



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

Dedication	9
Acknowledgments	10
About the Author	11
Preface	12
Introduction	13

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
Add 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()	73
init-method	73
Ready State	74
Destroying Beans	74
The ApplicationContext: Talking to Your Beans	75
Get That Context!	75
Tips for Unit Testing and Loading Contexts	76
Internationalization and MessageSource	77
Event Publishing and Subscribing	78
A Closer Look at MyUser's applicationContext.xml	78
Summary	79

Chapter 4: Spring's MVC Framework 80

Unit Testing Spring Controllers	84
Configure DispatcherServlet and ContextLoaderListener	85
Modify web.xml to Use Spring's DispatchServlet	85
Remove Struts and add Spring Files	87
Create Unit Test for UserController	88
Create UserController and Configure action-servlet.xml	90
Create userList.jsp to Display List of Users	93
Create Unit Test for UserFormController	97
Create UserFormController and Configure It in action-servlet.xml	100
Create userForm.jsp to Allow Editing User's Information	104
Configure Commons Validator for Spring	106
SimpleFormController: Method Lifecycle Review	110
Spring's JSP Tags	113
Summary	114

Chapter 5: Advanced MVC: Templates, Validation, Exceptions and Uploading Files 115

Templating with SiteMesh	117
Installation and Configuration	118
Step 1: Configure SiteMesh in web.xml	119
Step 2: Create Configuration Files	120
Step 3: Create a Decorator	121
Templating with Tiles	124
Installation and Configuration	124
Step 1: Configure Spring to Recognize Tiles	125
Step 2: Create a Base Layout	126
Step 3: Create Page Definitions	128
Validating the Spring Way	132
Using Commons Validator	134
XDoclet	135
Chaining Validators	136
Validating in Business Delegates	136
Spring's Future Declarative Validation Framework	138
Exception Handling in Controllers	139
Uploading Files	143
Intercepting the Request	149
Sending E-Mail	152
Summary	155

Chapter 6: View Options 156

Views and ViewResolvers	157
Testing the View with jWebUnit	159
JSP	162
View Resolver Configuration	162
JSTL	164
The Display Tag	164
Tiles	166
Velocity	167
Using Velocity in MyUsers	168
View Resolver Configuration	168
SiteMesh and Velocity	170
Edit web.xml	172
Create Velocity Templates	172
userList.vm	173
userForm.vm	173
fileUpload.vm	175
dataAccessFailure.vm	175
Deploy and Test	175

Summary of Velocity	176
FreeMarker	177
View Resolver Configuration	177
SiteMesh and FreeMarker	178
Edit web.xml	181
Create FreeMarker Templates	182
userList.ftl	183
userForm.ftl	183
fileUpload.ftl	185
dataAccessFailure.ftl	185
Deploy and Test	186
XSLT	187
Create the View Class	188
Deploy and Test	191
Excel	192
Create the View Class	192
Deploy and Test	193
PDF	194
Create the View Class	194
Deploy and Test	195
Summary	196

Chapter 7: Persistence Strategies: Hibernate, iBATIS, JDBC, JDO and OJB . 199

Overview	200
Preparing for the Exercises	202
Hibernate	203
Dependencies	204
Configuration	205
Test It!	209
MySQL Configuration	211
Caching	212
Lazy-Loading Dependent Objects	214
Community and Support	215
iBATIS	216
Dependencies	217
Configuration	218
Test It!	222
Caching	222
Retrieving Generated Primary Keys	223
Community and Support	225
Spring JDBC	226

Mapping ResultsSets to Objects	227
Executing Update Queries	228
Retrieving Generated Primary Keys	229
Dependencies	229
Configuration	230
Test It!	232
Community and Support	233
JDO	234
JDO and Primary Keys	234
Dependencies	235
Configuration	236
Test It!	241
Caching	244
Community and Support	245
OJB	246
Dependencies	246
Configuration	247
Test It!	250
Caching	250
Community and Support	250
Summary	251

Chapter 8: Testing Spring Applications 252

Overview	253
JUnit	255
Mock Testing	256
Integration Testing	257
Testing the Database Layer	259
DbUnit	262
Database Switching	264
Overriding Beans for Tests	267
Use JNDI DataSource in Tests	269
Load Context Once	271
Testing the Service Layer	273
Using Mocks for DAO Objects	274
EasyMock	274
jMock	277
Testing the Web Layer	282
Testing Controllers	282
Spring Mocks	283
Cactus	293

