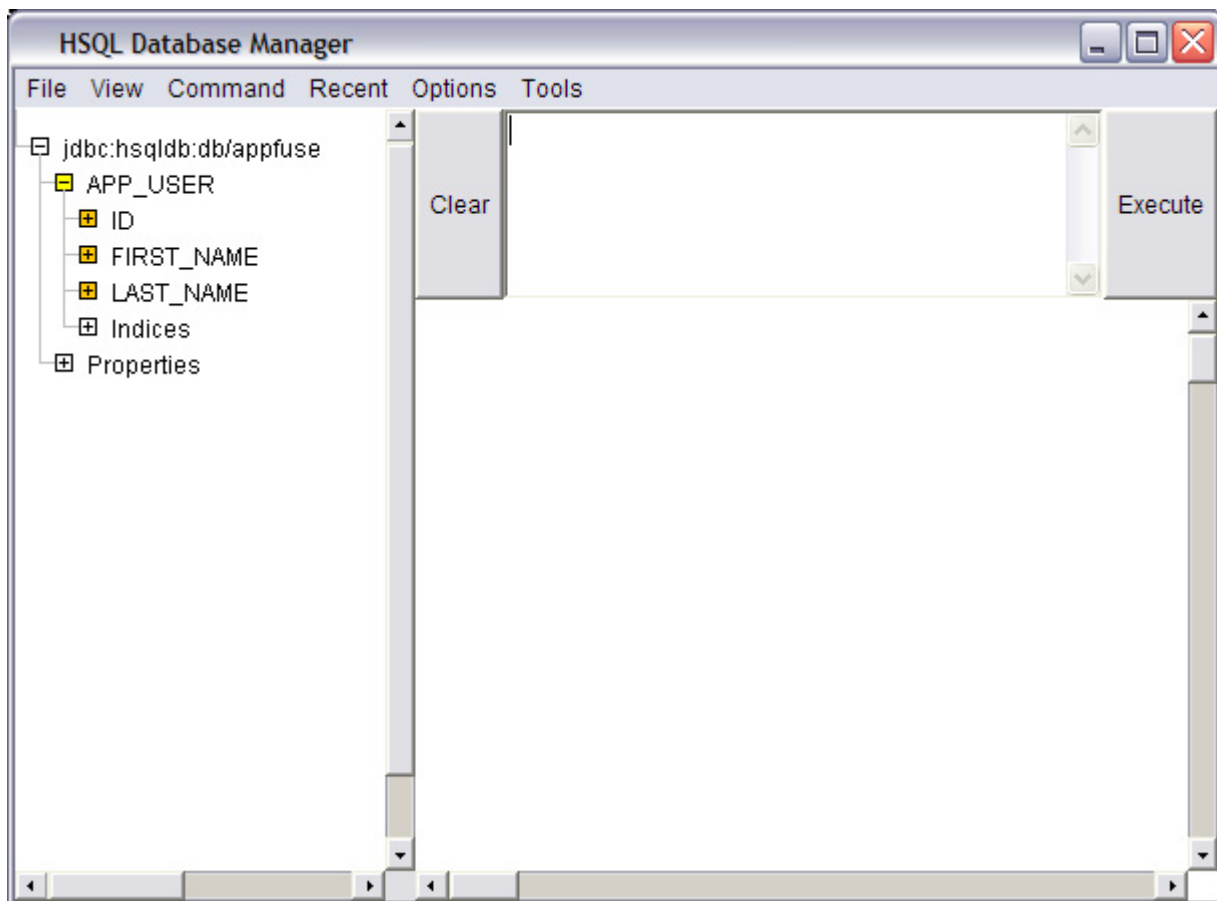


Figure 2.5: HSQL Database Manager

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 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.

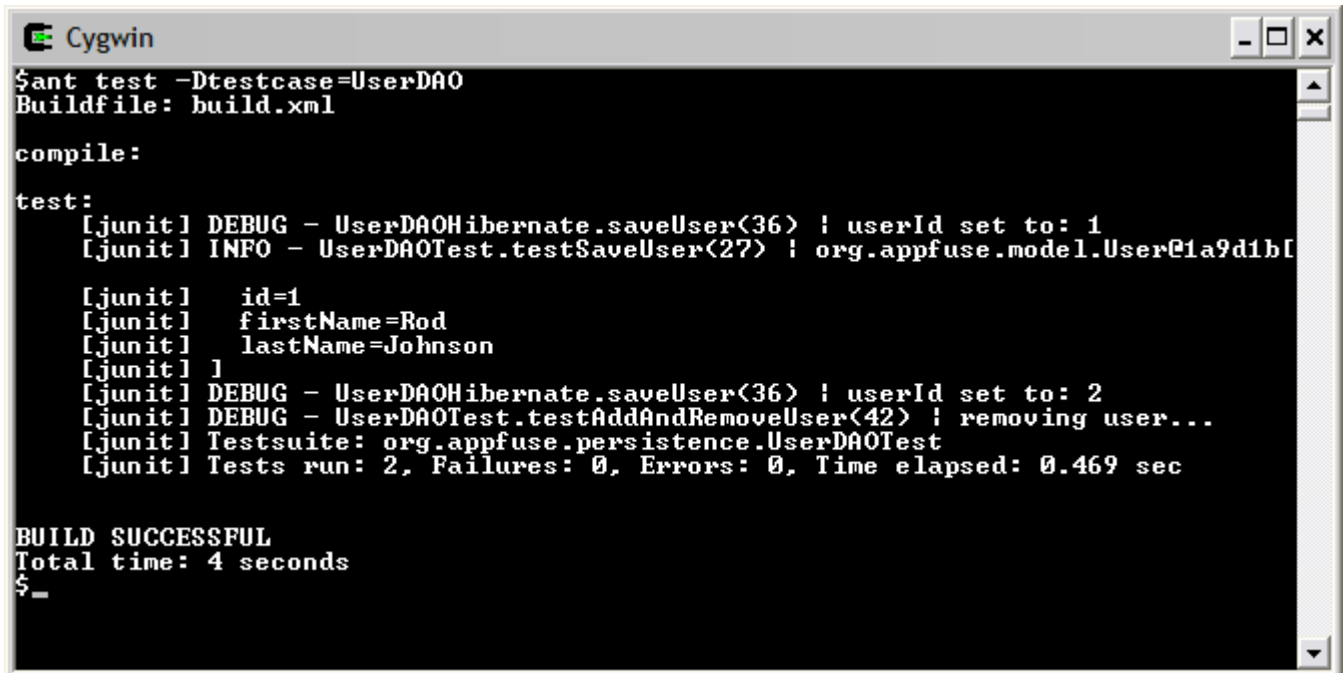
```
<bean id="userDAO"  
      class="org.appfuse.dao.hibernate.UserDAOHibernate">  
    <property name="sessionFactory">  
      <ref local="sessionFactory"/>  
    </property>  
</bean>
```

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:



```
Cygwin
$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 delegate² 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 = LoggerFactory.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.

- 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.

- 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:

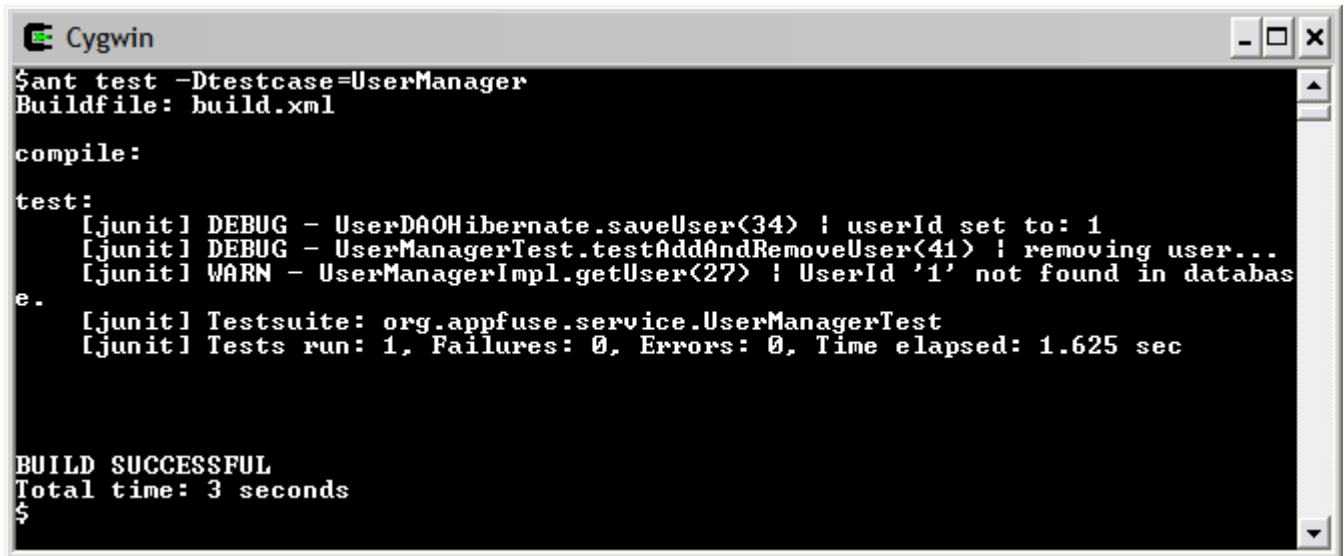
```
<bean id="userManager"
      class="org.springframework.transaction.interceptor.TransactionProxy
        FactoryBean">
  <property name="transactionManager">
    <ref local="transactionManager"/>
  </property>
  <property name="target">
    <ref local="userManagerTarget"/>
  </property>
  <property 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:



```
Cygwin
$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(2?) ! UserId '1' not found in database
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:

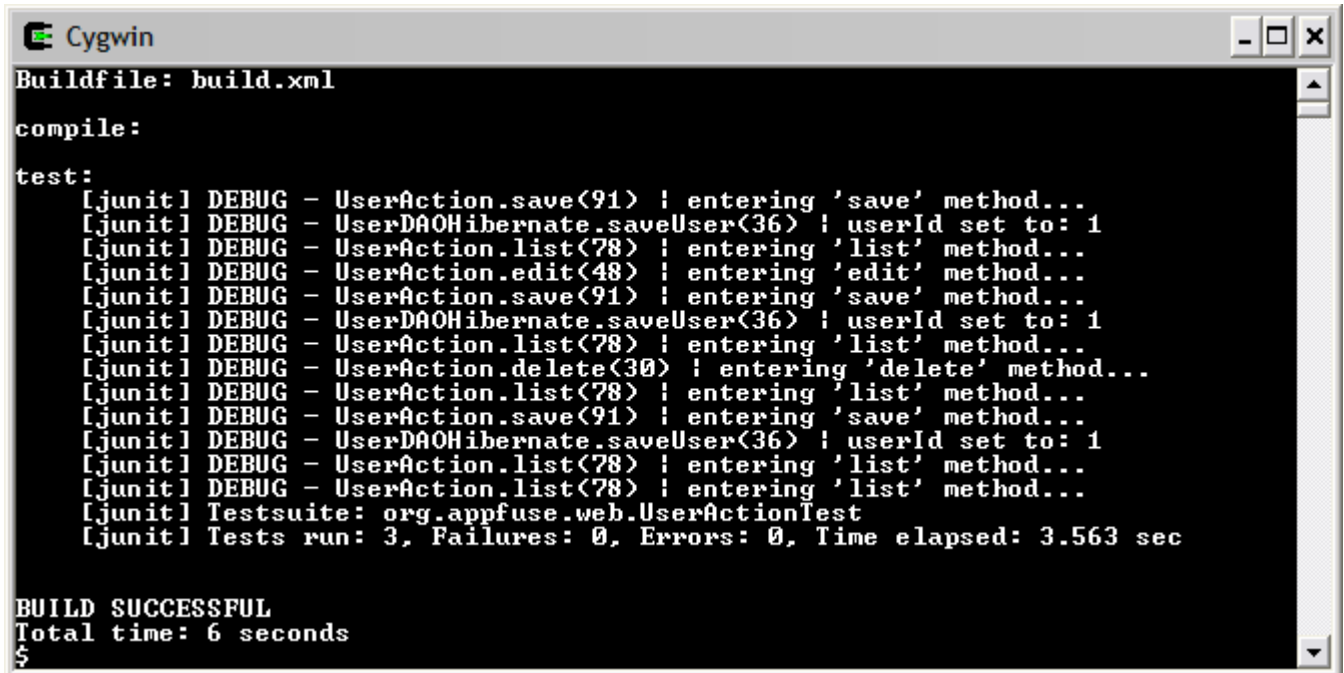
```
user.saved=User has been saved successfully.  
user.missing=No user found with this id.  
user.deleted=User successfully deleted.
```

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 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&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:

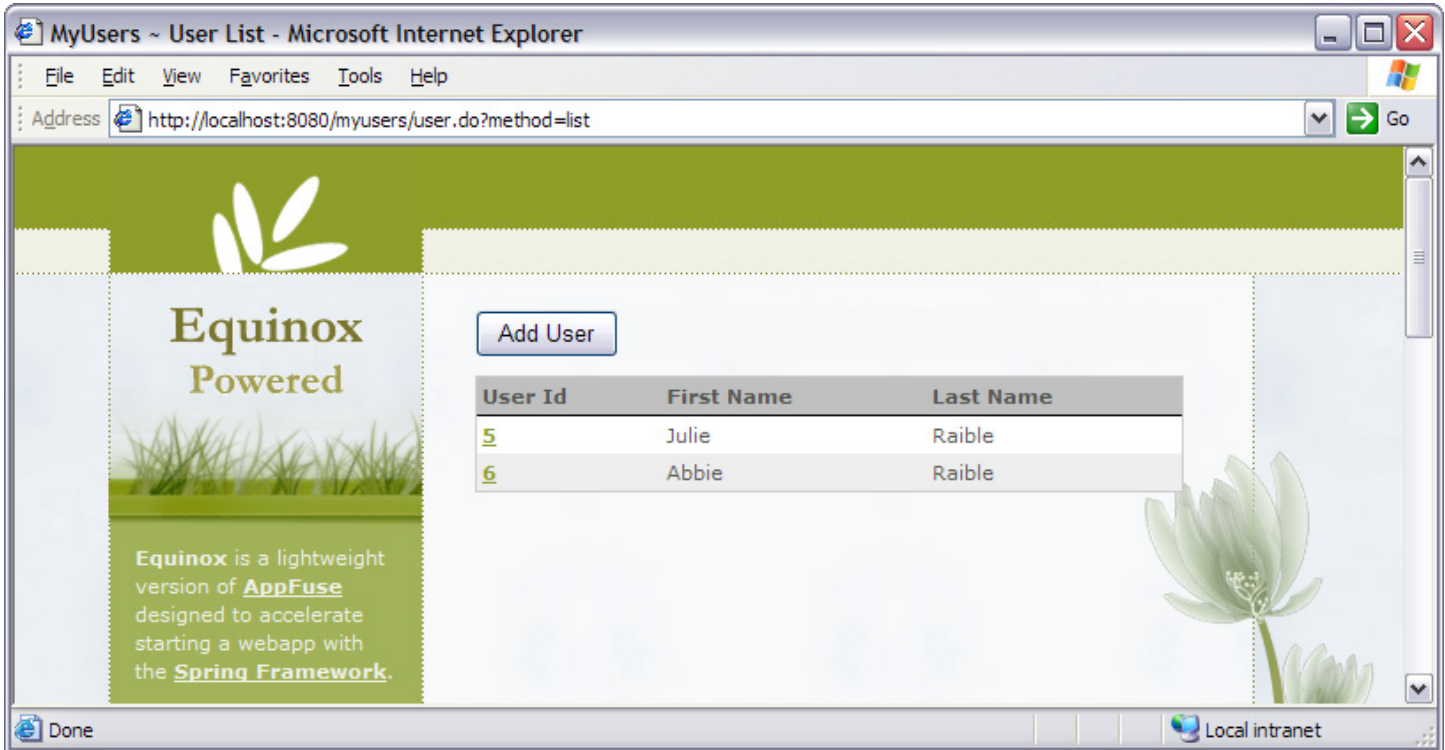


Figure 2.9: Results of `ant deploy reload` command

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.

- 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>
</tr>
</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:

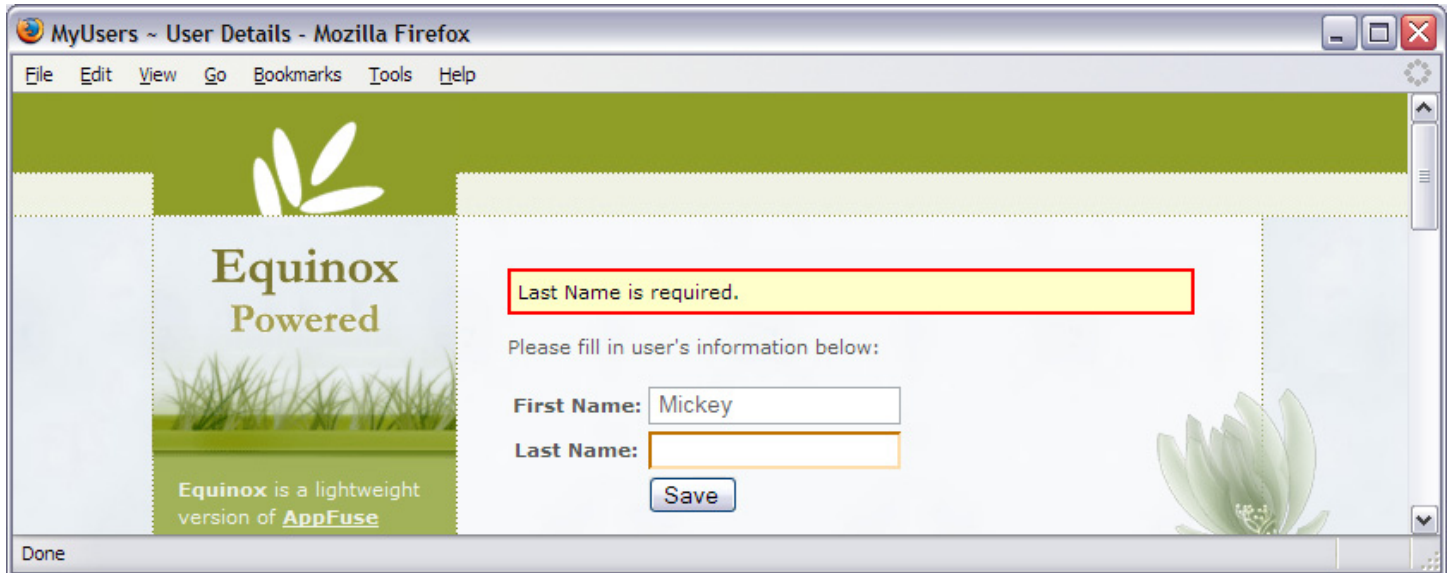


Figure 2.10: Result of the `ant deploy` command

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.

The BeanFactory and How It Works

An Introduction to the Bean Definitions, the BeanFactory and ApplicationContext

The BeanFactory represents the heart of Spring, so it's important to know how it works. This chapter explains how bean definitions are written, their properties, dependencies, and autowiring. It also explains the logic behind making singleton beans versus prototypes. Then it delves into Inversion of Control, how it works, and the simplicity it brings. This chapter dissects the Lifecycle of a bean in the BeanFactory to show how it works. This chapter also inspects the applicationContext.xml file for the MyUsers application created in Chapter 2.

Spring is an excellent tool for integrating Inversion of Control (IoC) with your application. It uses a **BeanFactory** to manage and configure beans. In most cases, however, you won't interact with the BeanFactory; you will use the **ApplicationContext**, which adds more enterprise-level, J2EE functionality, such as internationalization (i18n), custom converters (for converting Strings to Object types), and event publication/notification. This chapter explains how the BeanFactory works, IoC as it relates to the BeanFactory, how to configure beans for the BeanFactory, and how to use the ApplicationContext.

About the BeanFactory

The BeanFactory is an internal interface that configures and manages virtually any Java class. The **XMLBeanFactory** reads *bean definitions* from an XML file, while the **ListableBeansFactory** reads definitions from properties files. When the BeanFactory is created, Spring validates each bean's configuration. However, the properties of each bean are not set until the bean is created. Singleton beans are instantiated by the BeanFactory at startup time, while other beans are created on demand. According to the BeanFactory's Javadocs, "There are no constraints on how the definitions could be stored: LDAP, RDBMS, XML, properties file, etc." At the time of this writing, only XML and properties file implementations exist. Since the `XMLBeanFactory` is the most commonly used method to configure J2EE applications, this chapter uses XML for all examples.

The BeanFactory is a workhorse that initializes beans and calls their lifecycle methods. It should be noted that most lifecycle methods only apply to singleton beans. Spring cannot manage prototype (non-singleton) lifecycles. This is because, after they're created, prototypes are handed off to the client and the container loses track of it. For prototypes, Spring is really just a replacement for the "new" operator.

A Bean's Lifecycle in the BeanFactory

Figure 3.1 illustrates a bean's *lifecycle*. An outside force controls the bean, which is the Inversion of Control container. The IoC container defines the rules by which the bean operates. The rules are *bean definitions*. The bean is *pre-initialized* through its dependencies. The bean then enters the *ready* state where the beans are ready to go to work for the application. Finally, the IoC container *destroys* the bean.

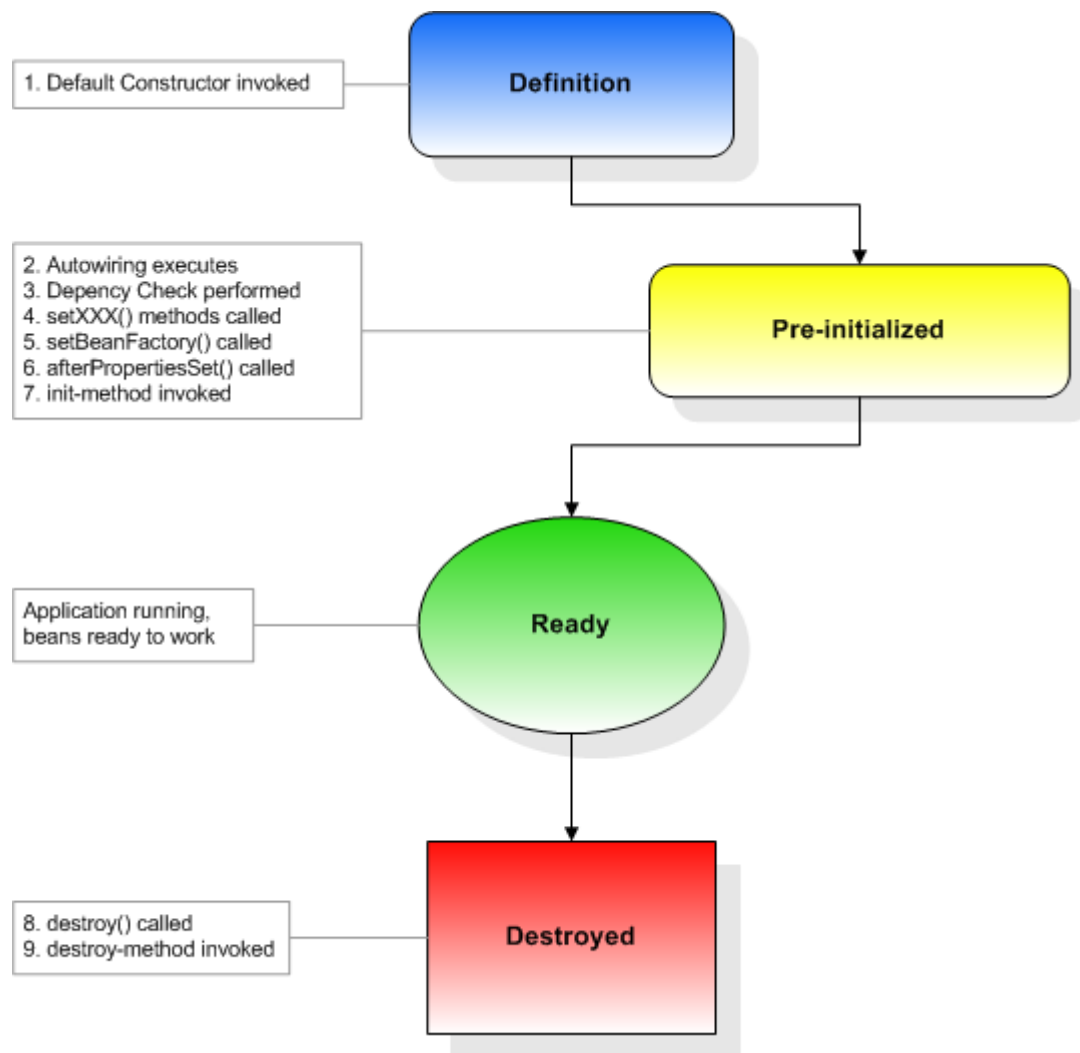


Figure 3.1: A Bean's Lifecycle in the BeanFactory

Inversion of Control

Inversion of Control is a powerful concept in application development. One form of IoC is *Dependency Injection*, [as described by Martin Fowler](#). Dependency Injection uses the Hollywood Principle: “Don’t call me, I’ll call you.” In other words, your classes don’t look up or instantiate the classes they depend on. The *control* is *inverted* and some form of container sets the dependencies. Using IoC often leads to much cleaner code and provides an excellent way to de-couple dependent classes. Dependency Injection exists in three forms:

- **Setter-based:** Classes are typically JavaBeans, with a no-arg constructor, and with *setters* for the IoC container to use when wiring dependencies. This is the variant recommended by Spring. While Spring supports constructor-based injection, a large number of constructor arguments can be difficult to manage.
- **Constructor-based:** Classes contain constructors with a number of arguments. The IoC container discovers and invokes the constructor based on the number of arguments and their object types. This approach guarantees that a bean is not created in an invalid state.
- **Getter-based (or *method injection*):** This is similar to setter-based, except you add a getter to your class. The IoC container overrides this method when it runs inside, but you can easily use the getter you specify when testing. This approach has only recently been discussed; more information is available on [TheServerSide](#).

Below is an example of a class before it’s been made IoC-ready. It is a Struts Action called `ListUsers` that depends on a `UserDAO`, which, in turn, requires a connection as part of its constructor.

```
public class ListUsers extends Action {

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        // get a connection from the database
        Connection conn = DatabaseUtils.getConnection();
        UserDAO dao = DAOFactory.createUserDAO("hibernate", conn);

        List users = dao.getUsers();

        DatabaseUtils.closeConnection(conn);
        return mapping.findForward("success");
    }
}
```

The above design is ugly. Clean it up by implementing an IoC-ready `ListUsers` class.

```
public class ListUsers extends Action {
    private UserDAO dao;
    public void setUserDAO(UserDAO userDAO) {
        this.dao = userDAO;
    }

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        List users = dao getUsers();

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

The above class doesn't contain any lookup code. This makes it clean and workable for testing. For instance, in your test you can use a Mock Object for the DAO to eliminate any dependencies on your data tier. A good IoC container allows you to wire a connection (or `DataSource`) to the `UserDAO`. With Spring you can choose to wire connections in the data layer, or use a filter to open a connection per request.

The Bean Definition Exposed

A bean definition, or `<bean>`, is really quite simple, as illustrated by the following example:

```
<bean id="example" class="org.appfuse.util.Converter"/>
```

At the very least, a bean has an *id* (or *name*) attribute and a *class* attribute. You don't have to set any properties on the bean if you don't want to, but that's similar to invoking "new" on an object. Most bean definitions contain some kind of property setting, unless they're simply used to bind interfaces to implementations (emulating the factory pattern).

The first required bean attribute is *id*. Like any good XML document, the bean definition's allowed attributes and child elements are dictated by a Document Type Definition (DTD). The DTD for Spring is appropriately named *spring-beans.dtd* and is [publicly available on the web](#). The *id* attribute of a bean is a real XML ID, which means it must be unique throughout the XML document. You can only define one *id* per bean. To add aliases to a bean, or to use illegal XML characters in a bean's *id*, specify the *name* attribute. This attribute allows one or more bean *ids*, each separated by a comma or semicolon. Using an *id* attribute is the preferred and recommended way to configure beans.

The second required bean attribute is *class*. While Spring can manage practically any Java class, the most common pattern is a default (empty) constructor with setters for dependent properties. The value specified in this attribute must be a fully-qualified class name (package name + class name). Otherwise, you can use the *parent* attribute. This can be useful if you want to duplicate a class and override its properties.

The table below lists all the attributes that you can define in a `<bean>` element.

Table 3.1: Bean Definition Attributes as defined by the *spring-beans.dtd* file

Attribute	Description	Frequency of Use
id	This XML ID element enables reference checking. To use a name that's illegal as an XML ID (such as 1 or 2), use the optional <i>name</i> attribute. If you specify neither an id nor a name, Spring assigns the class name as the id.	High
name	Use this attribute to create one or more aliases illegal as an id. Multiple aliases can be comma- or space-delimited. Use this attribute if you use the <code>ContextLoaderPlugin</code> with Struts and Spring manages your Actions.	Medium
class	The <i>class</i> and <i>parent</i> attributes are interchangeable. A bean definition must specify the fully-qualified name of the class (package name + class name), or the name of the parent bean.	High
parent	Note: A <i>child</i> bean definition that references a parent can override property values of the <i>singleton</i> attribute. It inherits all of the parent's other attributes, such as <i>lazy-init</i> and <i>autowire</i> .	Low
singleton	This attribute determines if the bean is a <i>singleton</i> (one shared instance returned by all calls to <code>getBean(id)</code>), or a <i>prototype</i> (independent instance resulting from each call to <code>getBean(id)</code>). The default value is <i>true</i> .	Low
lazy-init	If <i>true</i> , the bean will be lazily initialized. If <i>false</i> , it will get instantiated on startup by bean factories that perform eager initialization of singletons.	Low
autowire	<p>This attribute controls the <i>autowiring</i> of bean properties. If used, Spring automatically figures out the dependencies using one of the following modes:</p> <p>no: (Default) You must define bean references in the XML file using the <code><ref></code> element. Recommended to make documentation more explicit.</p> <p>byName: Autowires by property name. If a DAO exposes a <code>dataSource</code> property, Spring will try to set this to the value of the "dataSource" bean in the current factory.</p> <p>byType: Autowires if exactly one bean of the property type is in the factory.</p> <p>constructor: same as <i>byType</i> for constructor arguments.</p> <p>autodetect: chooses the constructor or <i>byType</i> through inspection of the bean class.</p> <p>Note: While this attribute reduces the size of your XML file, it reduces the readability and self-documentation supplied by declaring properties. For larger applications, autowiring is discouraged because it removes the transparency and structure from collaborating classes.</p>	Low

Table 3.1: Bean Definition Attributes as defined by the *spring-beans.dtd* file

dependency-check	<p>This attribute checks to see if all a bean's dependencies (expressed in its properties) are satisfied.</p> <p>None: no dependency checking (the default). Properties with no value specified are not set.</p> <p>simple: checks type dependencies including primitives and Strings.</p> <p>object: checks other beans in the factory.</p> <p>all: includes both of the above types.</p>	Low
depends-on	<p>This attribute names the beans that this bean depends on for initialization. The bean factory will guarantee that these beans are initialized first.</p>	Low
init-method	<p>This is a no-argument method to invoke after setting a bean's properties.</p>	Low
destroy-method	<p>This is a no-argument method to invoke on factory shutdown.</p> <p>Note: This method is only invoked on singleton beans!</p>	Low

Configuring Properties and Dependencies

The BeanFactory could be compared to EJB's lifecycle, except Spring is much simpler; you can wire up pretty much *any* class, and you don't need to extend or implement interfaces. With bare-bones EJBs, you're required to implement many lifecycle classes that you may never even use. With Spring-managed beans, you can manage the lifecycle using *init-method* and *destroy-method* attributes, and you only need to implement interfaces if you want to physically talk to the BeanFactory during the initialization process.

A bean *property* is a member variable of a class. For example, your bean might have a "maxSize" property and a method to set it:

```
private int maxSize;

public void setMaxSize(int maxSize) {
    this.maxSize = maxSize;
}
```

You can set the value of "maxSize" by using the following XML fragment on your bean's definition:

```
<property name="maxSize"><value>1000</value></property>
```

A bean *dependency* is a complex a property of a bean. The bean *depends on* this property in order to operate. Dependencies refer to other classes, rather than simple values. For instance, the "dataSource" property in the example below refers to another bean. The "dataSource" is a *dependency* of the "sessionFactory" bean, whereas the "mappingResources" is just a property with values.

```
<bean id="sessionFactory" class="...">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <property name="mappingResources">
        <list>
            <value>org/appfuse/model/User.hbm.xml</value>
        </list>
    </property>
    ...
</bean>
```

In this example, the reference to the "dataSource" bean is set using a `<ref>` tag. Let's take a closer look at this tag.

Specifying Dependencies with <ref>

The <ref> tag that points to the dataSource uses a “local” attribute to point to the “dataSource” bean. In addition to local, other options exist for pointing to dependent beans. The following list shows the available attributes for the <ref> element:

- **Bean:** Finds the dependent bean in either the same XML file or another XML file that has been loaded into the ApplicationContext.
- **Local:** Finds the dependent bean in the current XML file. This attribute is an XML IDREF so it must exist or validation will fail.
- **External:** Finds the bean in another XML file and does not search the current XML file.

From this list, <ref bean="..."> and <ref local="..."> are most commonly used. *Bean* is the most flexible option, allowing you to move beans between files, but *local* has the convenience of built-in XML validation.

In addition to specifying values from String, you can also read values from a *.properties* file using the `PropertyPlaceholderConfigurer` class. Below is an example using a *database.properties* file in the classpath.

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location">
      <value>database.properties</value>
    </property>
</bean>

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
      <value>${db.driverName}</value>
    </property>
    <property name="url"><value>${db.url}</value></property>
    <property name="username"><value>${db.username}</value></property>
    <property name="password"><value>${db.password}</value></property>
</bean>
```

You can use a special <null/> element to set a property to Java's null value. An empty value, <value></value>, will result in setting an empty String ("").

Pre-Initializing Your Beans

To pre-initialize your beans is to prepare them to work for your application. You do this by configuring their properties and dependencies. The following sections discuss the most critical steps of this process.

Autowiring

Even though autowiring is not recommended for larger applications as noted below, you may choose to use it for your smaller ones. If you choose to autowire a bean, I recommend that you use `autowire="byName"`, because it's good practice to keep the names of your setters and bean ids in synch.

For example, you could define the `UserDAO` from *Chapter 2* with autowiring, and then you wouldn't have to specify the "sessionFactory" property.

```
<bean id="userDAO" autowire="byName"
      class="org.appfuse.persistence.hibernate.UserDAOHibernate"/>
```

If you use `autowire="byType"`, Spring will look for a bean that is a Hibernate `SessionFactory`. The problem with this approach is that you may have multiple session factories talking to two databases. With *byName*, you can give those beans different names and label your setters appropriately. The *constructor* and *autodetect* options will suffer from the same affliction, since they employ *byType* under the covers.

Note: Autowiring causes you lose the self-documenting features of the XML file. Without autowiring, you'll know what a bean's dependencies are because they're specified in XML. With autowiring, you might have to look at a class's Javadocs or source to figure it out. Also, though rare, the BeanFactory could autowire some dependencies that you didn't want set.

Dependency Checks

Defining a *dependency-check* attribute in your bean's definition is useful when you want to ensure that all properties are properly set on a bean. A properly structured bean has default values. Some properties may not be needed in certain scenarios, limiting this feature's usefulness. The default is "none," meaning dependency checking is not activated, but you can enable this on a per bean basis. A "simple" verifies primitive types and set collections. An "object" value checks a bean's dependencies (also called collaborators). The "all" value includes both "simple" and "object."

setXXX()

The `setXXX()` methods are simply the setters that inject dependencies into a class. These properties are configured in the context file and can be primitives (that is, `int` or `boolean`), object types (`Long`, `Integer`), null values or references to other objects.

To demonstrate how each of these types is set, the `Smorgasbord` class below has all the previously mentioned property types:

```
package org.appfuse.model;

// organize imports with your IDE

public class Smorgasbord extends BaseObject {
    private Log log = LogFactory.getLog(Smorgasbord.class);
    private int daysToJavaOne;
    private boolean attendingJavaOne;
    private Integer streetsInDenver;
    private Long peopleInDenver;
    private DataSource dataSource;

    public void setDaysToJavaOne(int daysToJavaOne) {
        this.daysToJavaOne = daysToJavaOne;
    }

    public void setAttendingJavaOne(boolean attendingJavaOne) {
        this.attendingJavaOne = attendingJavaOne;
    }

    public void setStreetsInDenver(Integer streetsInDenver) {
        this.streetsInDenver = streetsInDenver;
    }

    public void setPeopleInDenver(Long peopleInDenver) {
        this.peopleInDenver = peopleInDenver;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public String toString() {
        log.debug(super.toString());
        return super.toString();
    }
}
```

Rather than writing a Unit Test for this class example, use the *dependency-check* attribute in the bean's definition, as well as the *init-method* attribute to call its `toString()` method.

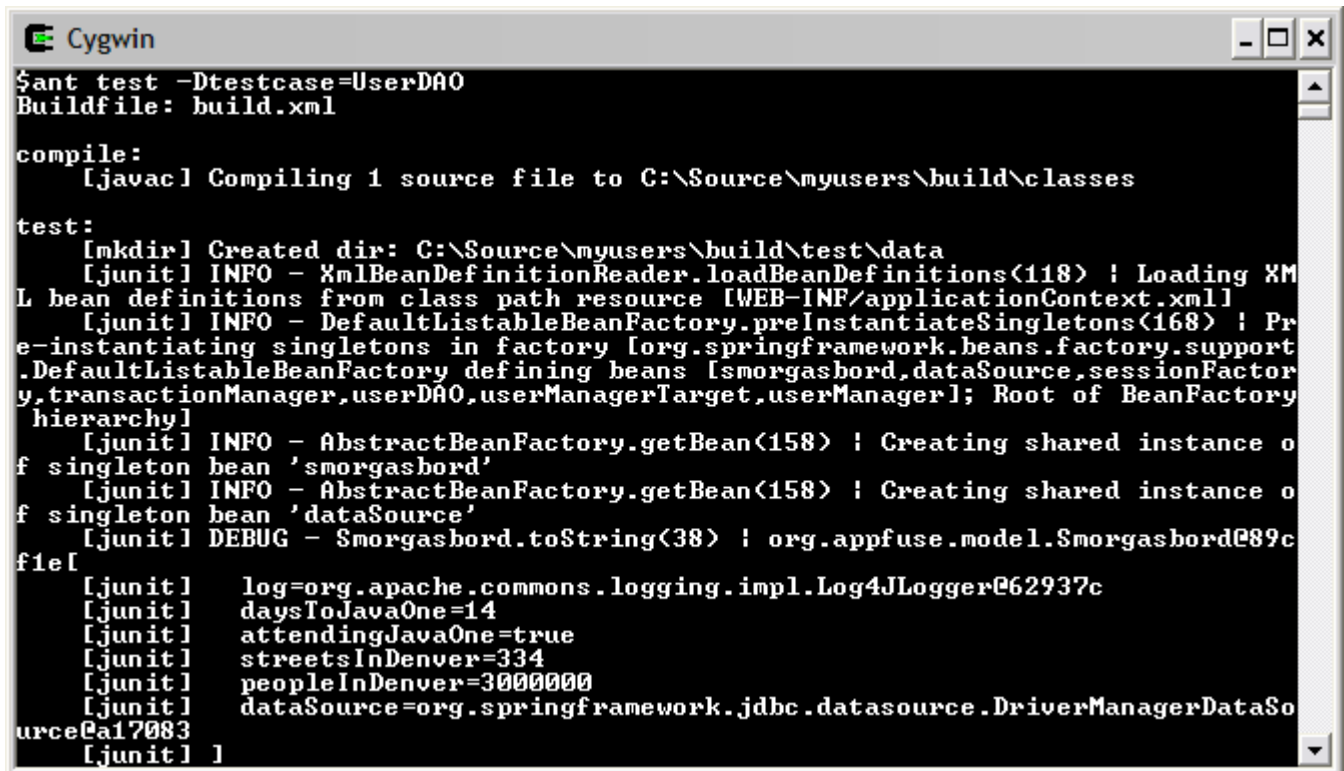
Note: For your professional applications, I strongly recommend always writing a Unit Test.

```
<bean id="smorgasbord" class="org.appfuse.model.Smorgasbord"
    dependency-check="all" init-method="toString">
    <property name="daysToJavaOne"><value>14</value></property>
    <property name="attendingJavaOne"><value>true</value></property>
    <property name="streetsInDenver"><value>334</value></property>
    <property name="peopleInDenver"><value>3000000</value></property>
    <property name="dataSource"><ref local="dataSource"/></property>
</bean>
```

You can turn on informational logging for the `org.springframework.beans` package by adding the following to `web/WEB-INF/classes/log4j.xml`:

```
<logger name="org.springframework.beans">
    <level value="INFO"/>
</logger>
```

Now if you run any tests or deploy and run `MyUsers`, you should see the following output in your console. This example uses `ant test -Dtestcase=UserDAO`.



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

compile:
[javac] Compiling 1 source file to C:\Source\myusers\build\classes

test:
[mkdir] Created dir: C:\Source\myusers\build\test\data
[junit] INFO - XmlBeanDefinitionReader.loadBeanDefinitions(118) : Loading XM
L bean definitions from class path resource [WEB-INF/applicationContext.xml]
[junit] INFO - DefaultListableBeanFactory.preInstantiateSingletons(168) : Pr
e-instantiating singletons in factory [org.springframework.beans.factory.support
.DefaultListableBeanFactory defining beans [smorgasbord,dataSource,sessionFactor
y,transactionManager,userDAO,userManagerTarget,userManager]; Root of BeanFactory
hierarchy]
[junit] INFO - AbstractBeanFactory.getBean(158) : Creating shared instance o
f singleton bean 'smorgasbord'
[junit] INFO - AbstractBeanFactory.getBean(158) : Creating shared instance o
f singleton bean 'dataSource'
[junit] DEBUG - Smorgasbord.toString(38) : org.appfuse.model.Smorgasbord@89c
file[
[junit] log=org.apache.commons.logging.impl.Log4JLogger@62937c
[junit] daysToJavaOne=14
[junit] attendingJavaOne=true
[junit] streetsInDenver=334
[junit] peopleInDenver=3000000
[junit] dataSource=org.springframework.jdbc.datasource.DriverManagerDataSo
urce@17083
[junit] ]
```

Figure 3.2: Results of the `ant test -Dtestcase=UserDAO` command

setBeanFactory()

After the initialization methods are called, the BeanFactory checks for classes implementing the `BeanFactoryAware` and `BeanNameAware` interfaces. These interfaces provide a means for beans to find out more information about where they came from and who they are. The `BeanFactoryAware` interface defines one method:

```
public void setBeanFactory(BeanFactory beanFactory)
    throws BeansException;
```

If you implement this interface, it references to the BeanFactory, which you can use to look up other beans. It basically documents a bean's origins.

In order for a bean to discover its "id", use the `BeanNameAware` interface. This interface has a single method:

```
public void setBeanName(java.lang.String name);
```

In most cases, you won't need access to the BeanFactory because you can talk to other beans by wiring them as dependencies (using `<ref bean="..." />`). Accessing the bean's name may be helpful if you want to configure the same class as two different beans with different dependency implementations. Using this, you can perform conditional logic based on which "name" is configured.

After calling methods from the `BeanFactoryAware` and `BeanNameAware` interfaces, beans enter into a *ready* state. This is when your application has completed starting up (in Tomcat, for example).

afterPropertiesSet()

You can configure your beans for post-initialization processing using one of two approaches: 1) use the “init-method” attribute as illustrated in the example above, or 2) implement `InitializingBean` and its `afterPropertiesSet()` method. (The diagram shows both methods, but only one is required.) Clearly, using “init-method” is a much cleaner and simpler way to do this. However, implementing `InitializingBean` can be helpful for testing when you’re not using Spring to manage your beans. For instance, you can call this method in your tests and verify that your mock objects have been set correctly. It’s also useful for guaranteeing that your bean will be configured correctly; you’re not depending on someone to write the bean’s definition correctly.

The previous example injected property values into the `Smorgasbord` class. It set primitive values, object values and even a reference (`dataSource`) to another bean in the factory. Not only does Spring’s DTD support simple `<value>` elements in properties, it also supports setting Properties, Lists and Maps. Below is an example ([from Spring’s documentation](#)) of using these more complex properties:

```
<!-- results in a setPeople(java.util.Properties) call -->
<property name="people">
  <props>
    <prop key="HarryPotter">The magic property</prop>
    <prop key="JerrySeinfeld">The funny property</prop>
  </props>
</property>
<!-- results in a setSomeList(java.util.List) call -->
<property name="someList">
  <list>
    <value>a list element followed by a reference</value>
    <ref bean="dataSource"/>
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry key="yup an entry">
      <value>just some string</value>
    </entry>
    <entry key="yup a ref">
      <ref bean="dataSource"</ref>
    </entry>
  </map>
</property>
```

init-method

The *init-method* attribute of a bean definition calls a method after all the properties of a bean have been set. This has the same functionality as implementing the `InitializingBean` interface, except that it doesn’t tie your bean to Spring.

Ready State

After your beans have been pre-initialized and any *setup* methods have been called, they enter into a ready state. The ready state means that your application can get these beans and use them as needed. The entire lifecycle to enter the ready state is very quick. It increases based on the number of beans in your app. However, it only occurs at startup.

Destroying Beans

When you shut down (or reload) your application, beans that are singletons once again get lifecycle methods called on them. First, beans that implement the `DisposableBean` interface will have their `destroy()` methods called. Next, beans with a “destroy-method” specified in their bean definitions will have that method invoked.

The ApplicationContext: Talking to Your Beans

Understanding the BeanFactory is important when developing with Spring, but you probably won't need to interface with it in your application. In most cases, you'll use the ApplicationContext, which adds more enterprise-level, J2EE functionality, such as internationalization (i18n), custom converters (for converting Strings to Object types) and event publication/notification. An ApplicationContext is instantiated with bean definition files and beans can be easily retrieved using `context.getBean("beanId")`. Instantiating the ApplicationContext and loading the bean definition XML files is the hardest part, so let's look at the different ways to do this.

Get That Context!

Spring gives you many options for loadings its bean definitions. In the current release (1.0.2), bean definitions must be loaded from files. You could also implement your own ApplicationContext and add support for loading from other resources (such as a database). While many *Contexts* are available for loading beans, you'll only need a few, which are listed below. The others are internal classes that are used by the framework itself.