Testing File Upload and E-Mail	298
Testing Views	302
jWebUnit	302
Canoo WebTest	309
Summary	313

Chapter 9: AOP 314

Overview	315
Getting Started	317
Logging Example	317
Definitions and Concepts	320
Pointcuts	321
Weaving Strategies	322
JDK Dynamic Proxies	322
Dynamic Byte-Code Generation	323
Custom Class Loading	323
Language Extensions	323
Convenient Proxy Beans	324
AutoProxy Beans	325
Practical AOP Examples	326
Transactions	326
Caching in the Middle Tier	327
Event Notification	334
Summary	339

To Abbie and Jack – the coolest kids in town.

I'd like to thank my wife, Julie, for being such a great mother and terrific wife. Your ability to entertain Abbie and comfort Jack at the same time is amazing. Thank you for building our home and giving birth to Jack while I wrote this book. Abbie and Jack, thanks for reminding me that work is not as important as playing with you. Your smiles and giggles always make my day.

I'd like to thank my parents, Joe and Barb, for instilling in me the passion to learn. I learned a lot about computers from my Dad's passion for them. Dad, thanks for being such a good role model and great friend. To my sister Kalin, I appreciate your listening to all my stories when we walked home from the bus stop. I can't help but think that my storytelling as a child helped me become an author.

Thanks to Rod Johnson for providing J2EE developers with such a great framework for developing J2EE applications more easily. Juergen, Colin, Keith and the rest of the Spring Framework team – keep up the coding! Your time, commitment and user support are greatly appreciated. Open-source is a great way to develop software, and your team is one of the best.

To the [SourceBeat](#) founders, Matt Filios and James Goodwill, thanks for the opportunity to write and to do it such an innovative way. SourceBeat is a great model from which readers will truly benefit. Dion, I appreciate all the time you spent plowing through the code in *Spring Live* and ensuring all the examples worked. You've been a great technical editor. Amy, thanks for changing all my "we's" to "you's" and causing my words to make sense. Your ability to make a technical book flow is remarkable.

Matt Raible is a J2EE Consultant and Developer living in Denver, Colorado. He grew up in a log cabin in the backwoods of Montana, without any electricity or running water. He loves Montana and its delicious huckleberries and tries to visit a few times a year.

Matt has been developing websites since before Netscape 1.0. He wrote a lot of HTML, JavaScript and CSS while in college in the 1990s. When he graduated with degrees in Russian, International Business and Finance, he found that he actually liked computers more and became a consultant.

Matt has been developing Java web applications since 1999 and is President of Raible Designs, Inc. He is also a member of the J2EE 5.0 Expert Group and hopes to help them make J2EE easier for developers. Rather than just talking about technologies that make things easier, Matt also uses them in his open-source application starter kit called [AppFuse](#). He is actively involved in many open-source projects and blogs about his experiences regularly on www.raibledesigns.com.

Thank you for buying a subscription to *Spring Live*. Unlike traditional books, this book will change, grow and become better over the next year. Your subscription entitles you to have a say in what goes into this book over the next 12 months. While the 1.0 version has been fun to write, the updates to this book are what excite me.

Together, we can act as a team to produce an in-depth, pragmatic and interesting book about the Spring Framework. As you provide feedback, I can improve existing content in the book and add new content that interests you. This book is yours as much as it is mine; let's make it great together.

Spring Live is an introduction to the Spring Framework. Version 1.0 targets readers who have no experience with Spring. The update chapters will target the advanced Spring user. As the Spring Framework grows and improves over the next year, you will need a book like *Spring Live* to keep up and get your job done. If you know of projects that are coming up and using a certain technology in Spring, let me know; maybe I can write a chapter on it before you start.

This book will be managed similar to an open-source project. There will be monthly releases of the book and they will be announced on my [Spring Live Weblog](#). All of the source code developed in the chapters is available at <http://sourcebeat.com/downloads>. This page has links to download before and after snapshots of each chapter's code.

If you find issues or would like to request features, please enter them in [JIRA](#). For general discussion questions, you can use the [Spring Live Forums](#) or [Equinox User Mailing List](#). I've also setup a [Confluence Wiki](#) for tracking [FAQs](#) and [update chapters](#). You can also send e-mail to me directly at mattr@sourcebeat.com.