- **ClassPathXmlApplicationContext:** Loads context files from the classpath (that is, `WEB-INF/classes` or `WEB-INF/lib` for JARs) in a web application. Initializes using a new `ClassPathXmlApplicationContext(path)` where *path* is the path to the file. The *path* argument can also be a String array of paths. This is a good context for using in unit tests.
- **FileSystemXmlApplicationContext:** Loads context files from the file system, which is nice for testing. Initializes using a new `FileSystemXmlApplicationContext(path)` where *path* is a relative or absolute path to the file. The path argument can also be a String array of paths.
- **XmlWebApplicationContext:** Loads context files internally by the `ContextLoaderListener`, but can be used outside of it. For instance, if you are running a container that doesn't load Listeners in the order specified in `web.xml`, you may have to use this in another Listener. Below is the code to use this Loader.

```
XmlWebApplicationContext context =  
    new XmlWebApplicationContext();  
context.setServletContext(ctx);  
context.refresh();
```

Once you've obtained a reference to a *context*, you can get references to beans using `ctx.getBean("beanId")`. You will need to cast it to a specific type, but that's the easy part. Of the above contexts, `ClassPathXmlApplicationContext` is the most flexible. It doesn't care where the files are, as long as they're in the classpath. This allows you to move files around and simply change the classpath.

Tips for Unit Testing and Loading Contexts

Writing unit tests with Spring is generally pretty easy. Spring-ready beans can be instantiated and tested sans-container with mocks put into the setters. Testing is also easy because of Spring's interface-based design, which allows you to choose how to test implementing classes. Testing with the least amount of setup can be achieved by using a `ClassPathXmlApplicationContext`, getting a reference to your bean, and calling methods on it. This method allows you to write your tests to interfaces, not to implementation classes. If you decide to swap out implementations (by changing the “class” attribute of your bean), you don't have to change anything in your test.

However, one issue with using this is the `ApplicationContext` can take a few seconds to initialize. As your application grows, the time-to-initialize will increase. Furthermore, if you load the context in a `setUp()` method, the context will be instantiated each time before a `testXXX()` method is called. Luckily, a couple of simple solutions exist to load the context only once per `TestCase`. The first is to use JUnit's `TestSetup` class in a suite, or you can put the context-loading code in a static block of your test.

```
protected static ApplicationContext ctx = null;

static {
    ctx = new ClassPathXmlApplicationContext("/appContext.xml");
}
```

The second solution is to directly test the implementation classes, without ever using Spring's `BeanFactory` or `ApplicationContext`. This generally involves creating an instance with the “new” operator, setting its dependencies manually, and invoking methods to test. Using this technique allows you to replace dependent classes with mock objects, which can speed up your tests and isolate them from their environment dependency.

More information on unit testing will be given in *Chapter 8: Unit Testing with Spring*.

Internationalization and MessageSource

Internationalization, or i18n, is an important concept in application development, particularly in web applications. It's likely that your users will originate from other countries and will speak different languages. They'll probably have their browsers set to show sites in their native language first.

The ApplicationContext interface extends the MessageSource interface, which gives it messaging (i18n) functionality. In conjunction with the NestingMessageSource, capable of hierarchical message resolving, these are the basic interfaces Spring provides to resolve messages. When loading an ApplicationContext, it searches for a bean with the name "messageSource" defined in its context. If no such bean is found in the current or parent contexts, a StaticMessageSource will be created so that getMessage() calls don't fail.

Two MessageSource implementations exist in the current Spring code base. They are ResourceBundleMessageSource (which reads from a *.properties* file) and StaticMessageResource (hardly used, but allows for adding messages programmatically). Below is an example "messageSource" bean definition that will load *messages.properties* from the classpath:

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessage
        Source">
  <property name="basename"><value>messages</value></property>
</bean>
```

If you want to specify multiple ResourceBundles, you can set the "basenames" property instead of the single-value "basename" property.

```
<property name="basenames">
  <list>
    <value>messages</value>
    <value>errors</value>
  </list>
</property>
```

The next chapter defines a "messageSource" bean and interacts with it to get error and success messages.

Event Publishing and Subscribing

The `ApplicationContext` supports Event Handling via the `ApplicationEvent` class and `ApplicationListener` interface. If you'd like to use this functionality, you can implement `ApplicationListener` in a bean and when an `ApplicationEvent` is published to the context, your bean will be notified. See the Spring Events table for the three standard events.

Table 3.2: Spring Events

Event	Description
<code>ContextRefreshedEvent</code>	Event published when the <code>ApplicationContext</code> is initialized or refreshed. Initialized here means that all beans are loaded, singletons are pre-instantiated and the <code>ApplicationContext</code> is ready for use.
<code>ContextClosedEvent</code>	Event published when the <code>ApplicationContext</code> is closed, using the <code>close()</code> method on the <code>ApplicationContext</code> . Closed here means that singletons are destroyed.
<code>RequestHandledEvent</code>	A web-specific event telling all beans that a HTTP request has been serviced (this will be published after the request has been finished). Note that this event is only applicable for web applications using Spring's <code>DispatcherServlet</code> .

You can also implement custom events by calling the `publishEvent()` method on the `ApplicationContext`. See Spring's [Reference Documentation](#) for an example.

A Closer Look at MyUser's applicationContext.xml

In the MyUsers application from *Chapter 2*, you loaded your bean definitions from `web/WEB-INF/applicationContext.xml`. In this file, several beans are defined. Of the seven beans defined in this file, only three of them (`userDAO`, `userManagerTarget` and `/user`) refer to classes that you created. This really shows the power of Spring: four of the classes you used (`dataSource`, `sessionFactory`, `transactionManager` and `userManager`) are internal Spring classes upon which you set properties.

Now that you understand how a bean definition XML file is composed, I encourage you to take a closer look at the `applicationContext.xml` file from MyUsers. I think you will notice that it's rather simple and easy to comprehend.

Summary

In this chapter, you learned about Inversion of Control, which has recently aliased as Dependency Injection by Martin Fowler. Injecting dependencies using a container like Spring is a clean and powerful way to configure applications and reduce coupling.

The BeanFactory and Bean Definitions are the driving force behind Spring's IoC container, allowing you to specify dependencies and control your class's lifecycles in XML. Knowing how the BeanFactory works and how bean definitions are specified will help you to become an extremely efficient developer with Spring. Knowing how properties are set – whether they're Strings, Objects, or references to other beans – is a tremendous asset. You can also define more complex properties like Properties, Lists and Maps. This is where you will be wiring up your entire application, rather than in code itself. Now, your code is loosely coupled, and can take care of its concerns, and you can think of the ApplicationContext/BeanFactory as the container that couples everything together.

In the next chapter, you will convert the MyUsers application from *Chapter 2* to use Spring's MVC framework. I think you'll be amazed at how simple web development with Spring is, once you've gotten the internals set up and configured.

Spring's MVC Framework

A Web Framework with a Lifecycle

This chapter describes the many features of Spring's MVC framework. It shows you how to replace the Struts layer in MyUsers with Spring. It covers the DispatcherServlet, various Controllers, Handler Mappings, View Resolvers, Validation and Internationalization. It also briefly covers Spring's JSP Tags.

This chapter covers only what you need to know to develop a simple webapp with validation, including the following topics:

- Unit Testing Spring Controllers
- Converting Struts to Spring MVC
- Modify *web.xml* to use `DispatchServlet` instead of `ActionServlet`.
- Create Unit Test for `UserController` (used to display a list of users).
- Create `UserController` and configure *action-servlet.xml* for `DispatchServlet`.
- Modify *userList.jsp* to work with Spring.
- Create Unit Test for `UserFormController` (edits, saves and deletes users).
- Create `UserFormController` and configure it in *action-servlet.xml*.
- Modify *userForm.jsp* to use Spring's JSP Tags.
- Configure Common Validator for Spring.
- `SimpleFormController`: Method Lifecycle Review
- Spring's JSP Tags

Overview

Chapter 3 explored Spring's BeanFactory and its lifecycle, which controls how your beans are invoked and used. Spring carries this concept into the web tier in its MVC framework. In popular frameworks like Struts and WebWork, controllers usually contain a single method: `execute()`. The framework, regardless of whether you send a GET or POST request, calls this method. It's up to the developer to code any logic needed in this method; for example, you may populate drop downs, handle validation errors and set up the view to add a new record. You can code multiple methods in a Struts or WebWork Action and then dispatch to them based on request parameters or button names. But it doesn't change the fact that the method call doesn't care which request method (GET vs. POST) you used.

Spring's MVC is a bit friendlier. The simplest way to look at it is that it offers two controllers: a `Controller` interface and a `SimpleFormController` class. The `Controller` is best suited for displaying read-only data (such as list screens), while the `SimpleFormController` handles forms (such as edit, save, delete). The `Controller` interface shown below is quite simple, containing a single `handleRequest(request, response)` method:

```
package org.springframework.web.servlet.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the
     * DispatcherServlet will render. A null return is not an error:
     * It indicates that this object completed request processing
     * itself, thus there is no ModelAndView to render.
     */
    ModelAndView handleRequest(HttpServletRequest request,
                               HttpServletResponse response)
        throws Exception;
}
```

The `handleRequest()` method returns a [ModelAndView](#) class. This class holds both the model and the view, which are both very distinct. The *model* is the information that you intend to display, and the *view* is the logical name where you want to display it. The model can be a single object with a name, or it can be a `java.util.Map` containing several objects. The view can be a [View](#) object (which is an interface for the different view types), or it can be a String name determined by a [ViewResolver](#). The rich set of Views available in Spring is discussed meticulously in *Chapter 6*.

The `SimpleFormController`, on the other hand, is a concrete class with several methods invoked while processing a data entry form. The reason one is an interface and the other super-class is primarily for flexibility. All Controllers use the `Controller` interface in Spring, whereas the `SimpleFormController` is an implementation with default settings for many of its methods. If you don't need all the rich functionality of a `FormController`, you can extend the `AbstractCommandController` to populate your command beans from an `HttpServletRequest`. Spring's MVC has a deep hierarchy for its `FormControllers` that is out of the scope of this chapter. In most cases, you simply won't need them and `SimpleFormController` will fulfill most requirements.

With `SimpleFormController`, GET requests call certain methods, while POST requests call others. This corresponds with how most web applications work: a GET signifies an "edit," while a POST signifies a "save" or "delete." This allows for easy isolation of the two operations. In Struts, you can achieve similar functionality using a `DispatchAction` or one of its subclasses. You used a `DispatchAction` in *Chapter 2* to separate the different CRUD operations into different methods. Spring's approach is better; you can re-use methods for all CRUD actions. This chapter discusses the different `SimpleFormController` methods (and when they're called) in the Method Lifecycle Review section towards the end.

Note: With Struts, I use one Action for deleting, editing, saving and listing rows in a database table. By "listing," I mean the process of retrieving all the rows from a particular table. This satisfies most of my needs in web applications. With Spring, I use two controllers for my master/detail screens. Rather than having one Controller do all the work, it's simpler to create a controller for the listing (master) and another for the delete/edit/save (detail).

If you don't feel like creating a new Controller for each list screen, you can create a [MultiActionController](#) that has separate methods for each list screen.

A later chapter will conduct a detailed analysis of Spring MVC versus the more popular MVC frameworks available: Struts, WebWork, Tapestry and JSF. It will explain the strengths and weaknesses of each, as well as demonstrate how Spring's middle tier can integrate with each of them.

Unit Testing Spring Controllers

When I first started working with Spring's MVC framework, I found it somewhat difficult to test. I was surprised, because one of Spring's advertised benefits is "Applications built using Spring are very easy to unit test."¹ While it was easy to test Controllers (those classes that drive list screens), it was a bit more difficult to test `SimpleFormControllers`. The main problem was that none of the recommended solutions (such as [Mock Objects](#)) had APIs to handle the things you normally do in a webapp: setting request parameters/attributes, grabbing data from application scopes, etc. With Struts, I'm able to use [StrutsTestCase](#), which does a very nice job of providing mock implementations of most Struts and Servlet API classes.

Because Spring is open-source, I was able to dig in and see what the developers were using internally to test the Controllers. It turned out they had a number of home-grown Mocks, which covered most of the Servlet APIs that I needed. Shortly after discovering that, the Spring team cleaned up these mocks for public consumption and added them to the Spring distribution. You'll be using these classes when you write your unit tests. If you'd like to use similar mocks in your project, be sure to include *spring-mock.jar* in your classpath.

The next section discusses how to convert the web layer of MyUsers from Struts to Spring MVC.

1. From Rod Johnson's [Introducing the Spring Framework](#) on TheServerSide.com.

Converting Struts to Spring MVC

Spring's MVC framework is similar to Struts in that it uses a single instance of a controller by default. You can change your Controllers to create new instances for every request by adding `singleton="false"` to your controller's bean definition. This way, if you prefer WebWork's "new action per request," you can still get that functionality.

Spring MVC has a single servlet that handles all requests, similar to most Java web frameworks. It's called the `DispatcherServlet` and is responsible for "dispatching" request to handlers, which have mappings to tell it where to go next. To begin working with Spring MVC, first configure the MyUsers application to use the `DispatcherServlet` for its front controller, rather than Struts' `ActionServlet`.

If you've completed the Quick Start tutorial in *Chapter 2*, you should have no problems with the following set of instructions. If you have not completed the Quick Start tutorial, but you'd like to do this one, [download the completed project after Chapter 2](#).

Modify *web.xml* to Use Spring's `DispatcherServlet`

At this point, you should have the “myusers” project setup on your hard drive.

1. To begin, open *web/WEB-INF/web.xml* and modify the “action” servlet's `<servlet-class>` from:

```
<servlet-class>
    org.apache.struts.action.ActionServlet
</servlet-class>
```

to:

```
<servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>
```

In addition, change the action's `<servlet-mapping>` from `*.do` to `*.html`. You're serving up HTML, so it makes sense to use this instead of `.do?`. Also, there's no point in advertising the web framework you're using.

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

In *Chapter 2*, you used the Spring Plugin for Struts ([ContextLoaderPlugin](#)) to load the bean configuration files. This won't work if you're using Spring MVC, so use the [ContextLoaderListener](#) to load them. This class is a `ServletContextListener` that “listens” for application events, namely the startup and shutdown of your application.

Note: This Listener will only work with Servlet 2.3 containers, so if you're on an older container, use the [ContextLoaderServlet](#).

2. To begin configuring the `ContextLoaderListener` for MyUsers, add the following XML fragment to *web.xml*, directly after the “sitemesh” filter and before its `<filter-mapping>`.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/applicationContext.xml /WEB-INF/action-servlet.xml
    </param-value>
</context-param>
```

Notice that the paths to the two files are space-delimited. These paths can also be comma-delimited.

3. Now make sure the `ContextLoaderListener` is defined in *web.xml*. It should immediately follow the “sitemesh” `<filter-mapping>`.

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

Now you're ready to begin developing your Controllers.

Create Unit Test for UserController

To practice Test-Driven Development (TDD), start by writing a unit test for the `UserController`. This will replicate the functionality of the `UserAction.search()` method you created in *Chapter 2*. Basically, it returns a list of all the users from a business delegate (`UserManager`). If you're not familiar with TDD, here's a good definition from [Dave Thomas's weblog](#):

“For me, **test-driven** development is an important way of thinking about coding. It's about using tests to gain perspective on your design and implementation. You listen to what the tests are telling you, and alter to code accordingly. Finding it hard to test something in isolation? Refactor your code to reduce coupling. Is it impossible to mock out a particular subsystem? Look at adding facades or interfaces to make the separation cleaner. Tests drive the design, and tests verify the implementation.”

1. To begin creating your unit test (*UserControllerTest.java*), first remove the other Struts-specific things you added in *Chapter 2*. Remove `UserAction` and `UserActionTest` Struts classes, as well as a few Struts JARs in **web/WEB-INF/lib**. Here are a couple commands to accomplish this quickly:

```
rm src/org/appfuse/web/UserAction.java
rm test/org/appfuse/web/UserActionTest.java
rm web/WEB-INF/lib/struts*
rm web/WEB-INF/struts-config.xml
```

2. Now remove the definition for the `UserAction` class from *action-servlet.xml*. Delete the following lines from the file:

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

3. To add support for unit testing controllers, add *spring-mock.jar* to **web/WEB-INF/lib**. At the same time, put *spring-sandbox.jar* in the same directory. This will be used for its Commons Validator support (to be discussed later in this chapter). You can download both of these JARs from <http://sourcebeat.com/downloads>.
4. To create a JUnit Test for the Controller, create a *UserControllerTest.java* file in **test/org/appfuse/web**. This class should extend `junit.framework.TestCase` and have a `setUp()` method defined to load the context files using an `XmlWebApplicationContext`. The main reason for using this *ContextLoader* over a `ClassPathXmlApplicationContext` is so web-only beans can run.

```
public void setUp() {
    String[] paths = {"/WEB-INF/applicationContext.xml",
                     "/WEB-INF/action-servlet.xml"};
    ctx = new XmlWebApplicationContext();
    ctx.setConfigLocations(paths);
    ctx.setServletContext(new MockServletContext(""));
    ctx.refresh();
}
```

The above code will instantiate any beans defined in their respective XML files.

5. Now write a test method in order to test your Controller. This is where TDD comes into play. The test will drive the design. Write a test method to retrieve a list of users, verify the success, and confirm the view returned is the one you expect. Below is the entire *UserControllerTest*, so you can easily integrate it into your project. The `testGetUsers()` is the method you're most interested in.

```
package org.appfuse.web;

// use your IDE (Eclipse and IDEA rock!) to add imports

public class UserControllerTest extends TestCase {

    private XmlWebApplicationContext ctx;

    public void setUp() {
        String[] paths = {"/WEB-INF/applicationContext.xml",
                         "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ctx.setServletContext(new MockServletContext(""));
        ctx.refresh();
    }

    public void testGetUsers() throws Exception {
        UserController c = (UserController)
                           ctx.getBean("userController");
        ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
                           (HttpServletResponse) null);
    }
}
```

```
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
    }
}
```

Tip: When writing tests for your own project, I recommend creating a `BaseControllerTestCase` that extends `TestCase` and all your `*ControllerTest` classes. In this class's `setUp()` method, you can load the context for all child tests.

With the `testGetUsers()` method, you're grabbing the `UserController` and invoking its `handleRequest()` method, which returns a `ModelAndView`. This method is common to all Spring Controllers, so you'll actually use this same method to test your `FormControllers`. The `ModelAndView` is a unique concept in web frameworks. It contains information about the next page (the view) and what data to expose to it (the model). With Struts, the model and view are separated. The model is usually put into the request (or session) scope, and Actions typically return `ActionForwards`, which are just fancy wrappers around URLs.

Create `UserController` and Configure *action-servlet.xml*

Now that you've written your unit test, it's time to create the `UserController` class so you can compile it.

1. Create a `UserController.java` file in `src/org/appfuse/web`. This class should implement `org.springframework.web.servlet.mvc.Controller` and implement its `handleRequest(request, response)` method. You're also going to need to use the `userManager` to talk to get the list of users, so add a private `userManager` variable and a `setuserManager()` method for Spring's IoC container to use. When you configure this Controller (that is, bean) in the next section, you'll add the `userManager` as a dependency. So far, you have the following class structure:

```
package org.appfuse.web;

// Modern IDEs support easy importing

public class UserController implements Controller {
    private static Log log = LoggerFactory.getLog(UserController.class);
    private UserManager mgr = null;

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

    // put handleRequest() method here
}
```

Implement the `handleRequest()` method to get a list of users and route the user to *userList.jsp*.

```
public ModelAndView handleRequest(HttpServletRequest request,
                                HttpServletResponse response)
    throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'handleRequest' method...");
    }

    return new ModelAndView("userList", "users", mgr.getUsers());
}
```

This method is quite simple; in fact, it would be only one line without the logging statement at the beginning!

Compiling the `UserControllerTest` class should work now, but if you try to run the test it will fail.

```
[junit] IOException parsing XML document from class path resource [WEB-INF/action-servlet.xml]; nested exception is
java.io.FileNotFoundException: Could not open class path resource [WEB-INF/action-servlet.xml]
```

You can see from this error message that you must create the *action-servlet.xml* file to which our unit test refers. By default, Spring expects an XML file for its `DispatcherServlet`. The name of this file defaults to the servlet's name in *web.xml*. In your *web.xml*, you have the following line:

```
<servlet-name>action</servlet-name>
```

This name is used in the `<servlet-mapping>` as well as the first part of Spring's controller configuration file.

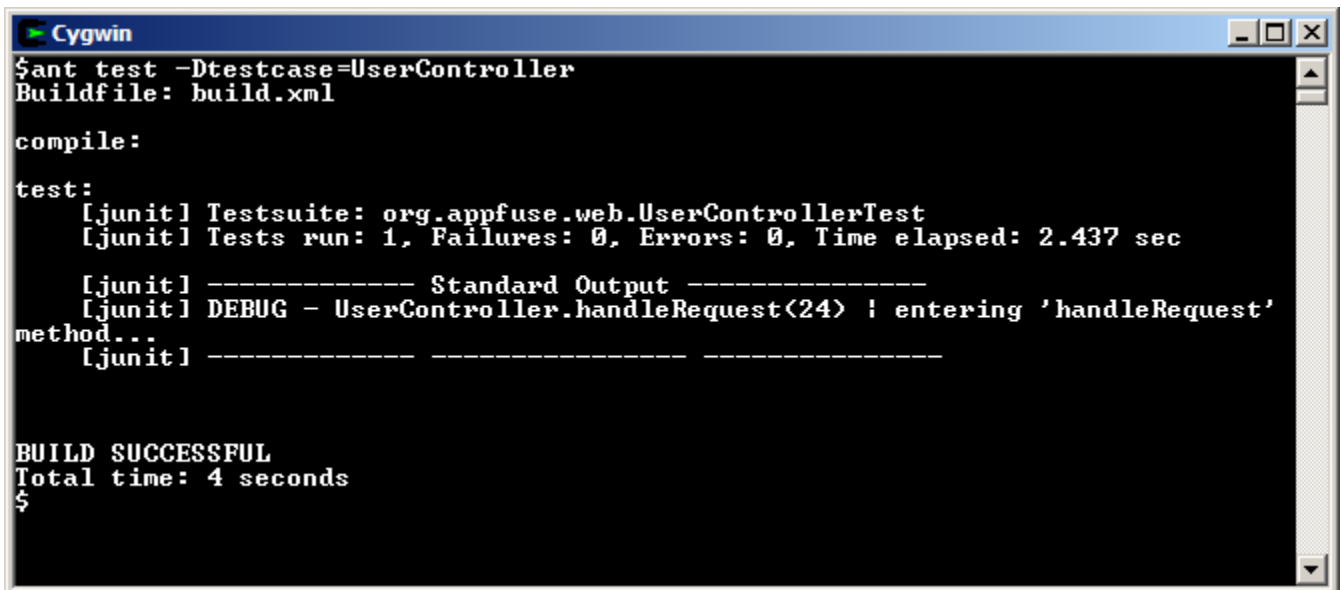
Note: You can override the default filename setting by specifying a `contextConfigLocation` init-parameter on the `DispatcherServlet`.

2. In this example, edit the *action-servlet.xml* file in the **web/WEB-INF/** directory. The *action-servlet.xml* file starts similar to any other Spring configuration file with the DTD at the top and the beginning `<beans>` element. This example also includes the definition for the `UserController` class.

```
<?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="userController" class="org.appfuse.web.UserController">
        <property name="userManager">
            <ref bean="userManager"/>
        </property>
    </bean>
</beans>
```

Now your `UserControllerTest` should run just fine. Execute it by running **ant test -Dtestcase=UserController** or run it as a JUnit Test in Eclipse or IDEA. From the command line, the output should look similar to the following screenshot.



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

compile:

test:
[junit] Testsuite: org.appfuse.web.UserControllerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 2.437 sec

[junit] ----- Standard Output -----
[junit] DEBUG - UserController.handleRequest(24) : entering 'handleRequest'
method...
[junit] -----

BUILD SUCCESSFUL
Total time: 4 seconds
$
```

Figure 4.1: Results of the `ant test -Dtestcase=UserController` command

Modify *userList.jsp* to Work with Spring

Now that the `UserController` is working, you must hook configure Spring so it knows the “`userList`” view actually points to the *userList.jsp*. The simplest way to do this is to use the `InternalResourceViewResolver`, which resolves names to files. It allows you to add a *prefix* and a *suffix*, so you can easily control where your JSPs reside.

1. Add the following XML block to *action-servlet.xml*, after the “`viewController`” bean definition:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
>
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

In the above “`viewResolver`” definition, notice that you’re specifying a `JstlView` class for the “`viewClass`” property. This is so you can use JSTL’s `<fmt:message>` tag, which requires a bit of preparation to use its i18n features with Spring MVC. The prefix is “/” and the suffix is “.jsp.” If you need to move your JSPs to “/WEB-INF/pages,” all you’ll need to change is the prefix.

Note: If you’re developing a production application, I highly recommend placing your JSPs under WEB-INF. If you’re using a Servlet 2.3+ container, any files under WEB-INF will not be accessible from the browser. This can be very useful during development because it enforces MVC and requires you to use a controller to access your views.

By adding the “`viewResolver`” definition, the “`userList`” view name in `UserController` will be resolved to “`/userList.jsp`.” You can use several other `ViewResolvers` depending on your view technology of choice. These will be discussed in *Chapter 6: View Options*. If you want more information now, refer to [Spring’s online View Resolver documentation](#).

2. Now configure URLs in the application so that “`/users.html`” will go to the `UserController`. With Struts, you would do this by specifying action “`paths`” in *struts-config.xml*. If you want to specify more than one path for an Action, you have to create a whole new action-mapping. With Spring MVC, simply define a “`Handler Mapping`” and define which URLs go to which controllers. In most cases, the `SimpleUrlHandlerMapping` is all you need. It allows you to specify url-patterns to bean names.

3. To map “/users.html” to the “userController” bean, add the following to `web/WEB-INF/action-servlet.xml`:

```
<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/users.html">userController</prop>
        </props>
    </property>
</bean>
```

Note: You could also use `BeanNameUrlHandlerMapping` instead of the `SimpleUrlHandlerMapping`. This handler simply looks for bean names (not ids) that match the URL. So if you gave `UserController` a name of “/users.html,” then this handler would resolve it correctly. The `BeanNameUrlHandlerMapping` is the default handler if you don’t define one.

4. After configuring the URL Controller mapping, convert the `userList.jsp` to use Spring-friendly JSP syntax. Open `web/userList.jsp` and change the URL any “user.do?action=*” URLs to be “editUser.html.” More specifically, change the “Add” button to:

```
<button onclick="location.href='editUser.html'">Add User</button>
```

5. Change the first column in the table to point to “editUser.html” instead of “user.do?method=edit.”

```
<td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
```

6. Change `<bean:message>` to `<fmt:message>` (find/replace “bean:message” with “fmt:message”). However, to enable i18n message lookups, you must add a “messageSource” bean to `action-servlet.xml`.

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>messages</value></property>
</bean>
```

This is essentially the same as adding a `<message-resources>` element to `struts-config.xml`. The “basename” property refers to “messages,” which means *look for messages.properties at the root of the classpath*. If you’d like to use more than one .properties file, use the “basenames” property with a `<list>` of `<values>`.

Note: The `ResourceBundleMessageSource` depends on Java’s `java.util.ResourceBundle`, which caches loaded bundles indefinitely. With this class, reloading a bundle during VM executing is not possible. If you need such functionality, refer to [ReloadableResourceBundleMessageSource](#).

7. To test that you've configured Spring's handlers and resolvers correctly, as well as modified *userList.jsp* successfully, start Tomcat, deploy MyUsers (**ant deploy**) and view <http://localhost:8080/myusers/users.html> in your browser. To add a couple of users to the database, run **ant populate**.

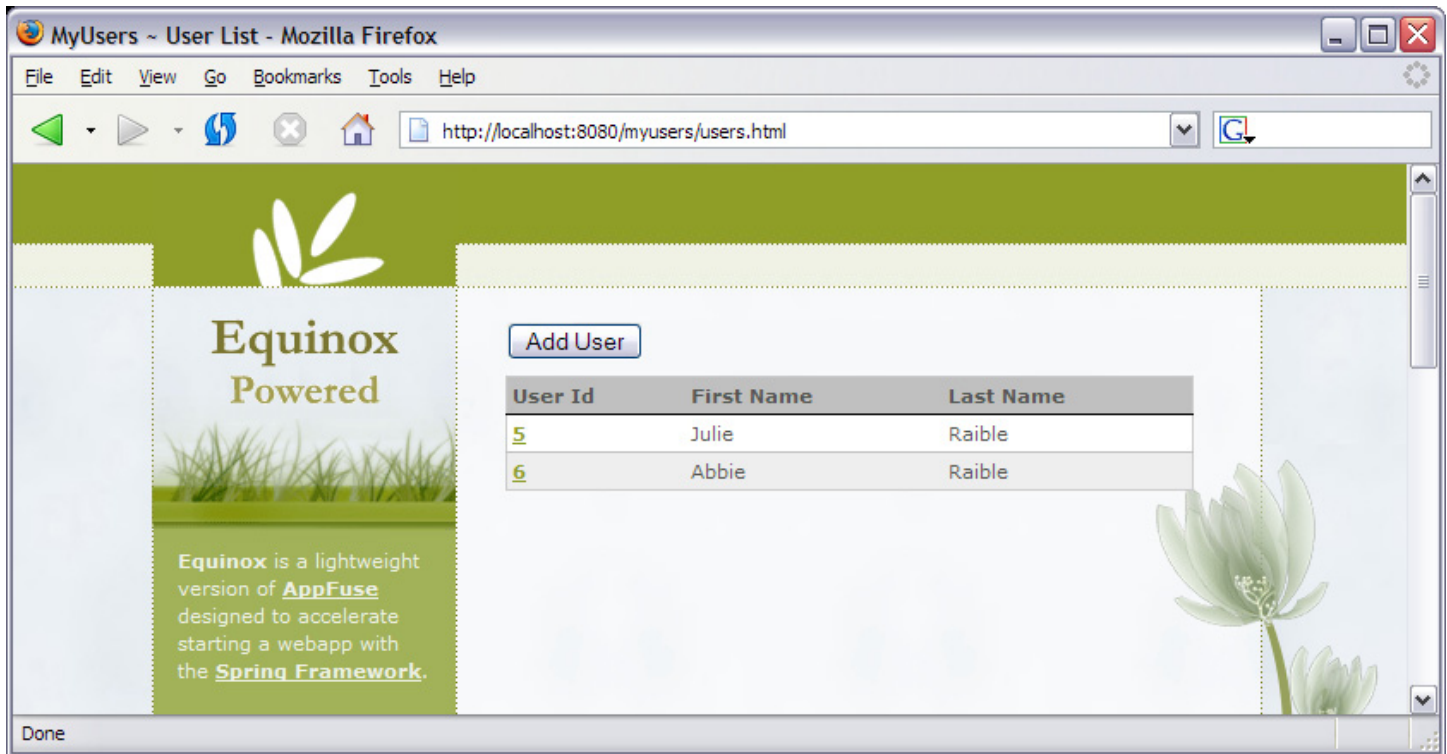


Figure 4.2: Results of the `ant populate` command

If your screen looks like the one above, congratulations! If not, send e-mail to the Equinox users' mailing list (equinox@appfuse.dev.java.net) to get help.

Create Unit Test for UserFormController

The list screen was the easy part since it simply retrieves and displays data. Now you must create a Controller and JSP to handle editing database records. Since you are practicing TDD, start by creating a unit test for the `UserFormController` (that you haven't created yet).

1. Create a `UserFormControllerTest.java` file in the **test/org/appfuse/web** directory. This file should extend JUnit's `TestCase` class and have the following `setUp()` and `tearDown()` methods:

```
package org.appfuse.web;

// resolve imports using your IDE

public class UserFormControllerTest extends TestCase {
    private static Log log =
        LoggerFactory.getLog(UserFormControllerTest.class);
    private XmlWebApplicationContext ctx;
    private UserFormController c;
    private MockHttpServletRequest request;
    private ModelAndView mv;
    private User user;

    public void setUp() throws Exception {
        String[] paths = {"/WEB-INF/applicationContext.xml",
                         "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ctx.setServletContext(new MockServletContext(""));
        ctx.refresh();
        c = (UserFormController) ctx.getBean("userFormController");
        // add a test user to the database
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        user = new User();
        user.setFirstName("Matt");
        user.setLastName("Raible");
        user = mgr.saveUser(user);
    }

    public void tearDown() {
        ctx = null;
        c = null;
        user = null;
    }

    // put testXXX methods here
}
```

Tip: If you're using an IDE, this is a good place to adjust your project's classpath to contain all the JARs in **web/WEB-INF/lib**. In the example you removed Struts-related JARs and added new ones for Spring, so it's probably a bit out of whack.

The `setUp()` method in this class is very similar to the `setUp()` method in `userManagerTest` from *Chapter 2*, except that it also loads *action-servlet.xml*. Separating your business components and data layer classes between the two files allows you to easily switch out the MVC framework without even touching *applicationContext.xml*. This is very powerful for decoupling your different tiers and allows for easy refactoring.

2. Add a few methods to test your CRUD actions (edit, save and delete). The test methods below use Spring's Servlet API mocks to allow for easy testing of Controllers.

```
public void testEdit() throws Exception {
    log.debug("testing edit...");
    request = new MockHttpServletRequest("GET", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertEquals("userForm", mv.getViewName());
}

public void testSave() throws Exception {
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    request.addParameter("firstName", user.getFirstName());
    request.addParameter("lastName", "Updated Last Name");
    mv = c.handleRequest(request, new MockHttpServletResponse());
    Errors errors =
        (Errors) mv.getModel()
            .get(BindException.ERROR_KEY_PREFIX + "user");
    assertNull(errors);
    assertNotNull(request.getSession().getAttribute("message"));
}

public void testRemove() throws Exception {
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("delete", "");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertNotNull(request.getSession().getAttribute("message"));
}
```

In the above methods, you should review a couple of classes. The first is Spring's `MockHttpServletRequest`. This class makes it easy to call GET and POST methods on a given URI (Uniform Resource Indicator). It has a number of constructors, each listed below.

```
public MockHttpServletRequest(ServletContext servletContext)
public MockHttpServletRequest(ServletContext servletContext,
                               String method, String URI)
public MockHttpServletRequest()
public MockHttpServletRequest(String method, String URI)
```

This is a very flexible class and is useful for testing Controllers. Originally, it was only used by Spring developers internally for testing Controllers. As of the 1.0.2 release, it's included in the *spring-mock.jar*.

The second class is the `Errors` object. This is an interface that is implemented by an object to store and expose information about data binding errors. In the `UserFormController` that you will create, you must register a data binder to handle the `java.lang.String` to `java.lang.Long` conversion for `user.setId()`. Registering converters for handling more complex object types (such as dates) will be covered in *Chapter 5*.

The `UserFormControllerTest` class will not compile until you create the `UserFormController` class. If you're using an IDE like IDEA or Eclipse, it will actually prompt you with an icon on the left to auto-create the new class.

Create `UserFormController` and Configure it in *action-servlet.xml*

Start by creating a `UserFormController.java` file in `src/org/appfuse/web`. This class extends `SimpleFormController`, which is a concrete `FormController` implementation that provides configurable form and success views. It automatically resubmits to the form view when validation errors occur, and displays the success view when a submission is valid. This class provides many methods to override in the lifecycle of a displaying a form, as well as submitting a form. This is one of the unique things about Spring's MVC framework versus others like Struts or WebWork. The latter frameworks typically only provide one method for you to override, and you don't have as much control over what happens when. Of course, with Spring's MVC you don't *have* to override its lifecycle methods; it's simply an option if you need it. Towards the end of this chapter is a detailed overview of the different lifecycle methods and when they're called.

The `UserFormController` is designed to be simple so you can easily grasp how Spring MVC works. In fact, you only need to override two methods: `onSubmit()` and `formBackingObject()`. The `onSubmit()` method handles form posts and the `formBackingObject()` method gives the request an `Object` that matches the fields in your HTML form. This method is a convenient location to fetch existing records, as well as good place to instantiate empty objects (for example, to display an empty form). This method's default implementation simply creates a new empty object. In Spring's terminology, this object is called a "Command class." Now that you're familiar with the guts of a `SimpleFormController`, take a look at the code. Below are the contents of the `UserFormController` class, minus the imports:

```
package org.appfuse.web;

// resolve imports using your IDE

public class UserFormController extends SimpleFormController {
    private static Log log =
        LoggerFactory.getLog(UserFormController.class);
    private UserManager mgr = null;

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

    public UserManager getUserManager() {
        return this.mgr;
    }

    /**
     * Set up a custom property editor for converting Longs
     */
    protected void initBinder(HttpServletRequest request,
                               ServletRequestDataBinder binder) {
        NumberFormat nf = NumberFormat.getNumberInstance();
        binder.registerCustomEditor(Long.class, null,
```



```

        new CustomNumberEditor(Long.class, nf, true));
    }

    public ModelAndView onSubmit(HttpServletRequest request,
                                HttpServletResponse response,
                                Object command, BindException errors)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'onSubmit' method...");
        }

        User user = (User) command;

        if (request.getParameter("delete") != null) {
            mgr.removeUser(user.getId().toString());
            request.getSession().setAttribute("message",
                getMessageSourceAccessor()
                    .getMessage("user.deleted",
                        new Object[] {user.getFirstName() +
                            ' ' + user.getLastName()}));
        } else {
            mgr.saveUser(user);
            request.getSession().setAttribute("message",
                getMessageSourceAccessor().getMessage("user.saved",
                    new Object[] {user.getFirstName() +
                        ' ' + user.getLastName()}));
        }

        return new ModelAndView(new RedirectView(getSuccessView()));
    }

    protected Object formBackingObject(HttpServletRequest request)
        throws ServletException {
        String userId = request.getParameter("id");

        if ((userId != null) && !userId.equals("")) {
            return mgr.getUser(request.getParameter("id"));
        } else {
            return new User();
        }
    }
}

```

From this code, you can see how the `formBackingObject()` method simply creates an empty object for adds, and a populated object for edits. You can also see how simple the `onSubmit()` method is; it calls the `UserManager` to save/delete the `User` object. You might notice a lack of exception handling. This is primarily to keep things simple. An exception handling strategy will be covered in *Chapter 5*.

1. Configure this Controller in the *action-servlet.xml* file. In this bean's definition, you will set a fair amount of declarative values: the "successView", the "formView" and the command class and its name. This is also where you will inject its dependency on the `UserManager`. Below is the definition you need to add to *web/WEB-INF/action-servlet.xml*:

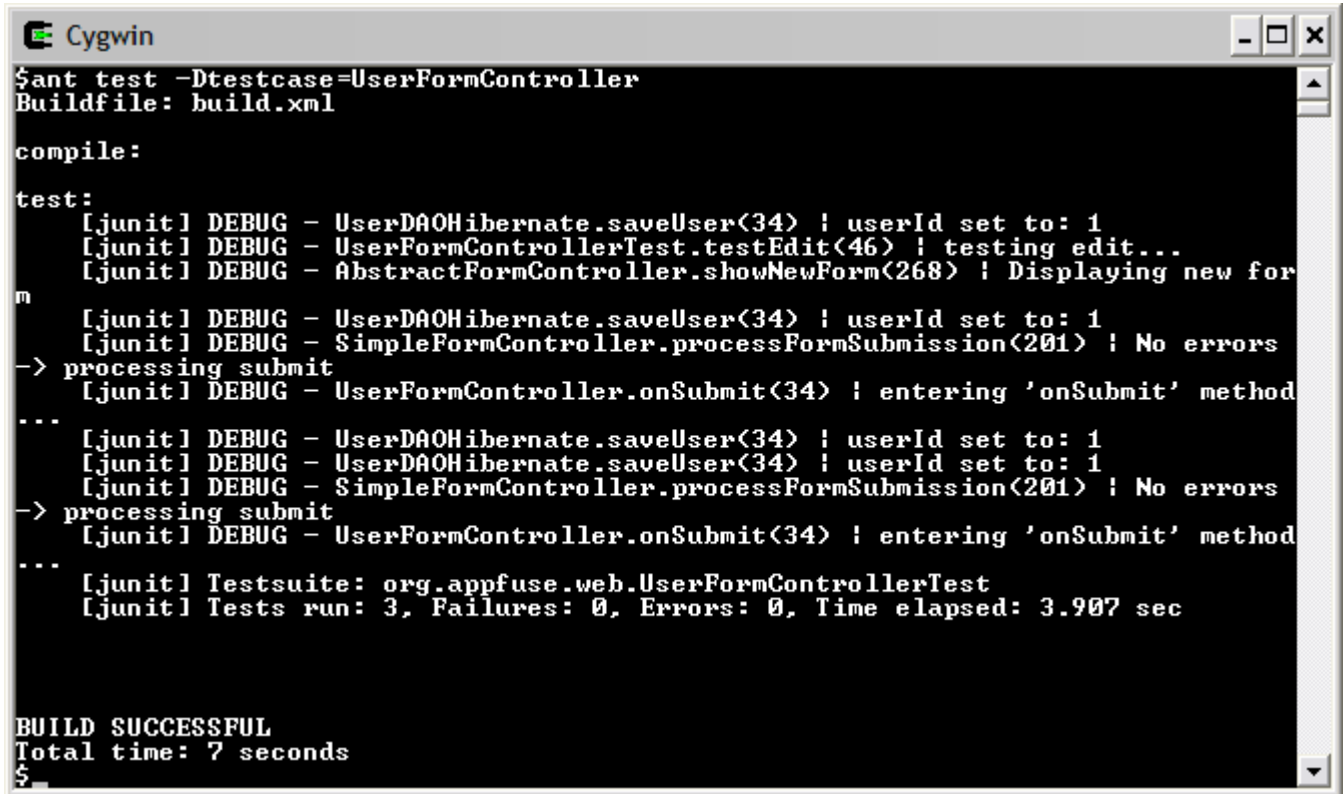
```
<bean id="userFormController"
      class="org.appfuse.web.UserFormController">
  <property name="commandName"><value>user</value></property>
  <property name="commandClass">
    <value>org.appfuse.model.User</value>
  </property>
  <property name="formView"><value>userForm</value></property>
  <property name="successView"><value>users.html</value></property>
  <property name="userManager"><ref bean="userManager"/></property>
</bean>
```

In this definition, the `commandName` property is optional. If you choose not to specify this property, it will default to "command." In the next section, when you create the JSP for this controller, you'll see where the `commandName` property is used. It is not by any code in this controller, nor in its `Test` class.

2. Since you're editing *action-servlet.xml*, add a URL mapping so that "editUser.html" resolves to use the "userFormController" bean. Do this by adding an additional line to the "mappings" property of the "urlMapping" bean.

```
<property name="mappings">
  <props>
    <prop key="/users.html">userController</prop>
    <prop key="/editUser.html">userFormController</prop>
  </props>
</property>
```

Now you should be able to run the `UserFormControllerTest` in your IDE or from the command line. Figure 4.3 shows the output on the command line from running `ant test -Dtestcase=UserForm`.



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

compile:
test:
[junit] DEBUG - UserDAOHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - UserFormControllerTest.testEdit(46) : testing edit...
[junit] DEBUG - AbstractFormController.showNewForm(268) : Displaying new form
[junit] DEBUG - UserDAOHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - SimpleFormController.processFormSubmission(201) : No errors
-> processing submit
[junit] DEBUG - UserFormController.onSubmit(34) : entering 'onSubmit' method
...
[junit] DEBUG - UserDAOHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - UserDAOHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - SimpleFormController.processFormSubmission(201) : No errors
-> processing submit
[junit] DEBUG - UserFormController.onSubmit(34) : entering 'onSubmit' method
...
[junit] Testsuite: org.appfuse.web.UserFormControllerTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.907 sec

BUILD SUCCESSFUL
Total time: 7 seconds
$
```

Figure 4.3: Results of the `ant test -Dtestcase=UserForm` command

If you see something similar to the output above, *nice work!*

Modify *userForm.jsp* to Use Spring's JSP Tags

In this section you will create the JSP to interact with the `UserFormController`.

In the Struts version of this application, the *userForm.jsp* was pretty simple: relying on Struts' `<html:form>` and `<html:text>` JSP tags to make things easy. Unfortunately, Spring doesn't have similar simplistic form-specific tags. However, they have good reasons. The main premise behind the lack of rich form tags is to give the user maximum control over the HTML. Since Spring's form-handling tags do not generate any HTML, the user has complete control over it. Since several Struts folks are migrating to Spring, there have been some discussions on the mailing list of producing something similar.

Besides unobtrusive form-handling tags, Spring also allows you to configure your form's action URL to whatever you like. This may be annoying because it requires more typing, as though you're hard-coding the URL to the controller.

1. Struts prepends the `contextPath`, and appends the suffix you defined in *web.xml* (such as `*.do`).

```
<html:form action="/user">
```

With Spring, an equivalent action declaration looks like the following:

```
<form action="<c:url value="/user.html"/>">
```

You could also use relative paths, which makes the Spring version less typing than the Struts version:

```
<form action="user.html">
```

The `<c:url>` version prepends your application's `contextPath` (for example, `/myusers`), allowing you to easily move the JSP to a sub-folder. Furthermore, it gives you full control over the extension you want to use. This works very nicely if you want to have secure and unsecure sections of your application. You can define servlet-mappings in *web.xml* (such as `*.html` and `*.secure`) and then protect any `*.secure` URLs. You cannot do this with Struts, since the form action's URL is always filled in for you.

- The next difference between a Struts JSP and a Spring JSP is how Spring binds object values to form input fields. Struts' input tags look up information from the `<html:form>` tag and match properties to getters in the `ActionForm`. Spring's `<spring:bind>` tag allows you to *bind* getters (properties) in your command object with input fields. Compare the differing syntax for the *firstName* input field. If you were using Struts, you'd use an `<html:text>` tag:

```
<html:text property="user.firstName"/>
```

Using Spring, the syntax is a bit more verbose. However, another major advantage of the `<spring:bind>` tag is that you can render error messages next to the field. This is shown on the third line below. Of course, including this line is completely optional.