Thanks to [Atlassian](#) for the JIRA and Confluence software and to [Javalobby](#) for hosting the weblog and forums.

Sincerely,

Matt Raible

This book is written for Java developers familiar with web frameworks. Its main purpose is to allow Java developers to get up to speed quickly with Spring. It is quite code-intensive because code examples are more useful than theory when learning a new technology.

This book includes a useable sample application that uses Spring and Hibernate to manage the persistence layer and the middle tier. This application uses a simple webapp starter kit I developed called Equinox. Equinox is really just an Ant build file, a directory structure, and the JARs you need to develop a Spring-based J2EE application.

In *Chapter 2*, Equinox is used to quickly develop a simple webapp, called MyUsers, that does CRUD on a database table. JUnit, Struts, Spring and Hibernate are used to develop MyUsers, and HSQL and Tomcat are used as a deployment platform. In *Chapter 4*, MyUsers is refactored to use Spring's MVC framework for the web tier. All the code developed in the chapters below is done using a test-first methodology.

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 view options in Spring's MVC architecture. At the time of this writing, the options are JSP, Velocity, FreeMarker, XSLT, PDF and Excel. This chapter aims to become a reference for configuring all Spring-supported views. It also contains a brief overview how each view works and compares constructing a page in MyUsers with each option. Additionally, it focuses on internationalization for each view option.

Chapter 7: Persistence Strategies: Hibernate, iBATIS, JDBC, JDO and OJB. 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). This chapter refactors the MyUsers application to support both JDBC and iBATIS as persistence framework options. It also implements the UserDao using JDO and OJB to showcase Spring's excellent support for these frameworks.

Chapter 8: Testing Spring Applications explains how to use test-driven development to create high-quality, well-tested, Spring-based applications. You will learn how to test your components using tools like EasyMock, jMock and DBUnit. For the Controllers, you will learn how to use Cactus for in-container testing, and Spring Mocks for out-of-container testing. Lastly, you will learn how to use jWebUnit and Canoo's WebTest for testing the web interface.

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 how AOP might help you.

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:

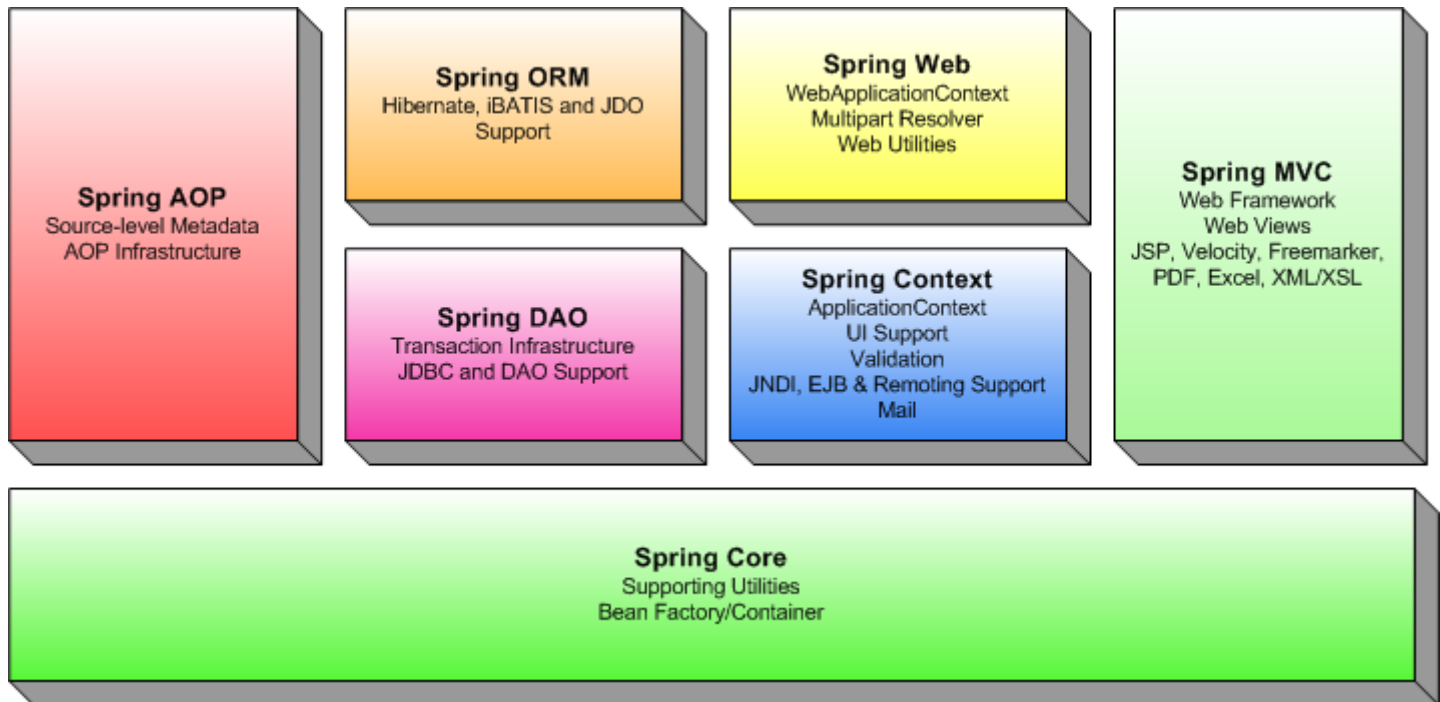


Figure 9.1: Spring's seven modules

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.

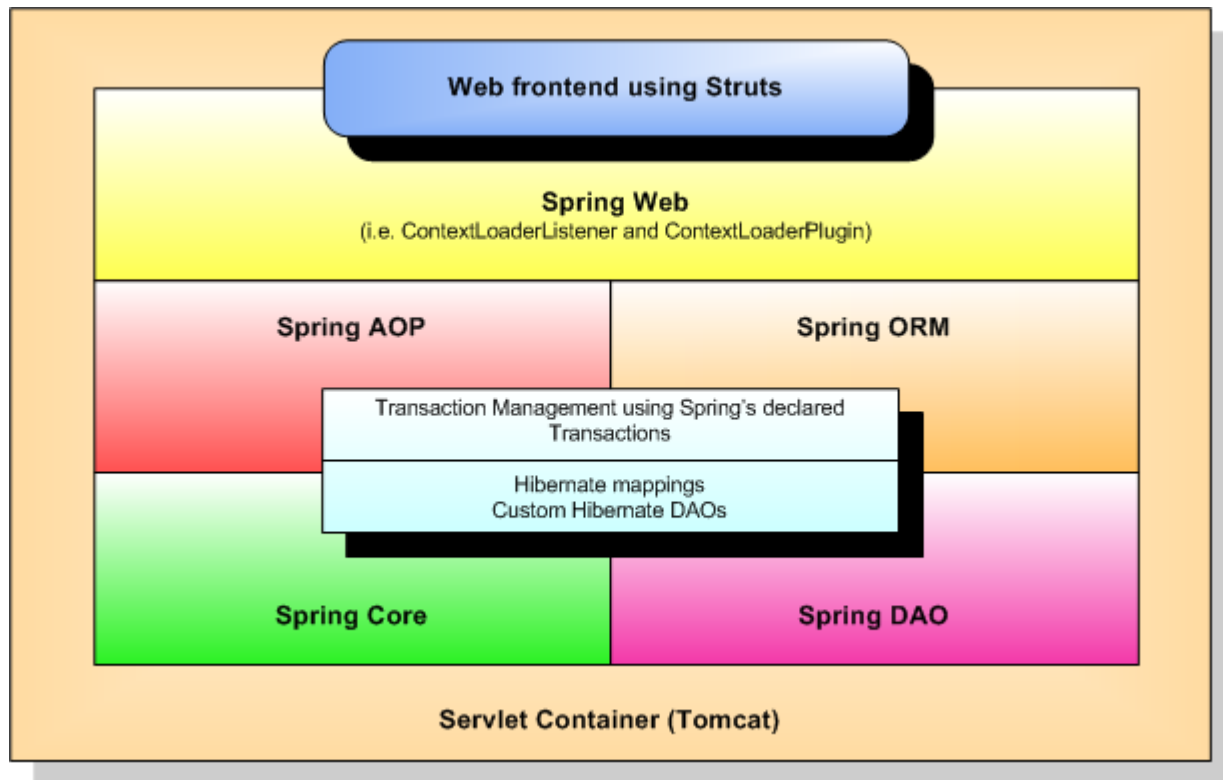


Figure 9.1: MyUsers application architecture using Spring

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.