```
<spring:bind path="user.firstName">
  <input type="text" name="firstName" value="${status.value}"/>
  <span class="fieldError">${status.errorMessage}</span>
</spring:bind>
```

The Struts `ActionForm` and Spring `Command` objects are very similar in their functionality. However, Spring allows easier binding to your domain objects, eliminating the need (in many cases) to develop a form just to handle web input. One case where you may still need a “web-only form object” is when you want to combine two domain objects into one form, or if you want to put two forms on one page.

- The last syntax change you must make in *userForm.jsp* is to replace `<bean:message>` with `<fmt:message>`, as well as change the submit buttons to be simple HTML buttons. Below you will find the full code for *web/userForm.jsp*:

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

<title>MyUsers ~ User Details</title>

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

<form method="post" action="<c:url value="/editUser.html"/>"
  onsubmit="return validateUser(this)">
  <spring:bind path="user.id">
  <input type="hidden" name="id" value="${status.value}"/>
  </spring:bind>
  <table>
  <tr>
    <th><fmt:message key="user.firstName"/>: </th>
    <td>
      <spring:bind path="user.firstName">
      <input type="text" name="firstName" value="${status.value}"/>
      <span class="fieldError">${status.errorMessage}</span>
      </spring:bind>
    </td>
```

```

</tr>
<tr>
  <th><fmt:message key="user.lastName"/>: </th>
  <td>
    <spring:bind path="user.lastName">
      <input type="text" name="lastName" value="{status.value}"/>
      <span class="fieldError">{status.errorMessage}</span>
      <c:if test="{not empty status.errorMessage}">
        </c:if>
    </spring:bind>
  </td>
</tr>
<tr>
  <td></td>
  <td>
    <input type="submit" class="button" name="save" value="Save"/>
    <c:if test="{not empty param.id}">
      <input type="submit" class="button" name="delete"
        value="Delete"/>
    </c:if>
  </td>
</tr>
</table>
</form>

<html:javascript formName="user"/>

```

Since this is the first time you're going to be using any Spring-specific JSP Tags, now is a good time to modify the *web/taglibs.jsp* file. Remove the Struts' taglib declarations and add Spring's. Below are the contents of the updated *taglibs.jsp* file for Spring:

```

<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
  prefix="html" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator"
  prefix="decorator"%>

```

You're getting very close to replicating the functionality from the Struts version of MyUsers in *Chapter 2*. The last thing you need to do is add validation.

Configure Commons Validator for Spring

At the time of this writing, Spring 1.0.2 has been released, and it does not contain a built-in declarative validation framework. The core validation framework requires you to create classes to validate other classes. While this is rather simple, I prefer the way I learned with Struts: using Commons Validator and declaring rules in an XML file. Daniel Miller [recently added support](#) for using Commons Validator with Spring.

For Struts users, using the validation framework they're familiar with should make Spring MVC even easier.

1. To migrate the *validation.xml* file from Struts to Spring MVC, you only need to change a few attribute values: the *formName* (*userForm* *user*) and the field names (remove “*user.*” since you're not wrapping it with a *DynaActionForm*). The modified-for-Spring version is as follows:

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

2. Replace the Struts-specific *validation-rules.xml* with a Spring-compatible one. This file defines all of the classes and methods to use for validation, as well as defining JavaScript methods for client-side validation. You can download this file from [AppFuse's CVS](#). Once you've replaced *web/WEB-INF/validation-rules.xml*, make sure the JavaScript at the bottom of *userForm.jsp* refers to “*user*,” not “*userForm*.”

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

Note: The above one-line JavaScript tag is not the recommended way to configure client-side validation with Commons Validator. The above method results in all the JavaScript functions being included in the final HTML. *Chapter 5* discusses a cleaner way, where the functions are referenced from an external JavaScript file.

3. To notify Spring that you want to use Commons Validator as your validation engine, add a couple of `<bean>` definitions to *web/WEB-INF/action-servlet.xml*:

```
<bean id="validatorFactory"
      class="org.springframework.validation.commons.DefaultValidatorFactory"

      init-method="init">
    <property name="resources">
      <list>
        <value>/WEB-INF/validator-rules.xml</value>
        <value>/WEB-INF/validation.xml</value>
      </list>
    </property>
  </bean>

<bean id="beanValidator"
      class="org.springframework.validation.commons.BeanValidator">
    <property name="validatorFactory">
      <ref local="validatorFactory"/>
    </property>
  </bean>
```

The first bean (`validatorFactory`) loads the Validator's methods and class mappings (*validation-rules.xml*), as well as the application-specific validation rules (*validation.xml*). The second bean (`beanValidator`) applies validation to any POJO. To configure your `UserFormController` to use the `beanValidator` for validation, you simply need to add a “`validator`” property to the “`userFormController`” definition.

```
<property name="validator"><ref bean="beanValidator"/></property>
```

Note: The developers of Spring plan to add their own declarative validation framework in the coming months. However, Commons Validator is currently the only Spring-compatible declarative validation framework. Commons Validator support is currently in Spring's sandbox, and there's talk of integrating it into the core at a future date.

4. Now run **ant deploy**, open your browser to <http://localhost:8080/myusers/editUser.html> and try to add a new user without specifying his last name; you should see the following JavaScript alert:

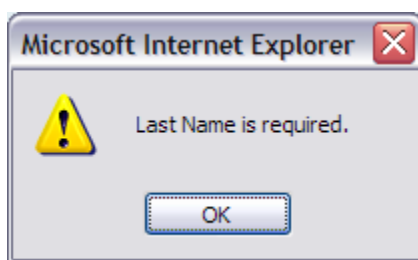


Figure 4.4: Result of adding a new user without a last name

If you want to make sure things are *really* working as expected, turn off JavaScript and ensure that server-side validation is working. This is easy in [Mozilla Firefox](#); just go to Tools > Options > Web Features and uncheck “Enable JavaScript.” Now if you clear the *lastName* field and save the form, you should see the following:

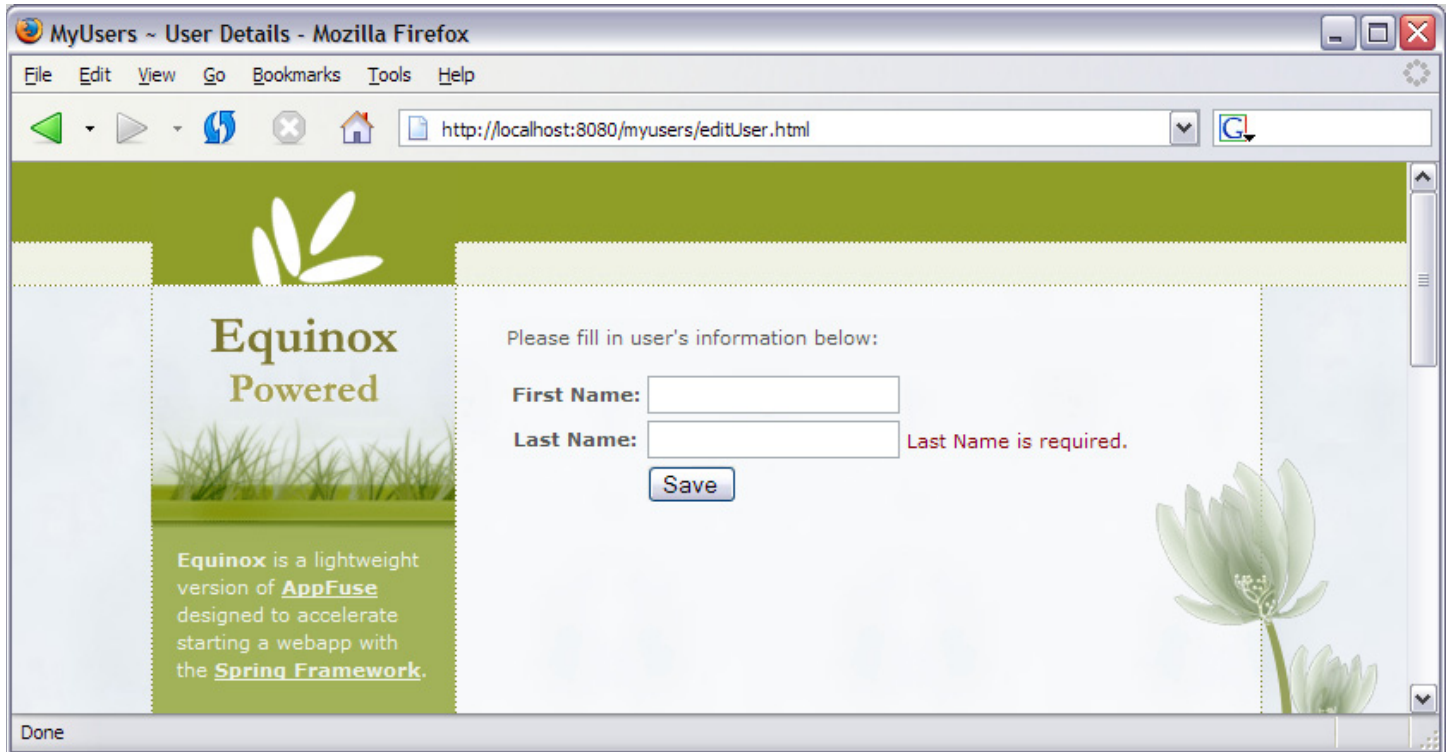


Figure 4.5: Results of entering a user without a last name in a browser

5. To capture all the errors at the top of your JSP (like Struts), add the following code block to the top of the JSP, just after the `<title>`:

```
<spring:bind path="user.*">
  <c:if test="${not empty status.errorMessages}">
    <div class="error">
      <c:forEach var="error" items="${status.errorMessages}">
        <c:out value="${error}" escapeXml="false"/><br />
      </c:forEach>
    </div>
  </c:if>
</spring:bind>
```

Run **ant deploy** and click “Save” without adding a lastName (with JavaScript turned off). Your screen should resemble the following screenshot:

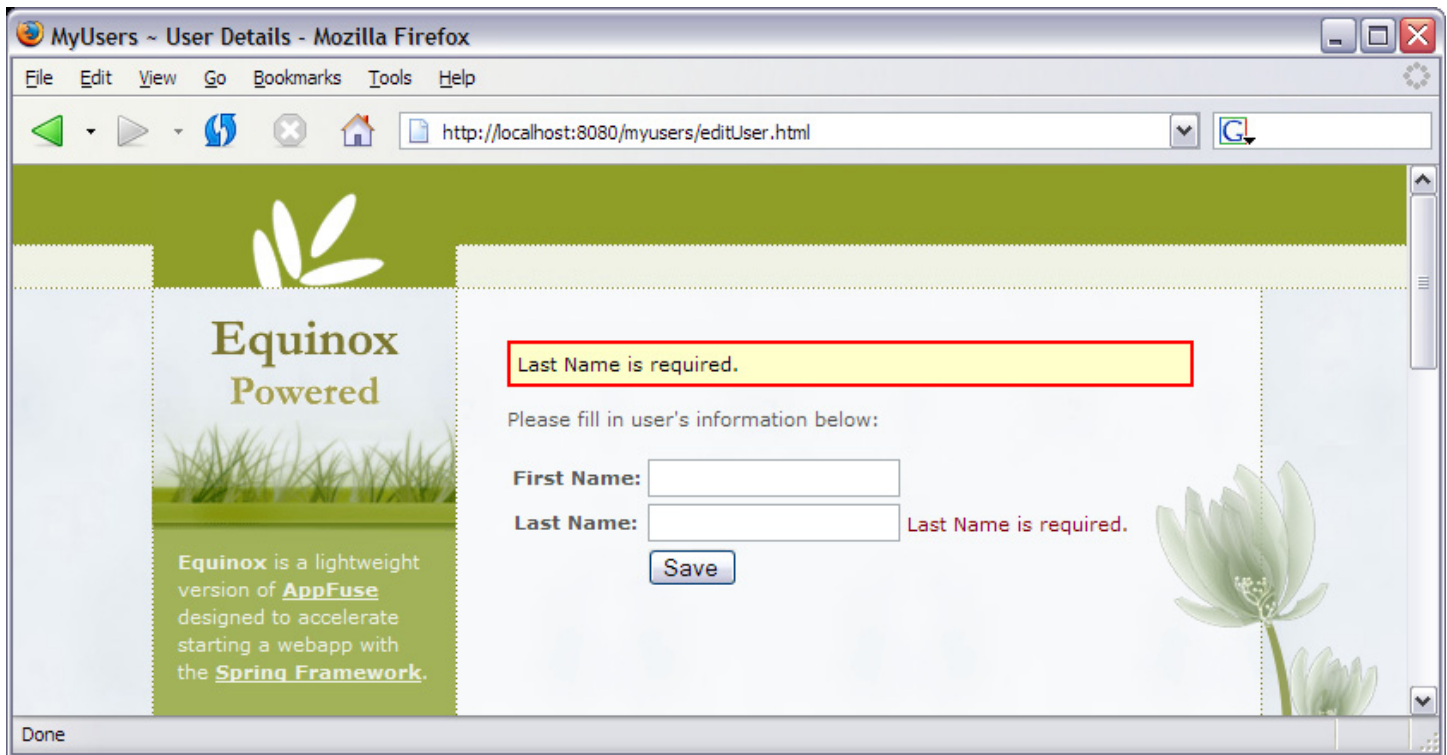


Figure 4.6: Error message added to the top

You’ve just converted a Struts app to use Spring for its MVC layer. *Congratulations!*

SimpleFormController: Method Lifecycle Review

The `SimpleFormController` is one of many *CommandControllers* in Spring's MVC package. These controllers are designed to interact with domain objects and dynamically bind parameters from the request to the objects. In comparison to Struts, Spring is much cleaner because it doesn't require your domain objects to implement an interface or extend a superclass. Rather than describing each command controller and its functionality, I'm simply going to point out that there are only two that you'll likely need:

`SimpleFormController` and `AbstractWizardFormController`.

- `SimpleFormController` is a concrete `FormController` that provides configurable form and success views, and an `onSubmit` chain for convenient overriding. It automatically resubmits to the form view in case of validation errors, and renders the success view in case of a valid submission.
- `AbstractWizardFormController` is a `FormController` for typical wizard-style workflows. In contrast to classic forms, wizards have more than one page view. Because of this, various methods allow the user to go next, back, cancel or finish.

`SimpleFormController` can be a bit overwhelming at first. I recommend that you read [its JavaDocs](#), which describe the lifecycle (that is, workflow) of its methods. Figure 4.7 illustrates the lifecycle for a GET request, and Figure 4.8 illustrates the lifecycle for a POST request.

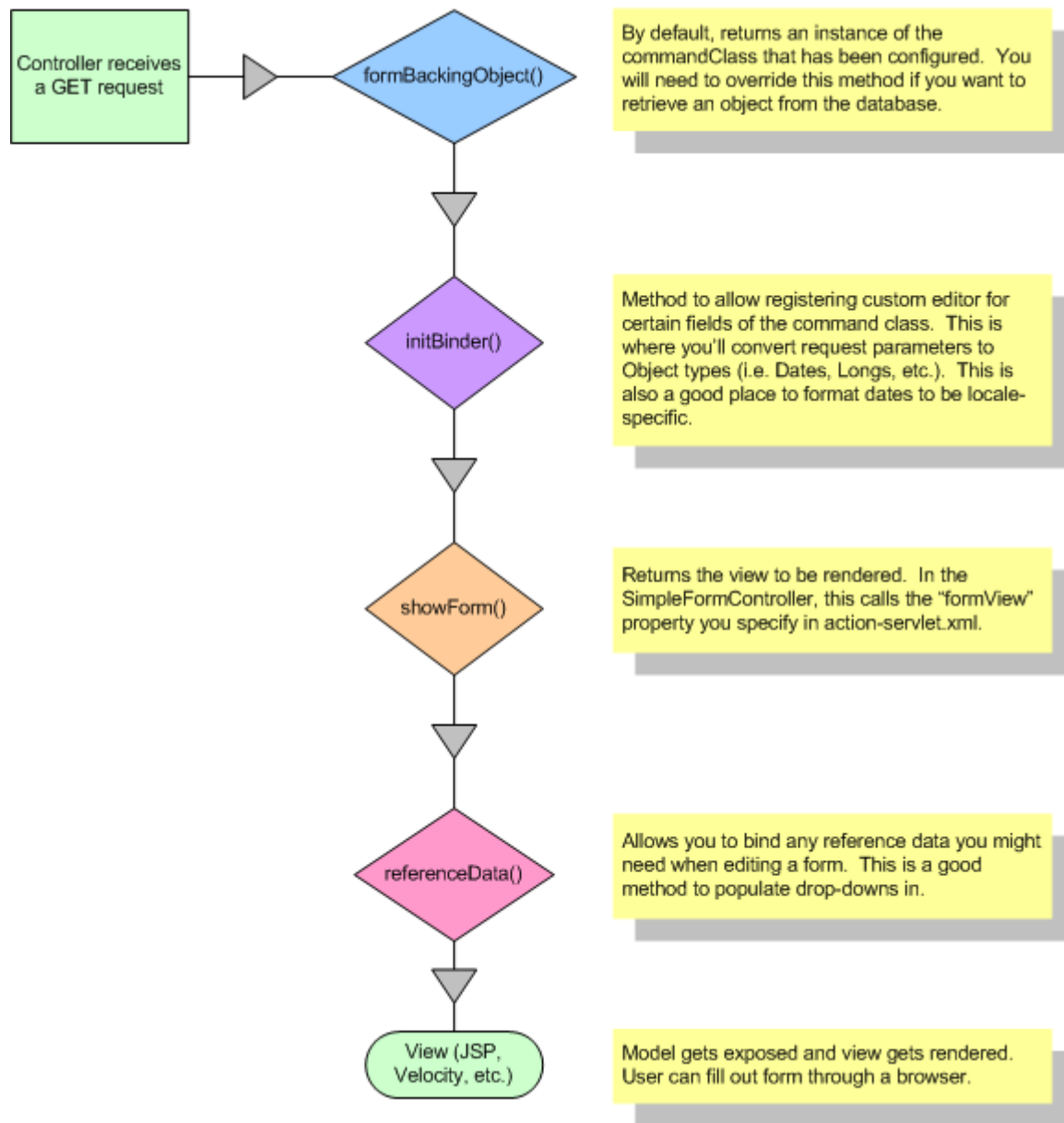
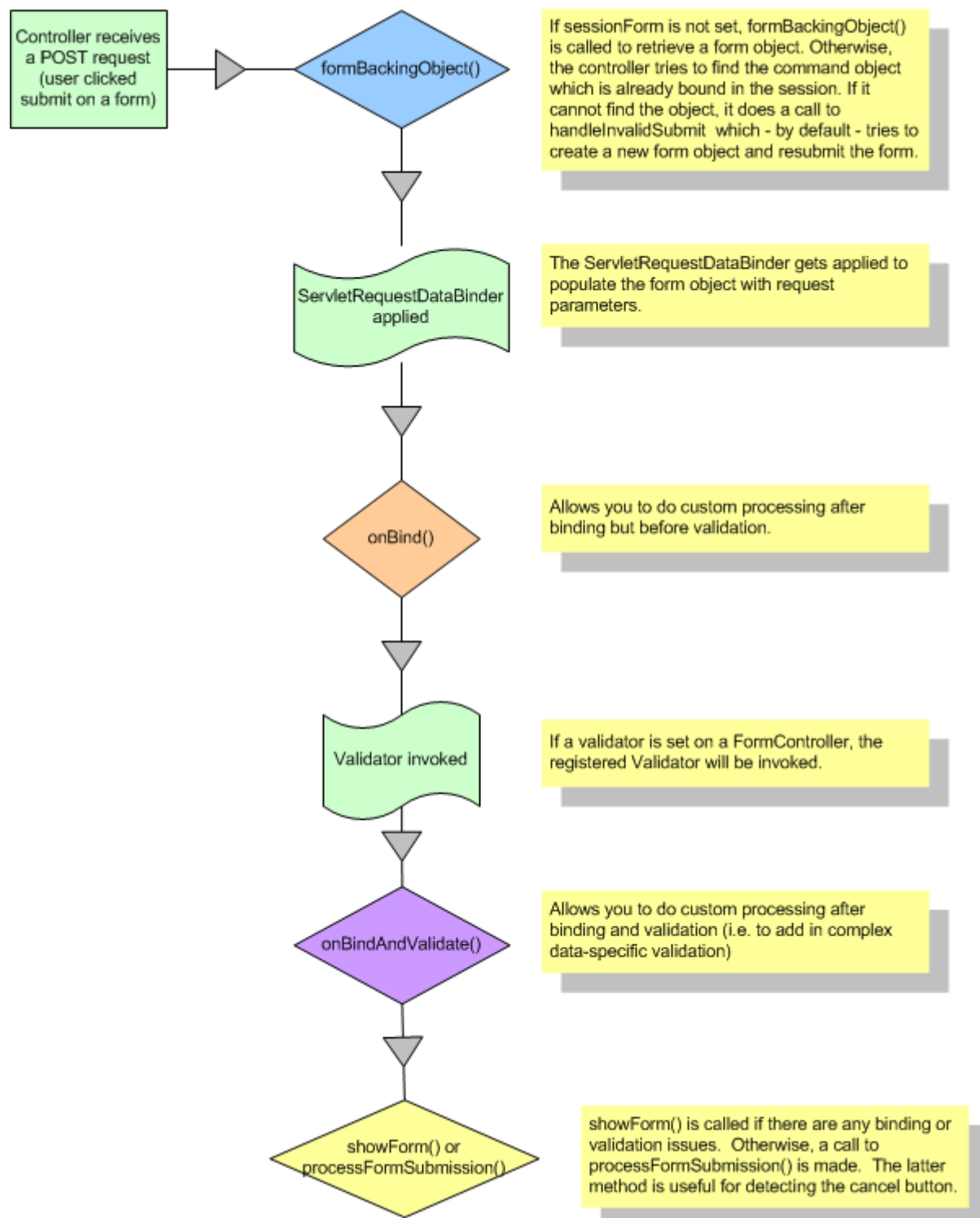


Figure 4.7: GET request lifecycle

**Figure 4.8:** POST request lifecycle

Spring's JSP Tags

We briefly touched on the `<spring:bind>` tag when modifying *userForm.jsp*; now let's look at the other JSP tags available. Below is a list of current tags that ship with Spring by default.

- `spring:hasBindErrors` provides support for binding the errors for an object. This tag seems to be most useful for checking a command object for bind errors.
- `spring:bind` binds command object properties to form fields. This tag exposes a “status” variable in the `pageContext`. This variable can be grabbed with JSP's EL; `${status.value}` will give you a properties value, and `${status.errorMessage}` will display any errors associated with a property. This is the most useful tag in Spring's taglib.
- `spring:transform` provides support for transforming properties not contained by a command object using `PropertyEditors` associated with the command object. This can be useful if you need to transform data from the `referenceData()` method (for example, drop-downs). It must be used *inside* a `spring:bind` tag. It's not currently used in any of Spring's sample apps.
- `spring:message` is similar to JSTL's `<fmt:message>` tag, but supports Spring's `MessageSource` concept. It also works with Spring's locale support. JSTL's `<fmt:message>` tag has fulfilled all my i18n needs with Spring MVC. It's not currently used in any of Spring's sample apps.
- `spring:htmlEscape` sets the default HTML escape value for the current page. The default is “false.” You can also set a `defaultHtmlEscape` *web.xml* context-param. Using the tag in a JSP overrides any settings in *web.xml*. It's not currently used in any of Spring's sample apps.
- `spring:theme` looks up the theme message in the scope of the current page. This tag will be covered in greater detail in Chapter 5 when we talk about Templating.

From the above list, you can see that many of the core tags are not used. Therefore, you've already seen the most important one: `spring:bind`.

Summary

This chapter covered a lot of material. It discussed unit testing and using *spring-mock.jar* to test Controller classes. Then it demonstrated how to convert the MyUsers app to use Spring MVC instead of Struts. Through this process, you learned how to create a Controller unit test and how to configure *action-servlet.xml* for Controllers, Handlers and Resolvers. You saw how to modify a Struts JSP to use Spring tags, and how to add declarative validation with Commons Validator. Lastly, it discussed the `SimpleFormController`'s lifecycle and Spring's JSP tags.

This chapter was designed to show you the basics of developing webapps with Spring MVC. My favorite part of Spring MVC is its method lifecycles. *Chapter 5* will cover more advanced validation and website *skinning* (also called *templating*). You'll also learn how to handle exceptions in Controllers and how to do a simple file upload.

Advanced MVC: Templates, Validation, Exceptions and Uploading Files

How to Integrate Tiles and SiteMesh for Page Decoration, Implementing Validation, Handling Exceptions, Uploading Files and Sending E-mail

This chapter covers advanced topics in web frameworks, particularly validation and page decoration. It shows the user how to use Tiles or SiteMesh to decorate a web application. It also explains how the Spring framework handles validation, and shows examples of using it in the web business layers. Finally, it explains a strategy for handling exceptions in the controllers, how to upload files and how to send e-mail.

Several years ago, web developers didn't have frameworks to help them develop Java/JSP-based applications. They were more concerned with getting it done than doing it right. Today, numerous *frameworks* are available to accelerate a developer's efficiency. They have built-in page decoration, validation engines, exception handling and file upload. Some even support *interceptors*, which can interrupt a web request and perform logic on-the-fly.

Spring supports all of these features as well. This chapter explores how to configure page decoration frameworks like SiteMesh and Tiles in your Spring-based webapp. It then shows how you can build from your Struts Validator knowledge and use the Commons Validator with Spring (which nicely supports client-side validation with JavaScript). After validation, this chapter covers exception handling, applying interceptors and uploading files. Lastly, it briefly covers sending e-mail.

This may sounds like a lot to accomplish in one chapter, but it will be simple and easy to understand. Furthermore, all of these technologies and concepts will be viewed in the context of the MyUsers application. At the end of this chapter, you'll have a reference application that employs all of these features.

Templating with SiteMesh

SiteMesh is an open-source layout and page decoration framework from the [OpenSymphony](#) project. It was originally created over 5 years ago, when Joe Walnes downloaded the first Sun servlet engine and wrote it using servlet chains. Over the years, the basic design has stayed the same; content is intercepted and parsed, and a decorator mapper finds a decorator and merges everything together. The diagram below shows a simplistic example of how this works.

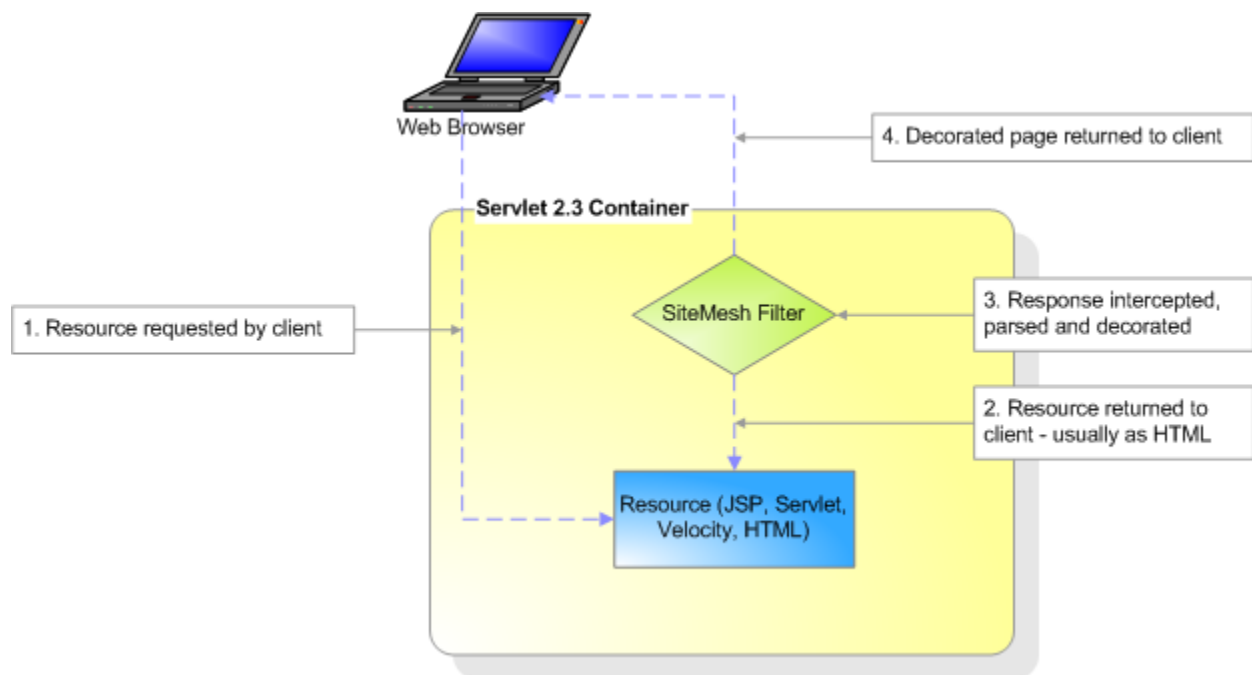


Figure 5.1: The SiteMesh process

Skinning is an essential element to every web application. The ability to edit one or two files to change the *entire* layout of an app is a must for maintainability. SiteMesh is a simple decoration framework and is very easy to install and configure.

Installation and Configuration

The installation instructions are for SiteMesh 2.0.2, which hasn't been released at the time of this writing. To get the latest JAR, you can either check out the project from its CVS server, or you can use the *sitemesh-2.0.2.jar* that ships with Equinox. Since SiteMesh is already configured in the MyUsers application, you must [download the MyUsers application for this chapter](#) to start from scratch. All of the topics for this chapter have not been configured in the downloaded application. After downloading it, extract it to "myusers-ch5" and then copy "myusers-ch5" to "myusers-sitemesh."

Note: You will copy "myusers-ch5" to "myusers-tiles" when you install and configure Tiles.

Run `ant remove clean install` in Tomcat; your screen should resemble the one below when you open <http://localhost:8080/myusers>. This is much different and quite bland when compared with previous screens you've seen. Install and configure SiteMesh to pretty it up a bit.

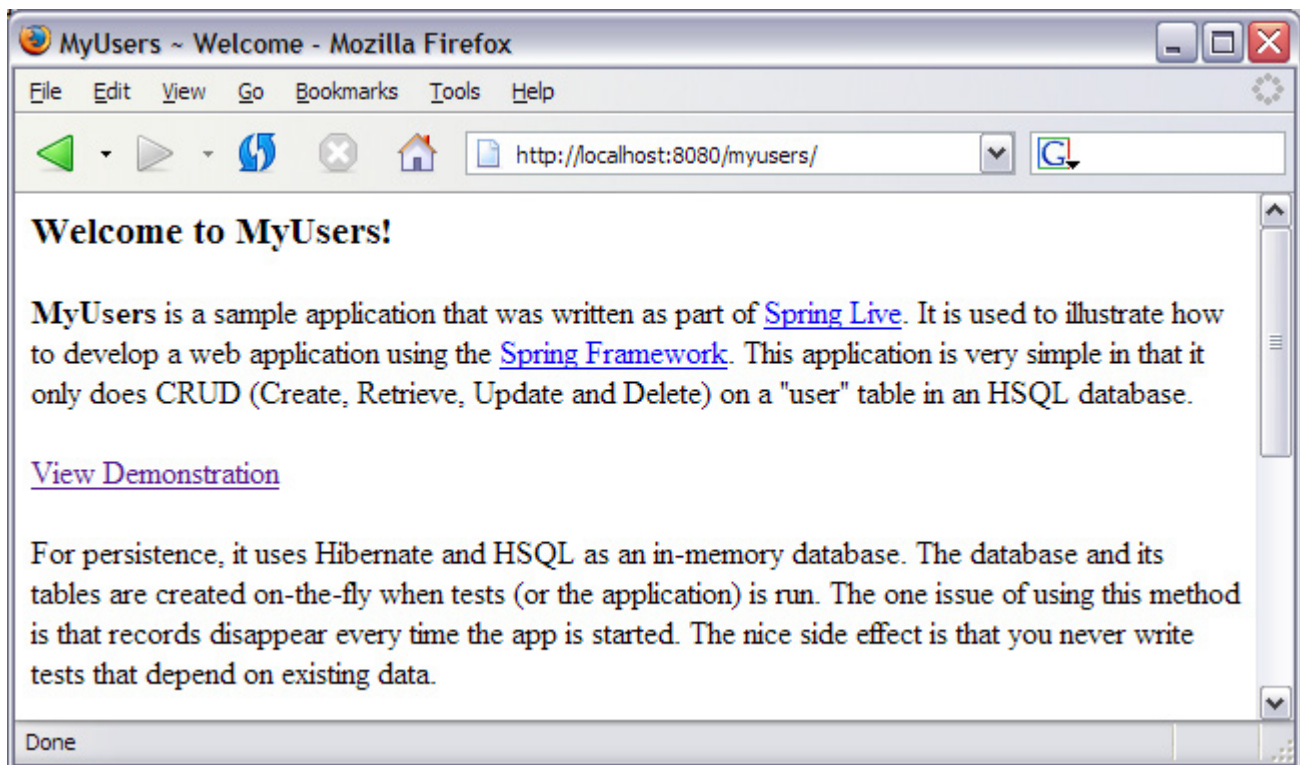


Figure 5.2: "Welcome to MyUsers" without decoration

Step 1: Configure SiteMesh in web.xml

The *sitemesh-2.0.2.jar* should already be in the **web/WEB-INF/lib** directory of “myusers-sitemesh,” so you won’t have to install that. However, if you were installing SiteMesh from scratch, you would download it from <http://www.opensymphony.com/sitemesh>.

1. Open your *web.xml* file (in **web/WEB-INF**) to edit.
2. At the top of this file, right after the `<display-name>` and before the `<context-param>`, add the following `<filter>` definition:

```
<filter>
  <filter-name>sitemesh</filter-name>
  <filter-class>
    com.opensymphony.module.sitemesh.filter.PageFilter
  </filter-class>
</filter>
```

3. After the `<context-param>` and before the `<listener>` element, add the following `<filter-mapping>`:

```
<filter-mapping>
  <filter-name>sitemesh</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Step 2: Create Configuration Files

1. Create a *sitemesh.xml* file in **web/WEB-INF**, as shown below:

```
<sitemesh>
  <page-parsers>
    <parser default="true"

class="com.opensymphony.module.sitemesh.parser.DefaultPageParser"/>
    <parser content-type="text/html"

class="com.opensymphony.module.sitemesh.parser.FastPageParser"/>
    <parser content-type="text/html; charset=ISO-8859-1"

class="com.opensymphony.module.sitemesh.parser.FastPageParser"/>
  </page-parsers>

  <decorator-mappers>
    <mapper

class="com.opensymphony.module.sitemesh.mapper.ConfigDecoratorMapper">
      <param name="config" value="/WEB-INF/decorators.xml"/>
    </mapper>
  </decorator-mappers>
</sitemesh>
```

This file configures *page-parsers* and *decorator-mappers*. The page-parsers are configured to parse certain content-types, and it's unlikely you'll ever need to change these values. The ConfigDecoratorMapper is one of [several possible decorator-mappers](#). Specifically, it tells SiteMesh to read decorators and mappings from the “config” property, which is */WEB-INF/decorators.xml* by default. Because you're using the default, you could remove the `<param>` element in *sitemesh.xml* and everything would work the same. The *decorators.xml* file specifies which URL patterns will use which decorators.

2. Create a *decorators.xml* file in **web/WEB-INF** and put the following XML into it.

```
<decorators defaultdir="/decorators">
  <decorator name="default" page="default.jsp">
    <pattern>/*</pattern>
  </decorator>
  <decorator name="none">
    <!-- These files will not get decorated. -->
    <pattern>/scripts/*</pattern>
  </decorator>
</decorators>
```

Step 3: Create a Decorator

Lastly, build a *decorator* that will act as the template to wrap the pages in your app. If you're familiar with Tiles, this is the same thing as creating a *base layout*.

1. Create a `web/decorators/default.jsp` file, as was configured in the `decorators.xml` file.
2. Put the following code into the `default.jsp` file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

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

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
    <title><decorator:title default="MyUsers"/></title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <c:set var="ctx" value="{pageContext.request.contextPath}"/>
    <link href="{ctx}/styles/global.css" type="text/css"
rel="stylesheet"/>
    <link href="{ctx}/images/favicon.ico" rel="SHORTCUT ICON"/>
    <decorator:head/>
    <!-- HTML & Design contributed by Boer Attila (http://www.calcium.ro)
-->
    <!-- Found at http://www.csszengarden.com/?cssfile=/083/083.css&page=2
-->
</head>
<body>
<a name="top"></a>
<div id="container">
    <div id="intro">
        <div id="pageHeader">
            <h1><span>Welcome to Equinox</span></h1>
            <div id="logo" onclick="location.href='<c:url value="/" />'"
                onkeypress="location.href='<c:url value="/" />'"></div>
            <h2><span>Spring Rocks!</span></h2>
        </div>

        <div id="quickSummary">
            <p>
                <strong>Equinox</strong> is a lightweight version of
                <a href="http://raibledesigns.com/appfuse">AppFuse</a>
                designed
                to accelerate starting a webapp with the
                <a href="http://www.springframework.org">Spring Framework</
a>.
            </p>
            <p class="credit">
```

```

083.css">
    <a href="http://www.csszengarden.com/?cssfile=/083/
Design and CSS</a> donated by <a href="http://
www.calcium.ro">
    Bo&eacute;r Attila</a>.
    </p>
</div>

<div id="content">
    <%@ include file="/messages.jsp"%>
    <decorator:body/>
</div>
</div>

<div id="supportingText">
    <div id="underground">
        <decorator:getProperty property="page.underground"/>
    </div>
    <div id="footer"></div>

</div>

<div id="linkList">
    <div id="linkList2">
    </div>
</div>

</div>

</body>
</html>

```

The important elements to look for are the `<decorator:*>` tags, which are highlighted in yellow. At the top of the document, a `<decorator:title>` grabs the `<title>` tag from JSP pages that are wrapped, and the `<decorator:head/>` tag pulls in anything defined in `<head>`. In the middle, a `<decorator:body/>` tag grabs the “body” of the page. Lastly, the `<decorator:getProperty>` tag pulls in a `<content>` tag that’s defined in your decorated JSPs. Here’s an example from the *index.jsp* page:

```

<content tag="underground">
    <h3>Additional Information</h3>
    <!-- more content here -->
</content>

```

3. Add the `<decorator>` taglib directive to `web/taglibs.jsp`. Add the following line at the bottom of this file:

```
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator"
    prefix="decorator" %>
```

4. Run **ant deploy reload**, and open your browser to <http://localhost:8080/myusers>; you should see a nice and pretty page like the one below.

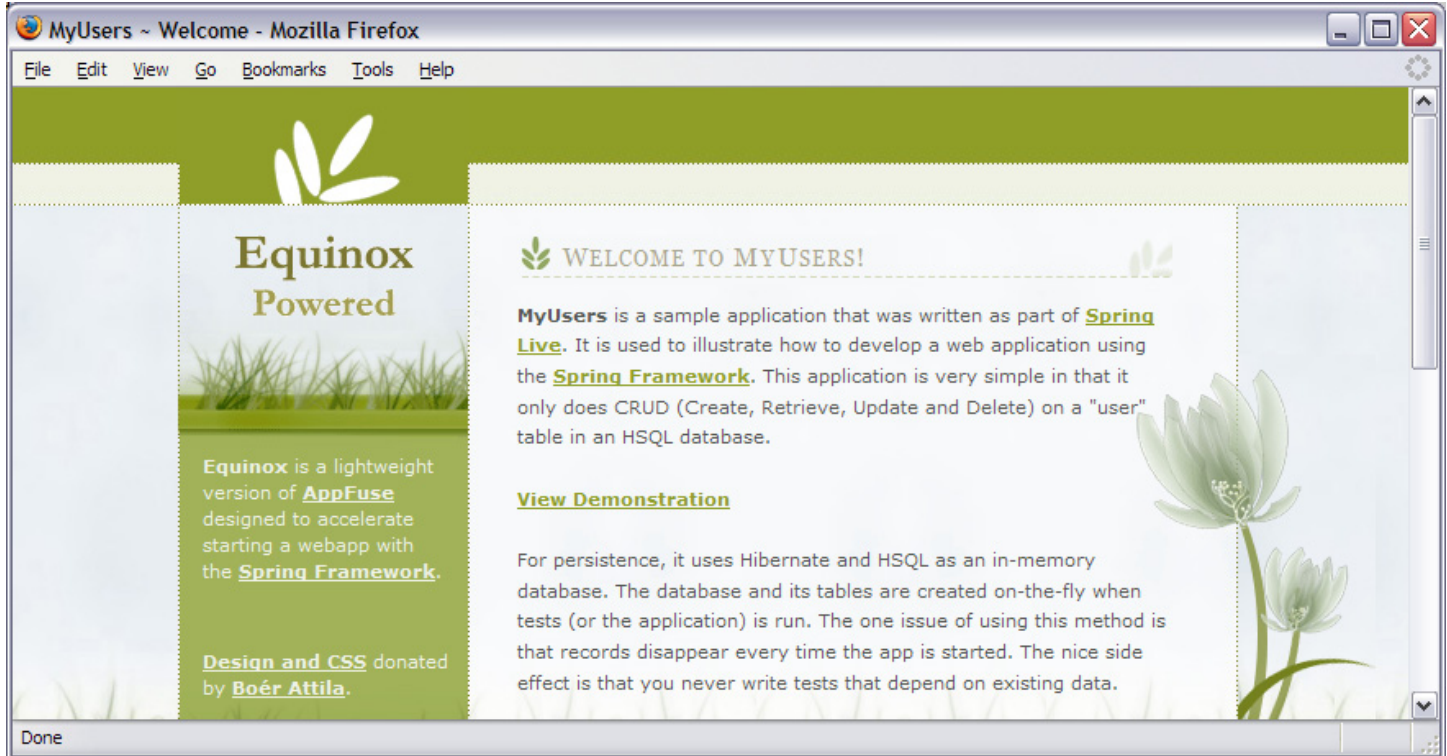


Figure 5.3: "Welcome to MyUsers" with decoration

This is a clean and simple, yet attractive design. For the most part, your decorated pages can use standard HTML elements. Yet they don't need all the elements (such as `<html>`, `<body>`) that static pages require. Even better, SiteMesh doesn't care what server-side technology you use for your application; it works with CGI, PHP, Servlets, Velocity and FreeMarker. This makes it very Spring-friendly since Spring supports so many different view technologies (covered thoroughly in the next chapter).

Templating with Tiles

Tiles is a templating and document layout engine much like SiteMesh. Tiles is the default layout engine for Struts, while most SiteMesh users tend to use WebWork (mainly due to the fact that both WebWork and SiteMesh come from OpenSymphony). Tiles was originally written by Cedric Dumoulin and released shortly after Struts 1.0. Initially, it was a separate add-on to Struts, much like the Validator. Much of the work in Struts 1.1 was devoted to extracting useful components from Struts into the Jakarta Commons projects. Because no one had the time to extract it, Tiles did not become a separate project, but became a part of the core Struts (in *struts.jar*).

The next few sections show you how to configure Tiles to work with Spring's MVC Framework.

Installation and Configuration

The instructions below are for Struts 1.1, with which Tiles is integrated; hence it does not have its own version number.

Copy “myusers-ch5” to “myusers-tiles.” If you haven't downloaded “myusers-ch5” yet, you can [download](http://sourcebeat.com/downloads) it from <http://sourcebeat.com/downloads>. At this point, if you run **ant remove clean install**, your screen will look like it did before the SiteMesh section (Figure 5.2).

Step 1: Configure Spring to Recognize Tiles

1. In order for Spring to recognize Tiles and use it for rendering views, create a “tilesConfigurer” bean definition in the *action-servlet.xml* file in **web/WEB-INF**, as shown below:

```
<bean id="tilesConfigurer"
      class="org.springframework.web.servlet.view.tiles.
        TilesConfigurer">
  <property name="factoryClass">
    <value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</
value>
  </property>
  <property name="definitions">
    <list>
      <value>/WEB-INF/tiles-config.xml</value>
    </list>
  </property>
</bean>
```

2. Change the “viewClass” property of the “viewResolver” bean from `JstlView` to `TilesJstlView`. You can also delete the “prefix” and “suffix” values in this bean definition. Below is the replacement “viewResolver” bean definition:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResource
        ViewResolver">
  <property name="requestContextAttribute">
    <value>rc</value>
  </property>
  <property name="viewClass">

<value>org.springframework.web.servlet.view.tiles.TilesJstlView</value>
  </property>
</bean>
```

The `TilesJstlView` class will now resolve any view names to definition names. For instance, in `UserController`, it returns the “userList” view from the `handleRequest()` method. This will now render the “userList” definition.

```
return new ModelAndView("userList", "users", mgr.getUsers());
```

Another example is the “formView” property of the `UserFormController` class. In *action-servlet.xml*, it’s set to “userForm,” which will render the “userForm” definition.

Step 2: Create a Base Layout

Now you must create a *base layout* JSP. This is basically a template (equivalent to SiteMesh's *decorator*) that controls the layout of the page and where certain components are inserted.

1. Create a `web/layouts/baseLayout.jsp` file. Fill this file with the code below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

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

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
    <title><tiles:getAsString name="title"/></title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <c:set var="ctx" value="{pageContext.request.contextPath}"/>
    <link href="{ctx}/styles/global.css" type="text/css"
rel="stylesheet"/>
    <link href="{ctx}/images/favicon.ico" rel="SHORTCUT ICON"/>
    <!-- HTML & Design contributed by Boer Attila (http://www.calcium.ro)
-->
    <!-- Found at http://www.csszengarden.com/?cssfile=/083/083.css&page=2
-->
</head>
<body>
<a name="top"></a>
<div id="container">
    <div id="intro">
        <div id="pageHeader">
            <h1><span>Welcome to Equinox</span></h1>
            <div id="logo" onclick="location.href='<c:url value="/" />'"
                onkeypress="location.href='<c:url value="/" />'"></div>
            <h2><span>Spring Rocks!</span></h2>
        </div>

        <div id="quickSummary">
            <p>
                <strong>Equinox</strong> is a lightweight version of
                <a href="http://raibledesigns.com/appfuse">AppFuse</a>
                designed
                to accelerate starting a webapp with the
                <a href="http://www.springframework.org">Spring Framework</
a>.
            </p>
            <p class="credit">
                <a href="http://www.csszengarden.com/?cssfile=/083/
083.css">
```

```

        Design and CSS</a> donated by <a href="http://
www.calcium.ro">
        Bo&eacute;r Attila</a>.
    </p>
</div>

<div id="content">
    <%@ include file="/messages.jsp"%>
    <tiles:insert attribute="content"/>
</div>
</div>

<div id="supportingText">
    <div id="underground">
        <c:out value="${underground}" escapeXml="false"/>
    </div>
    <div id="footer"></div>

</div>

<div id="linkList">
    <div id="linkList2">
    </div>
</div>

</div>

</body>
</html>

```

This file is very similar to the *web/decorators/default.jsp* that you created for SiteMesh, except that it uses the “tiles” JSP tag instead of the “decorator” tag.