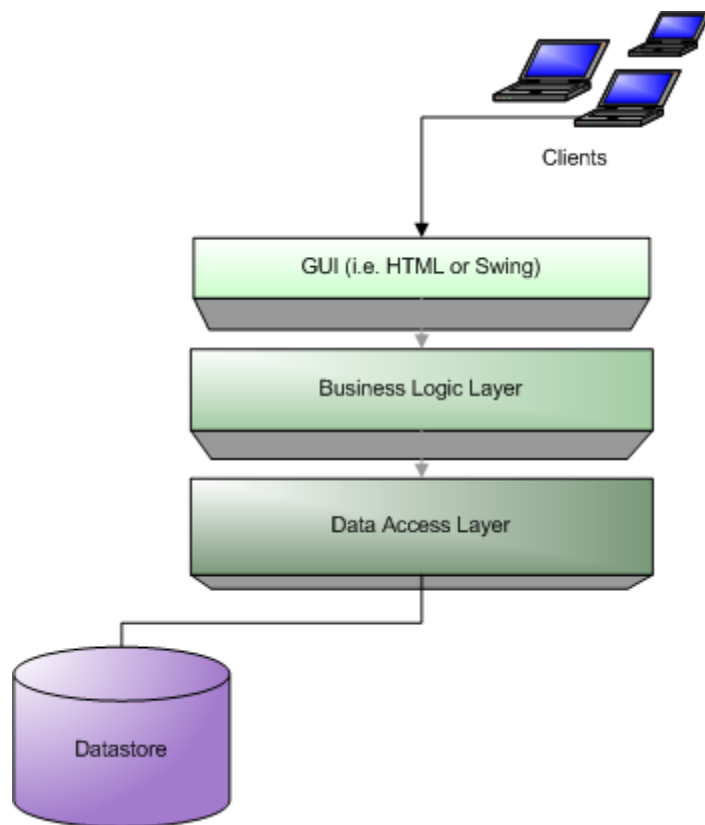


Figure 9.1: A typical J2EE application

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.dao.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 services 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 service, 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.

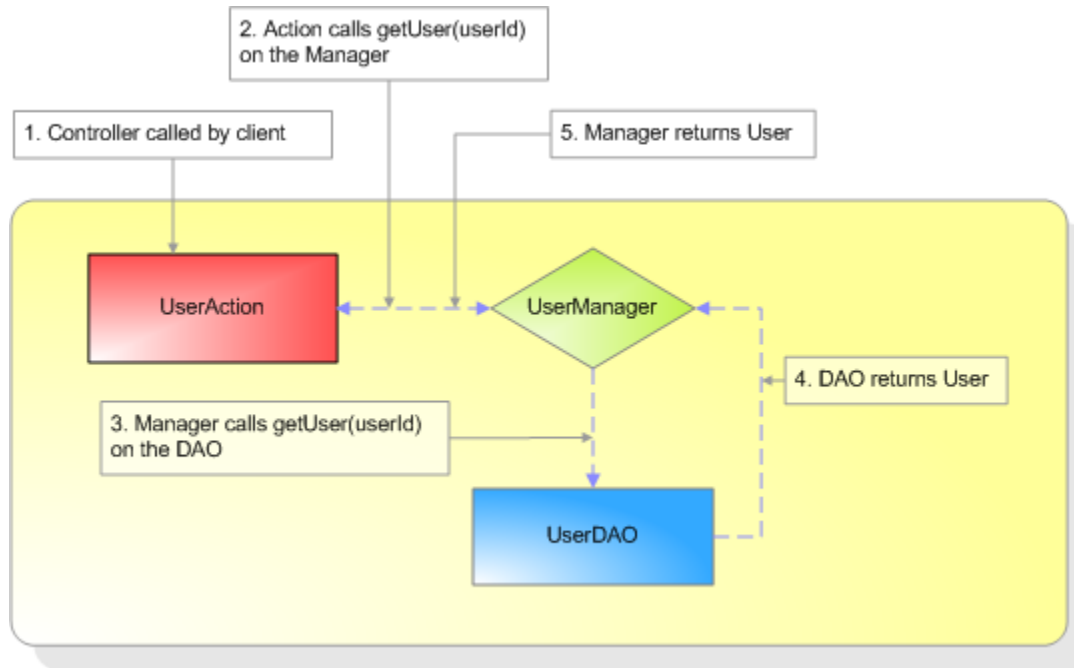


Figure 9.1: MyUsers application flow

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. Add Validation use 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.