2. Because of this, you must add the “tiles” tag to the *web/taglibs.jsp* file. Here’s what this file should look like after making this change:

```

<%@ page language="java" errorPage="/error.jsp" %>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
    prefix="html" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles"
    prefix="tiles" %>

```

Step 3: Create Page Definitions

Tiles supports two methods of configuring *page definitions*: configure a page's definition in a JSP, and configure each page definition in an XML file. The second method gives a cleaner separation of concerns and allows your JSPs to be agnostic of the fact that Tiles is using them.

1. To create page definitions for MyUsers, create a *tiles-config.xml* file in **web/WEB-INF** and populate it with the XML below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">

<tiles-definitions>
    <!-- base layout definition -->
    <definition name="baseLayout" path="/layouts/baseLayout.jsp">
        <put name="title" value="MyUsers"/>
    </definition>

    <!-- index definition -->
    <definition name="index" extends="baseLayout">
        <put name="title" value="MyUsers ~ Welcome"/>
        <put name="content" value="/index.jsp"/>
    </definition>

    <!-- user list definition -->
    <definition name="userList" extends="baseLayout">
        <put name="title" value="MyUsers ~ User List"/>
        <put name="content" value="/userList.jsp"/>
    </definition>

    <!-- user form definition -->
    <definition name="userForm" extends="baseLayout">
        <put name="title" value="MyUsers ~ User Details"/>
        <put name="content" value="/userForm.jsp"/>
    </definition>
</tiles-definitions>
```

The above file defines the page titles here instead of in the JSPs.

2. To produce clean HTML in your application, delete the `<title>` elements from *userList.jsp* and *userForm.jsp* in the **web** directory. There is no easy way to control the title in the JSP as with SiteMesh.

3. Configure Spring so it can resolve URLs to Tiles definitions by adding a bean definition. The `UrlFilenameViewController` is declared in the *action-servlet.xml* as follows:

```
<bean id="filenameController"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

4. In order to render the index page (*/index.html*) of MyUsers, configure the “urlMapping” bean with an additional mapping:

```
<prop key="/index.html">filenameController</prop>
```

5. Run **ant remove clean install** to see a similar view to the SiteMesh result ([Figure 5.3](#)) when you go to <http://localhost:8080/myusers/users.html>.

The problem that you’ll experience now is that if you go to <http://localhost:8080/myusers>, it shows the *index.jsp* page with no decoration.

6. Solve this by renaming *index.jsp* to *welcome.jsp* and create a new *index.jsp* with the following contents:

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

<tiles:insert definition="index"/>
```

7. Change *tiles-config.xml* to use the *welcome.jsp* for the content page:

```
<definition name="index" extends="baseLayout">
    <put name="title" value="MyUsers ~ Welcome"/>
    <put name="content" value="/welcome.jsp"/>
</definition>
```

While the previous solution works, it can be a real pain to create two JSPs to solve problems like this one. Therefore, I recommend using a servlet filter to redirect certain URLs to others. Using this solution, you must configure your application so that the root URL invokes the “/index.html” URL. You cannot do this using the `<welcome-file-list>` in *web.xml*, so use Paul Tuckey’s [URL Rewrite Filter](#) to make it happen.

Note: Paul Tuckey’s Rewrite Filter is modeled after [mod_rewrite](#) for the Apache HTTP Server. It redirects or forwards requested URLs in order to create tidy URLs, do browser detection, or gracefully handle moved content.

The *urlrewrite-1.2.jar* is already in **web/WEB-INF/lib**, so you only need to configure it in *web.xml* and add its configuration file.

8. Open *web.xml* and add the following `<filter>` definition:

```
<filter>
  <filter-name>UrlRewriteFilter</filter-name>
  <filter-class>
    org.tuckey.web.filters.urlrewrite.UrlRewriteFilter
  </filter-class>
</filter>
```

9. Add its mapping:

```
<filter-mapping>
  <filter-name>UrlRewriteFilter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

10. Configure this filter's rules by creating an *urlrewrite.xml* file in the **web/WEB-INF** directory:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 1.0//EN"
  "http://tuckey.org/res/dtd/urlrewrite1.dtd">

<urlrewrite>
  <rule>
    <from>/${</from>
    <to type="forward">index.html</to>
  </rule>
  <rule>
    <from>/index.jsp</from>
    <to type="forward">index.html</to>
  </rule>
</urlrewrite>
```

This configuration will route both <http://localhost:8080/myusers> and <http://localhost:8080/myusers/index.jsp> to <http://localhost:8080/myusers/index.html>, thereby invoking Spring's `DispatcherServlet`. The `/index.html` mapping will call the “filenameController” bean, which will use the URL to figure out it needs to render the “index” definition.

Finally, you must fix *index.jsp* so it sets the “underground” content as a request variable for the definition to pick up. With SiteMesh, you were able to set content from the JSP using the `<content>` tag and the `<decorator:getProperty>` tag in your decorator. Tiles does not have similar functionality, but you can mimic this pattern by setting a request attribute with JSTL.

11. Open *web/index.jsp* and change the `<content tag="...">...</content>` to the following:

```
<c:set var="underground" scope="request">
...
</c:set>
```

This text will then be picked up and rendered by the following line in *layouts/baseLayout.jsp*:

```
<c:out value="${underground}" escapeXml="false"/>
```

Note: This solution also works with SiteMesh if you'd prefer not to use its proprietary `<content>` tags.

The last couple of sections have given you the knowledge you need to configure the two most popular page layout and decoration engines. They both work with Spring's MVC framework and they're both relatively easy to configure (especially now that you have this guide). I recommend SiteMesh, because it is easier to work with, especially with new applications. However, it's up to you to choose the one you prefer.

Validating the Spring Way

Currently, Spring does not ship with the Commons Validator setup that the MyUsers app uses. It does, however, have a fairly simple validation system you can use, if you don't want to use Commons Validator. All you need to do is create a class that implements `org.springframework.validation.Validator`, which has the following methods:

```
/**
 * Return whether or not this object can validate objects
 * of the given class.
 */
boolean supports(Class clazz);

/**
 * Validate an object, which must be of a class for which
 * the supports() method returned true.
 * @param obj Populated object to validate
 * @param errors Errors object we're building. May contain
 * errors for this field relating to types.
 */
void validate(Object obj, Errors errors);
```

1. Disable Commons Validator in your *userForm.jsp* file. Simply remove the `onsubmit` attribute of the `<form>` tag:

```
<form method="post" action="<c:url value="/editUser.html"/>">
```

2. Create a new class named `UserValidator` in the `src/org/appfuse/web` directory:

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserValidator implements Validator {
    private Log log = LogFactory.getLog(UserValidator.class);
    public boolean supports(Class clazz) {
        return clazz.equals(User.class);
    }

    public void validate(Object obj, Errors errors) {
        if (log.isDebugEnabled()) {
            log.debug("entering 'validate' method...");
        }

        User user = (User) obj;

        if (user.getLastName() == null ||
            "".equals(user.getLastName().trim())) {
            errors.rejectValue("lastName", "errors.required",
                new Object[] { "Last Name" },
                "Value required.");
        }
    }
}
```

In the preceding code, this `Validator` only supports the `User` class and its `validate()` method will return an error if the `lastName` variable is empty.

3. To configure this `Validator` in `action-servlet.xml`, simply add the following bean definition:

```
<bean id="userValidator" class="org.appfuse.web.UserValidator"/>
```

4. Change the “validator” property of the “userFormController” bean to use “userValidator” instead of “beanValidator”:

```
<property name="validator"><ref bean="userValidator"/></property>
```

5. Run `ant deploy reload` and try to add a new user without a last name. To prove the `UserValidator` is being invoked, check your logs for the debug message when entering the `validate` method. Using this validation mechanism can be very powerful when you want to do more sophisticated validation (such as comparing properties against database values).

The next section reviews setting up Commons Validator for a Spring application and using it to verify the `lastName` field is not empty. Then it will show you how you can use both Commons Validator’s declarative validation and the `Validator` interface to create a *validation chain*.

Using Commons Validator

[Chapter 4](#) explained in detail how to use Commons Validator’s declarative validation framework, so I won’t go through all the particulars again. However, since this is the chapter on validation, here is a step-by-step overview of what you need to do:

1. Create a *validation.xml* file in **web/WEB-INF** and define your validation rules in it.
2. [Download](#) the Spring-specific *validation-rules.xml* file and install it in **web/WEB-INF**.
3. Add “validatorFactory” and “beanValidator” bean definitions to *web/WEB-INF/action-servlet.xml*.
4. Configure your *FormController* bean to use “beanValidator” for its “validator” property.

These steps enable server-side validation, but what about client-side validation? The method from *Chapter 4* works, but it includes all of the JavaScript validation functions in the page. A better way is to refer to a standalone JavaScript file that the user’s browser can cache. The following instructions assume you have no client-side validation configured on your form.

1. Add an `onsubmit` attribute to the `<form>` on which you want to enable validation.

```
<form method="post" action="<c:url value="/editUser.html"/>"
      onsubmit="return validateUser(this)">
```

2. At the bottom of the form, add the following lines of code to write JavaScript function calls for the form’s rules and to include the standalone JavaScript file. If you try this in *MyUsers*, make sure to replace the existing `<html:javascript>` tag.

```
<html:javascript formName="user"
  staticJavaScript="false" xhtml="true" cdata="false"/>
<script type="text/javascript"
  src="<c:url value="/scripts/validator.jsp"/>"></script>
```

3. Create a *validator.jsp* file in **web/scripts** (you must create the *scripts* directory) with the following code:

```
<%@ page language="java" contentType="javascript/x-javascript" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
  prefix="html" %>

<html:javascript dynamicJavaScript="false" staticJavaScript="true"/>
```

The hardest part about using Commons Validator is the setup and configuration process. Once you have that in place, creating the rules is fairly simple. You can even use XDoclet to generate the rules from your POJOs.

XDoclet

XDoclet is an open source code generation engine. It enables **Attribute-Oriented Programming** for Java. This means that you can add more significance to your code by adding metadata (attributes) to your java sources. This is done in special JavaDoc tags. For more information, please refer to the [XDoclet website](#). The metadata attributes that XDoclet uses are also called *annotations*. This concept has received such praise and use that annotations have been added as a new feature in J2SE 5.

At the time of this writing, this functionality doesn't exist in an XDoclet release, but it is checked into XDoclet's CVS.

1. To generate your validation rules from your POJOs, define a `<webdoclet>` task that uses the `<springvalidationxml>` task. An example is given below:

```
<target name="webdoclet"
  description="Generate web deployment descriptors">
  <taskdef name="webdoclet"
    classname="xdoclet.modules.web.WebDocletTask">
    <classpath>
      <path refid="xdoclet.classpath"/>
    </classpath>
  </taskdef>
  <webdoclet destdir="${webapp.target}/WEB-INF"
    force="${xdoclet.force}"
    mergedir="metadata/web"
    excludedtags="@version,@author"
    verbose="true">
    <fileset dir="src"/>
    <springvalidationxml/>
  </webdoclet>
</target>
```

2. Add `@spring.validator` tags to your POJO's *setters* as follows:

```
/**
 * @spring.validator type="required"
 */
public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

The above code will generate the same *validation.xml* file that you are using in MyUsers, which makes the *lastName* a required field. This example demonstrates how simple XDoclet can make declarative validation. If you'd like more information on XDoclet, please refer to [AppFuse](#) or [build XDoclet from CVS](#).

Chaining Validators

The previous two examples set a “validator” property on the “userFormController”. The bean referenced in this property referred to the custom “userValidator” or to Commons Validator’s “beanValidator”, which reads its rules from an XML file. With Spring MVC you can actually add multiple validators by setting a “validators” property as follows:

```
<property name="validators">
  <list>
    <ref bean="beanValidator"/>
    <ref bean="userValidator"/>
  </list>
</property>
```

This creates a sort of *validation chain* that can do simple validation using Commons Validator and more complex validation with a custom Validator implementation.

Validating in Business Delegates

While validation in the web tier seems to be the most common practice, there is a demand for validation in the business layer as well. Below is a simple example of how to use Spring’s validation in your middle tier.

1. In *UserManagerImpl.java*, you can change the `saveUser()` method to:

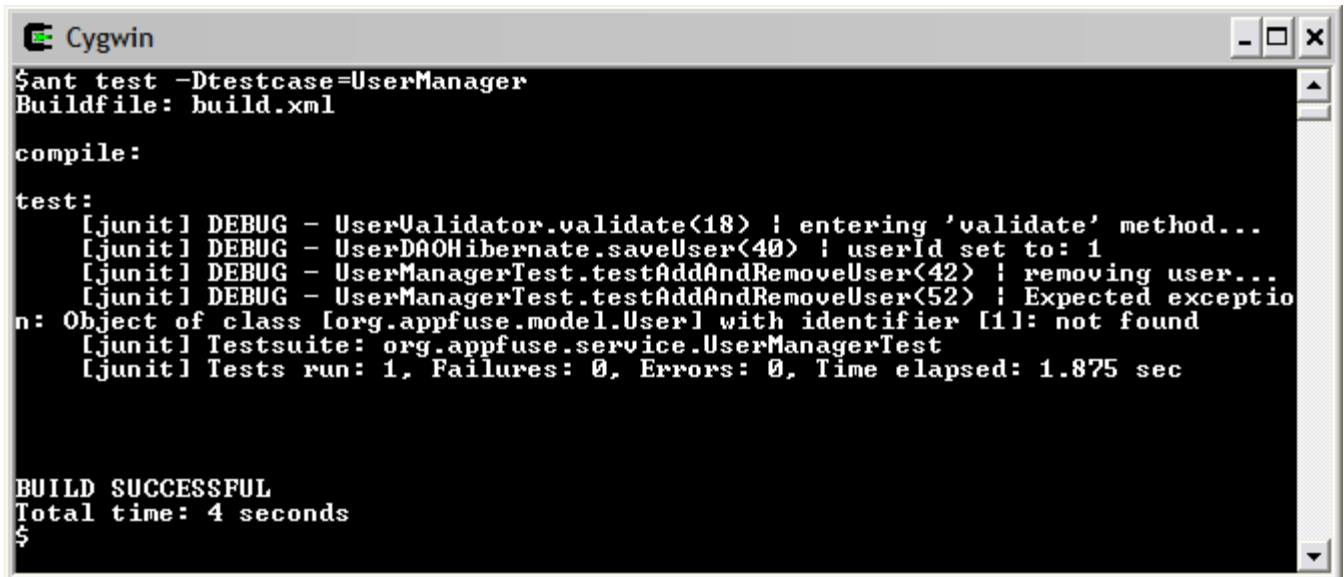
```
public User saveUser(User user) {
    BindException errors = new BindException(user, "user");
    new UserValidator().validate(user, errors);
    if (errors.hasErrors()) {
        throw new RuntimeException("validation failed!", errors);
    }
    dao.saveUser(user);
    return user;
}
```

2. Write a unit test to verify that the exception is thrown in *UserManagerTest.java*:

```
public void testWithValidationErrors() {
    user = new User();
    user.setFirstName("Bill");

    try {
        user = mgr.saveUser(user);
        fail("Validation exception not thrown!");
    } catch (Exception e) {
        log.debug(e.getCause().getMessage());
        assertNotNull(e.getCause());
    }
}
```

3. Run the test using `ant test -Dtestcase=UserManager`; you should see output similar to Figure 5.4.



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

compile:

test:
[junit] DEBUG - UserValidator.validate(18) ! entering 'validate' method...
[junit] DEBUG - UserDAOHibernate.saveUser(40) ! userId set to: 1
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(42) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(52) ! Expected exception
n: Object of class [org.appfuse.model.User] with identifier [1]: not found
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.875 sec

BUILD SUCCESSFUL
Total time: 4 seconds
$
```

Figure 5.4: Result of the `ant test -Dtestcase=UserManager` command

This example has a hard-coded validator, but it is not necessary to hard-code which validator to use. You could add a `setValidator()` method to the `UserManagerImpl` (and its interface), then use dependency injection to set the “validator” property declaratively in your *applicationContext.xml* file. To make this work, you would have to declare your “validator” bean in *applicationContext.xml*, or load *action-servlet.xml* in your test. All in all, it’s much easier to configure and use validation in the web tier.

Spring's Future Declarative Validation Framework

Commons Validator is the only declarative validation framework supported by Spring MVC. However, Keith Donald (a Spring Developer) is working feverishly to develop a more native and robust declarative validation framework. I asked him to provide me with a few details about it, and here are the key features/differentiators he sent me:

- A simple, consistent interface for defining new validation rules (rule providers simply implement a single "boolean test(argument)" method).
- Support for bean property expressions (for example, minProperty must be less than maxProperty). The property access strategy will be pluggable and not limited to java beans (for example, allowing map-backed storage, or buffered "form objects" on the rich client side of the house).
- Support for complex nested expressions (and/or/not), and all relational operators (>, >=, <, <=, !=, ==).
- Support for applying different sets of rules based on context or use-case.
- A reporting subsystem capable of iterating over rule structures, performing validation, and capturing/generating error message results. This allows you to assemble complex rules on-the-fly without having to hard code a lot of static messages; the reporter is capable of generating rule messages from the underlying structures automatically.
- Report field typing hints (the rules associated with a field to let the user know what they're expected to type).
- Integration with Spring Rich Client Platform (RCP) and Spring MVC environments.

From this list, you can see that validation in Spring has a very bright future.

Exception Handling in Controllers

Exception handling is something that every webapp should have. The Servlet API provides a simple mechanism for mapping particular exceptions and error-codes to specific views. In Equinox, for example, the following clause in its *web.xml* file says that “if a page is not found” go to the 404.jsp page:

```
<error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
</error-page>
```

The XML below says that any “500: Internal Server Errors” should go to *error.jsp*. If you’re seeing a lot of 500 errors when developing your app, you need better exception handling.

```
<error-page>
  <error-code>500</error-code>
  <location>/error.jsp</location>
</error-page>
```

In addition to *web.xml* error-pages, it’s a good idea to tell your JSP pages to go to an error page when they encounter an exception. The MyUsers application does this in *web/taglibs.jsp*, where you specify `errorPage="/error.jsp"`. In *web.xml*, you can additionally declare certain Exceptions go to certain pages using the `<exception-type>` mapping:

```
<error-page>
  <exception-type>
    org.appfuse.service.UserNotFound
  </exception-type>
  <location>/userNotFound.html</location>
</error-page>
```

Warning! SiteMesh [has a bug in Tomcat](#); it will not decorate `<error-page>` mappings in your *web.xml* file. If you want to map exceptions in *web.xml*, write these pages so they can stand alone, without being decorated.

While the Servlet API provides a nice means of handling exceptions, it’s difficult to extract information from the exception and perform logic on that information. It’s difficult to put try/catch statements in Controllers because they take up a fair amount of lines. It’s so much cleaner to simply call a business delegate’s method. An easy way to avoid try/catching exceptions in your Controllers is by adding “throws Exception” to the

The ability to forward to a particular view for a specific exception is also possible with Spring. The easy way to do this is to define a `SimpleMappingExceptionHandler` in your *action-servlet.xml* and specify which Exceptions go to which view name.

1. Add the following definition in *web/WEB-INF/action-servlet.xml*.

```
<bean id="exceptionResolver"
      class="org.springframework.web.servlet.handler.SimpleMappingException
        Resolver">
  <property name="exceptionMappings">
    <props>
      <prop
        key="org.springframework.dao.DataAccessException">
          dataAccessFailure
        </prop>
      </props>
    </property>
  </bean>
```

In the above code, “dataAccessFailure” refers to the name of a view that the “viewResolver” bean can determine. To Tiles users, this might refer to a definition in *tiles-config.xml*.

2. To test that the above mapping works, modify the `UserDAOTest` (in *test/org/appfuse/dao*). Near the bottom of the `testAddAndRemoveUser()` method, you should have the following line:

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

3. Replace this line with the following code to ensure that an exception is thrown when a user isn’t found:

```
try {
    user = dao.getUser(user.getId());
    fail("User found in database");
} catch (DataAccessException dae) {
    log.debug("Expected exception: " + dae.getMessage());
    assertTrue(dae != null);
}
```

4. Modify the `getUser()` method in `UserDAOHibernate` (in *src/org/appfuse/dao*) to throw an exception when a user is not found:

```
public User getUser(Long id) {
    User user = (User) getHibernateTemplate().get(User.class, id);
    if (user == null) {
        throw new ObjectRetrievalFailureException(User.class, id);
    }
    return user;
}
```

5. Verify that everything is working as planned by running `ant test -Dtestcase=UserDAO`.

6. Create a *dataAccessFailure.jsp* file in the **web** folder:

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

<h3>Data Access Failure</h3>
<p>
    <c:out value="${requestScope.exception.message}"/>
</p>

<!--
<%
Exception ex = (Exception) request.getAttribute("exception");
ex.printStackTrace(new java.io.PrintWriter(out));
%>
-->

<a href="<c:url value='/'/>">&#171; Home</a>
```

If you're using the "myusers" project with SiteMesh, this is all you need to do. Tiles users must complete one more step:

7. Add a "dataAccessFailure" definition in *tiles-config.xml*.

```
<definition name="dataAccessFailure" extends="baseLayout">
    <put name="title" value="Data Access Failure"/>
    <put name="content" value="/dataAccessFailure.jsp"/>
</definition>
```

Tip: You can mix and match view classes, such as using JSTL for one view and Tiles for the next. You can do this by using a `ResourceBundleViewResolver` for the "viewResolver" and specifying a properties file with the view's names and paths. You can see an example of this in the Petclinic sample app that ships with Spring. However, this won't help you mix SiteMesh and Tiles.

8. Run **ant deploy reload** and open <http://localhost:8080/myusers/editUser.html?id=100>; you should see an error page stating that this user doesn't exist.

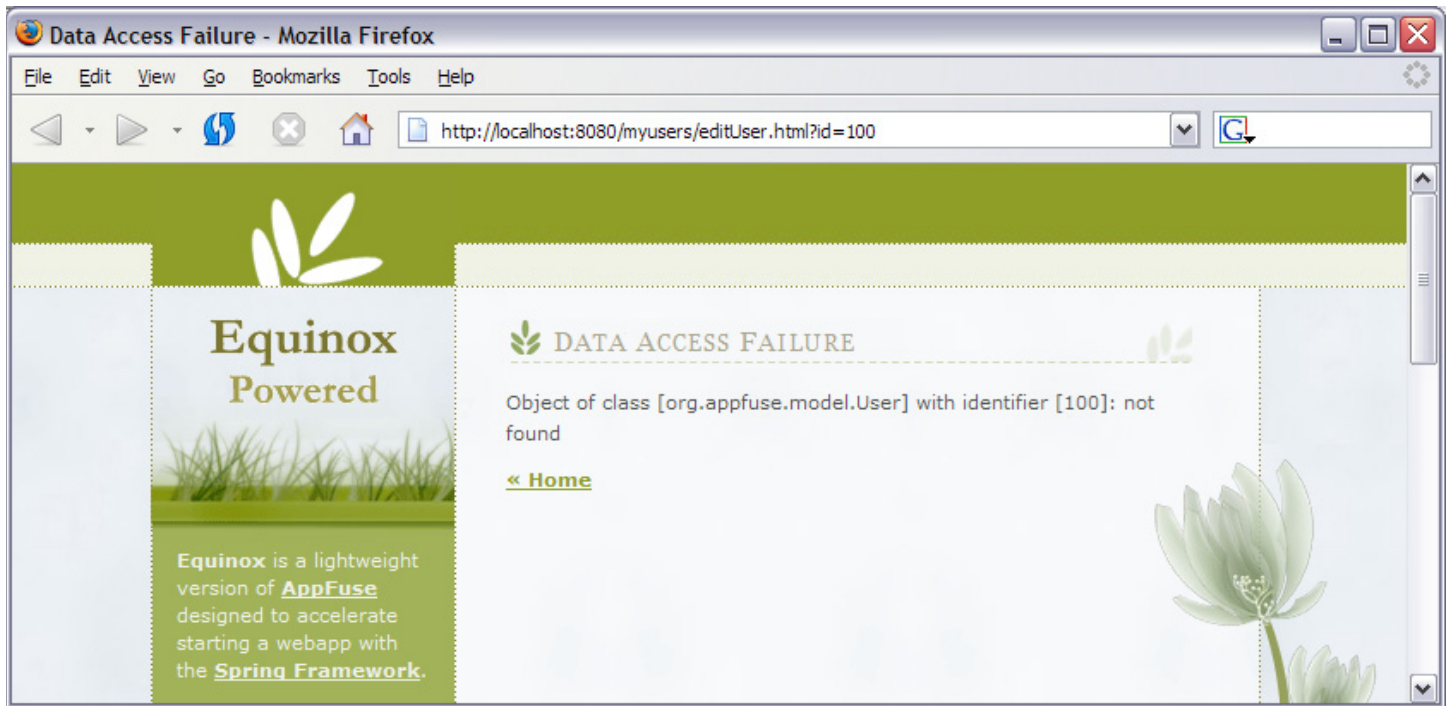


Figure 5.5: Error page stating that the user does not exist

If you need more robust functionality than the `SimpleMappingExceptionHandler` gives you, look at implementing the [HandlerExceptionHandler](#) for a more customized *ExceptionHandler*.

Uploading Files

Every so often, you might run into a requirement to upload files in your web application. The good news is that Spring MVC provides support for uploading files. Better yet, you get a choice of file upload implementations: [Commons FileUpload](#) or [COS FileUpload](#).

Note: If you're using a Servlet 2.2 container, you will not be able to use Spring's built-in file upload support. However, Commons FileUpload has excellent support for doing file uploads with a 2.2 container.

1. Define a bean with an id of “multipartResolver” in *web/WEB-INF/action-servlet.xml*.

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.
        CommonsMultipartResolver">
  <!-- There's a bug in 1.0.2 where a MockServletContext won't
        work with a 'multipartResolver' properly. This is a
        workaround. http://tinyurl.com/39fvt -->
  <property name="uploadTempDir"><value></value></property>
</bean>
```

This bean provides *multipart* support to the `DispatcherServlet`. A *multipart* request is an `HttpServletRequest` that contains both binary and text data. You indicate that you want a form to submit a multipart request by adding an `enctype="multipart/form-data"` attribute to an HTML `<form>`. When Spring detects a multipart request, it simply wraps the current request with its `MultipartHttpServletRequest`, which allows you to access normal `HttpServletRequest` methods, as well as a few new ones (like `getFile(String name)`).

2. To implement a file upload feature in `MyUsers`, add the preceding bean definition, as well as two classes: a `FileUpload` command class and a `SimpleFormController` to handle the uploading process. Create the command class for this example in `src/org/appfuse/web` with the following contents:

```
package org.appfuse.web;

public class FileUpload {
  private byte[] file;

  public void setFile(byte[] file) {
    this.file = file;
  }

  public byte[] getFile() {
    return file;
  }
}
```

3. In the same directory, create a `FileUploadController` class that extends `SimpleFormController`. In the code below, the most important part is the `initBinder()` method, which registers a *PropertyEditor* to grabbing the uploaded file's bytes. Without this, the upload process will fail.

```
package org.appfuse.web;

// use your IDE to organize imports

public class FileUploadController extends SimpleFormController {
    private static Log log =
        LogFactory.getLog(FileUploadController.class);

    protected void initBinder(HttpServletRequest request,
                               ServletRequestDataBinder binder)
        throws ServletException {
        binder.registerCustomEditor(byte[].class,
                                     new ByteArrayMultipartFileEditor());
    }

    protected ModelAndView onSubmit(HttpServletRequest request,
                                    HttpServletResponse response,
                                    Object command, BindException errors)
        throws ServletException, IOException {

        FileUpload bean = (FileUpload) command;
        byte[] bytes = bean.getFile();

        // cast to multipart file so we can get additional information
        MultipartHttpServletRequest multipartRequest =
            (MultipartHttpServletRequest) request;
        CommonsMultipartFile file =
            (CommonsMultipartFile) multipartRequest.getFile("file");

        String uploadDir = getServletContext().getRealPath("/upload/");

        // Create the directory if it doesn't exist
        File dirPath = new File(uploadDir);

        if (!dirPath.exists()) {
            dirPath.mkdirs();
        }

        String sep = System.getProperty("file.separator");
        if (log.isDebugEnabled()) {
            log.debug("uploading to: " + uploadDir + sep +
                      file.getOriginalFilename());
        }
    }
}
```

```
File uploadedFile = new File(uploadDir + sep +
                                file.getOriginalFilename());
FileCopyUtils.copy(bytes, uploadedFile);

// set success message
request.getSession().setAttribute("message", "Upload completed.");

String url = request.getContextPath() + "/upload/" +
             file.getOriginalFilename();

Map model = new HashMap();
model.put("filename", file.getOriginalFilename());
model.put("url", url);

return new ModelAndView(getSuccessView(), "model", model);
}
}
```

4. Put the Controller's bean definition in *action-servlet.xml*:

```
<bean id="fileUploadController"
      class="org.appfuse.web.FileUploadController">
  <property name="commandClass">
    <value>org.appfuse.web.FileUpload</value>
  </property>
  <property name="formView"><value>fileUpload</value></property>
  <property name="successView">
    <value>fileUpload</value>
  </property>
</bean>
```

5. Define the URL to access this Controller by adding another `<prop>` to the "urlMapping" bean:

```
<prop key="/fileUpload.html">fileUploadController</prop>
```

6. Create *web/fileUpload.jsp* with an upload form and a link to display an uploaded file:

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

<h3>File Upload</h3>

<c:if test="${not empty model.filename}">
  <p style="font-weight: bold">
    Uploaded file (click to view): <a
  href="${model.url}">${model.filename}</a>
</p>
</c:if>

<p>Select a file to upload:</p>
<form method="post" action="fileUpload.html" enctype="multipart/form-
data">
  <input type="file" name="file"/><br/>
  <input type="submit" value="Upload" class="button"
    style="margin-top: 5px"/>
</form>
```

SiteMesh users are finished at this point. Tiles users continue below.

7. Add a definition for the “fileUpload” view:

```
<definition name="fileUpload" extends="baseLayout">
  <put name="title" value="My Users ~ File Upload"/>
  <put name="content" value="/fileUpload.jsp"/>
</definition>
```

8. To verify success, open your browser to <http://localhost:8080/myusers/fileUpload.html>. Your browser window should resemble the view in Figure 5.6.

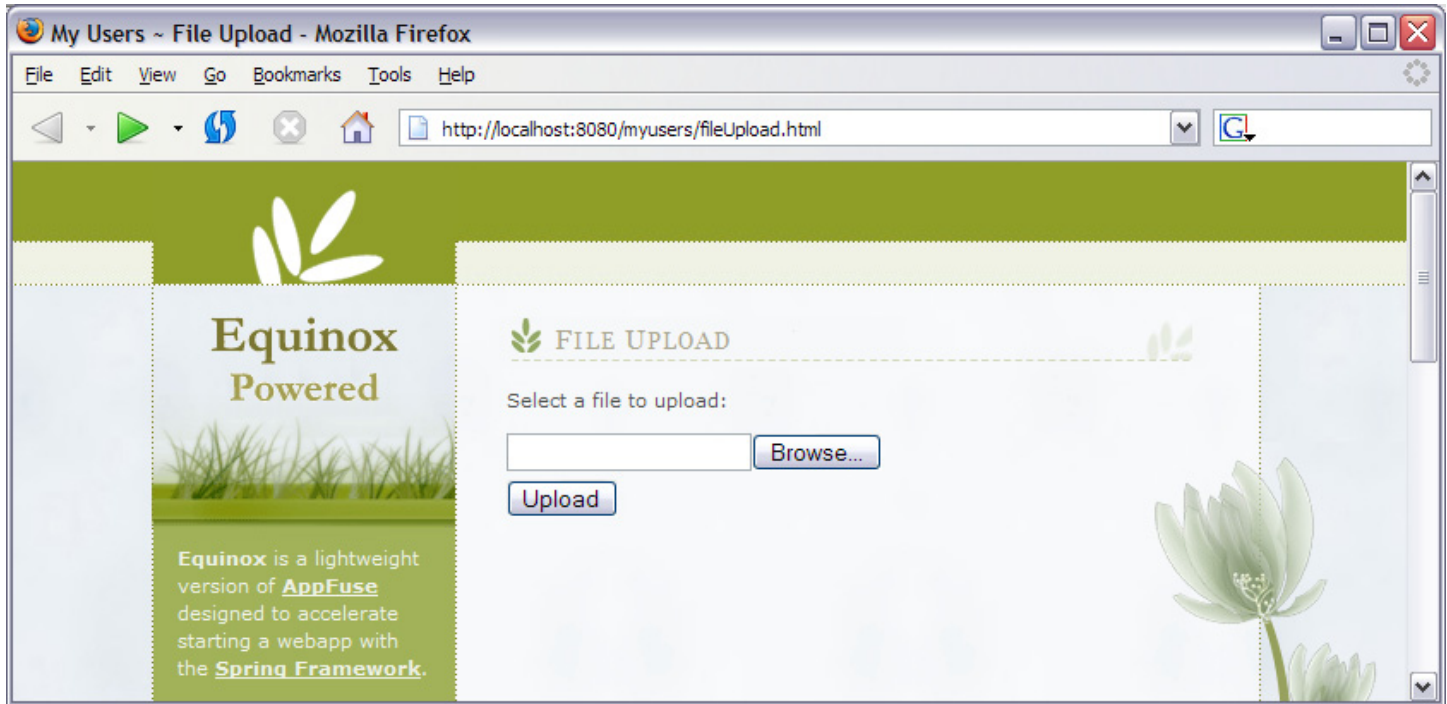


Figure 5.6: "File Upload" page

After uploading a file, you should see a screen like the one below.

Warning! Some files (such as *.html) will not allow viewing by clicking on the filename, so I recommend choosing LICENSE.txt in the myusers directory.

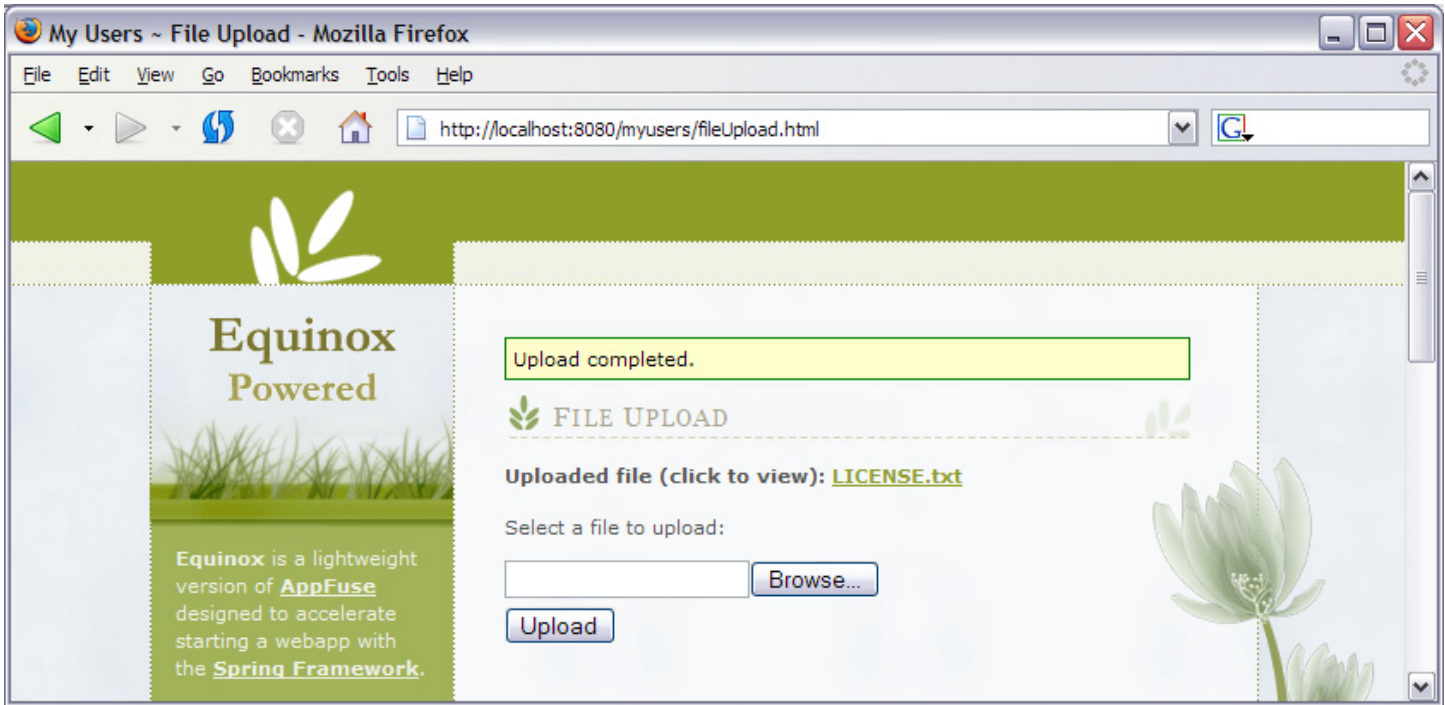


Figure 5.7: A successful file upload

If you were able to replicate the screenshots above, *congratulations!* Now you know how to do File Upload with Spring. *Chapter 8: Unit Testing with Spring* covers TDD on the `FileUploadController`.

Intercepting the Request

Many MVC Frameworks have the ability to specify *Interceptors* for Controllers. Interceptors are classes that intercept the request and perform some sort of logic: controlling flow, setting request attributes, etc. They are similar to ServletFilters, the major difference being that Filters are configured in web.xml and Interceptors are configured in a framework configuration file.

The `HandlerInterceptor` is an interface that contains methods for intercepting the executing of a handler before execution (`preHandle`), after execution (`postHandle`) and after rendering the view (`afterCompletion`). A couple of useful built-in Spring Interceptors are the `OpenSessionInViewInterceptor` and the `UserRoleAuthorizationInterceptor`. The first is Hibernate-specific, while the second prevents certain roles from accessing certain URLs.

Tip: The `OpenSessionInViewInterceptor` has a sister filter (`OpenSessionInViewFilter`) with the same functionality. Both are used for lazy-loading Hibernate-managed objects when the view renders. The Filter is MVC-framework agnostic.

Below is an example of configuring and using the `UserRoleAuthorizationInterceptor`. Configure the `MyUsers` application to protect a particular url-pattern. Lock down the entire application so only users with a “tomcat” role can access it. Then configure the interceptor to allow only users with a “manager” role to upload files.

1. Add the following to the very bottom of the *web.xml* file in **web/WEB-INF**:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>

  <auth-constraint>
    <role-name>tomcat</role-name>
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>My Users</realm-name>
</login-config>

<security-role>
  <role-name>tomcat</role-name>
</security-role>

<security-role>
  <role-name>manager</role-name>
</security-role>
```

2. Run **ant deploy reload** and go to <http://localhost:8080/myusers>. You will be prompted to log in. Use the username “tomcat” and password “tomcat” to log in with the “tomcat” role, and use “admin/admin” to log in with the “manager” role. These users and roles are configured in Tomcat’s *conf/tomcat-users.xml* file (in the `$CATALINA_HOME` directory).
3. To add an Interceptor to only allow *managers* to upload files, define `UserRoleAuthorizationInterceptor` as a bean in *web/WEB-INF/action-servlet.xml*:

```
<bean id="managersOnly"
  class="org.springframework.web.servlet.handler.UserRoleAuthorization
    Interceptor">
  <property name="authorizedRoles">
    <value>manager</value>
  </property>
</bean>
```

4. Add a new `SimpleUrlHandlerMapping` that uses this interceptor on the “/fileUpload.html” mapping:

```
<bean id="managerMappings"
      class="org.springframework.web.servlet.handler.SimpleUrlHandler
        Mapping">
  <property name="interceptors">
    <list>
      <ref bean="managersOnly"/>
    </list>
  </property>
  <property name="mappings">
    <props>
      <prop key="/fileUpload.html">
        fileUploadController
      </prop>
    </props>
  </property>
</bean>
```

5. Close your browser (to logout) and reopen to <http://localhost:8080/myusers/fileUpload.html>. The login prompt displays. If you log in as “tomcat/tomcat,” you will see a 403 error page, which means “access denied.”

Tip: You can easily customize this page by specifying a 403 `<error-page>` in *web.xml*.

6. Log in again with “admin/admin,” but remember to close your browser to erase your previous login credentials.

This is just a simple example of using an Interceptor to control access to a URL. You could easily use this same configuration with a different class and URL mapping to do other things (for example, to make sure certain attributes are always in the request).

Sending E-Mail

E-mail is an excellent notification system; it also can act as a rudimentary workflow system. Spring makes it easy to send e-mail, and it hides the complexity of the underlying mail system. The main interface of Spring's mail support is called `MailSender`. It also has a `SimpleMailMessage` class that encapsulates common attributes of a message (*from*, *to*, *subject*, *message*). If you want to send e-mails with attachments, you can use the `MimeMessagePreparator` to create and send messages.

Create a simple example in the `FileUploadController` to send an e-mail when the file upload has completed. In *Chapter 9: AOP*, you will extract this logic out into a `NotificationAdvice` class.

1. Add a "mailSender" bean to *action-servlet.xml*. The "host" property should match an SMTP server that does not require authentication. If your host requires authentication, you can add "username" and "password" properties:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host"><value>localhost</value></property>
</bean>
```

Note: While your project will build without any JavaMail JARs in your classpath, you will not be able to instantiate the `ApplicationContext` in tests. To fix this, add *activation.jar* and *mail.jar* to the classpath in *build.xml*.

2. Add a property and setter for `MailSender` in the `FileUploadController` class. At the same time, add a property/setter combination for a `SimpleMailMessage`. Setting the `SimpleMailMessage` with dependency injection allows you to configure a default *from* and *subject* in *action-servlet.xml*.

```
private MailSender mailSender;
private SimpleMailMessage message;

public void setMailSender(MailSender mailSender) {
  this.mailSender = mailSender;
}

public void setMessage(SimpleMailMessage message) {
  this.message = message;
}
```

3. To specify the defaults values for an e-mail, add the XML below to *action-servlet.xml*:

```
<bean id="mailMessage"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="from">
        <!-- The <value> and CDATA below must be on the same line -->
        <value><![CDATA[Uploader <spring@sourcebeat.com>]]></value>
    </property>
    <property name="subject">
        <value>File finished uploading</value>
    </property>
</bean>
```

4. Modify the “fileUploadController” bean definition to inject the “mailSender” and “message” properties.

```
<bean id="fileUploadController"
    class="org.appfuse.web.FileUploadController">
    <!-- other properties hidden for brevity -->
    <property name="mailSender"><ref bean="mailSender"/></property>
    <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

5. Add the following code to the end of the `onSubmit()` method in `FileUploadController` (before returning the `ModelAndView`):

```
// Notify user that file has finished uploading
SimpleMailMessage msg = new SimpleMailMessage(this.message);
msg.setTo("springlive@raibledesigns.com");
msg.setText("File \"" + file.getOriginalFilename() +
    "\" has finished uploading.");
try {
    mailSender.send(msg);
} catch (MailException ex) {
    log.error(ex.getMessage());
}
```

Tip: Be sure to change the e-mail address for `msg.setTo()` or you'll just be sending the e-mail to me!

This example simply logs exceptions because it's not critical that this message be sent. If notification e-mails are critical in your application, handle the exception with a `SimpleMappingExceptionResolver`.

6. To test the previous configuration (you must have access to a SMTP server), run **ant deploy reload**, go to <http://localhost:8080/myusers/fileUpload.html> and login as “admin/admin.” Then upload a file and wait for the e-mail. You should see a similar result as the e-mail below.

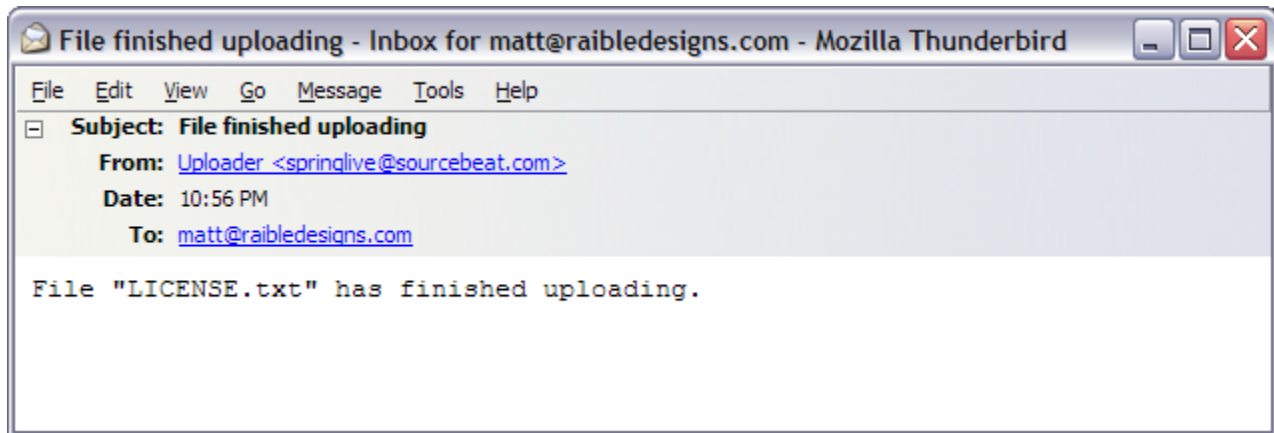


Figure 5.8: Successful e-mail message

Tip: You can also configure the “mailSession” bean to use a `MailSession` object from JNDI. You can learn more about this on TheServerSide.com, as well as how to use Velocity templates in the article titled [“Sending Velocity-based E-Mail with Spring.”](#)

Summary

This chapter covered several advanced Spring MVC topics. You learned how to configure and use the two most popular page decoration frameworks: SiteMesh and Tiles. Validation is an essential part of web application, and you saw how to configure Commons Validator, as well as how to implement a native Spring Validator. Exception handling is important too, and you should now have a grasp of how to throw and handle exceptions. Uploading files is fairly easy once you have an example, which you now have from this chapter. Interceptors are similar to Servlet Filters, but Spring has some built-in ones that can be quite useful, like the `UserRoleAuthorizationInterceptor`. Finally, you saw how using e-mail as a notification mechanism is much easier by using Spring's JavaMail support.

Chapter 6 explores the different views that Spring supports, including Velocity, FreeMarker, XML/XSL, XSLC and PDF.

If you have any feedback on this chapter, or any of the others, please e-mail me at mattr@sourcebeat.com. Remember, this is a dynamic book that will live and breathe for quite some time. I can easily update or clarify any information between monthly releases. Also, if you'd like some material covered more in-depth, don't hesitate to ask!