NOTE

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:

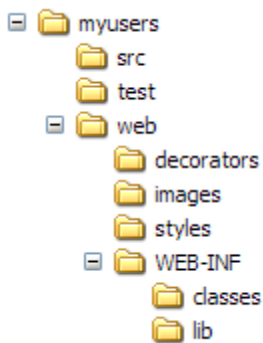


Figure 2.2: MyUsers application directory structure

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>
```

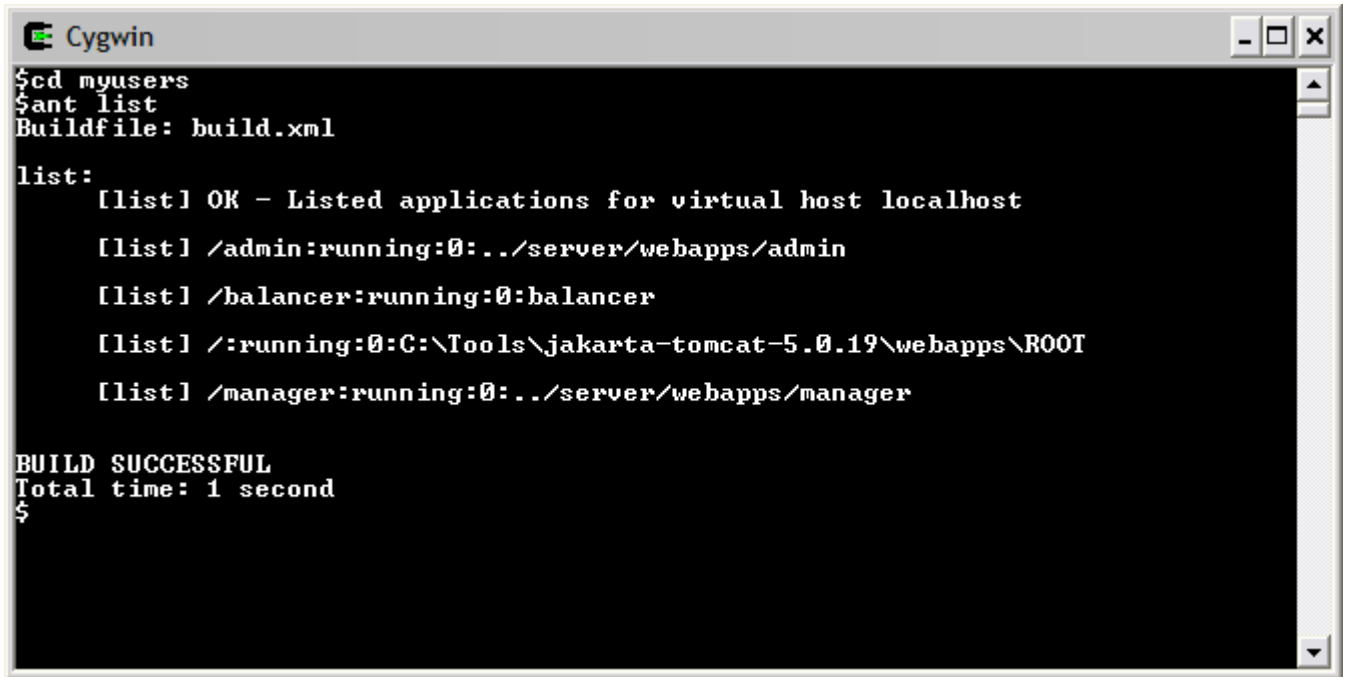
In the targets listed above, several `${tomcat.*}` variables need to be defined. These are in the *build.properties* file in the base directory. By default, they are defined as follows:

```
# 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 9.1: Equinox Welcome page



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. You can also change the database settings to use an absolute path.

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 service 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 {
        super.setUp();
        dao = (UserDAO) ctx.getBean("userDAO");
    }

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

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);
    assertNotNull("primary key assigned", user.getId());
    log.info(user);
    assertNotNull(user.getFirstName());
}

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

    dao.saveUser(user);

    assertNotNull(user.getID());
    assertEquals(user.getFirstName(), "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:



NOTE

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.



NOTE

In a real-world application, these methods should not be in a parent class, but rather in each subclass. Making them abstract in *BaseObject* can help to enforce this rule. [Commonclipse](#) is an Eclipse plugin that can generate these methods for you.

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:

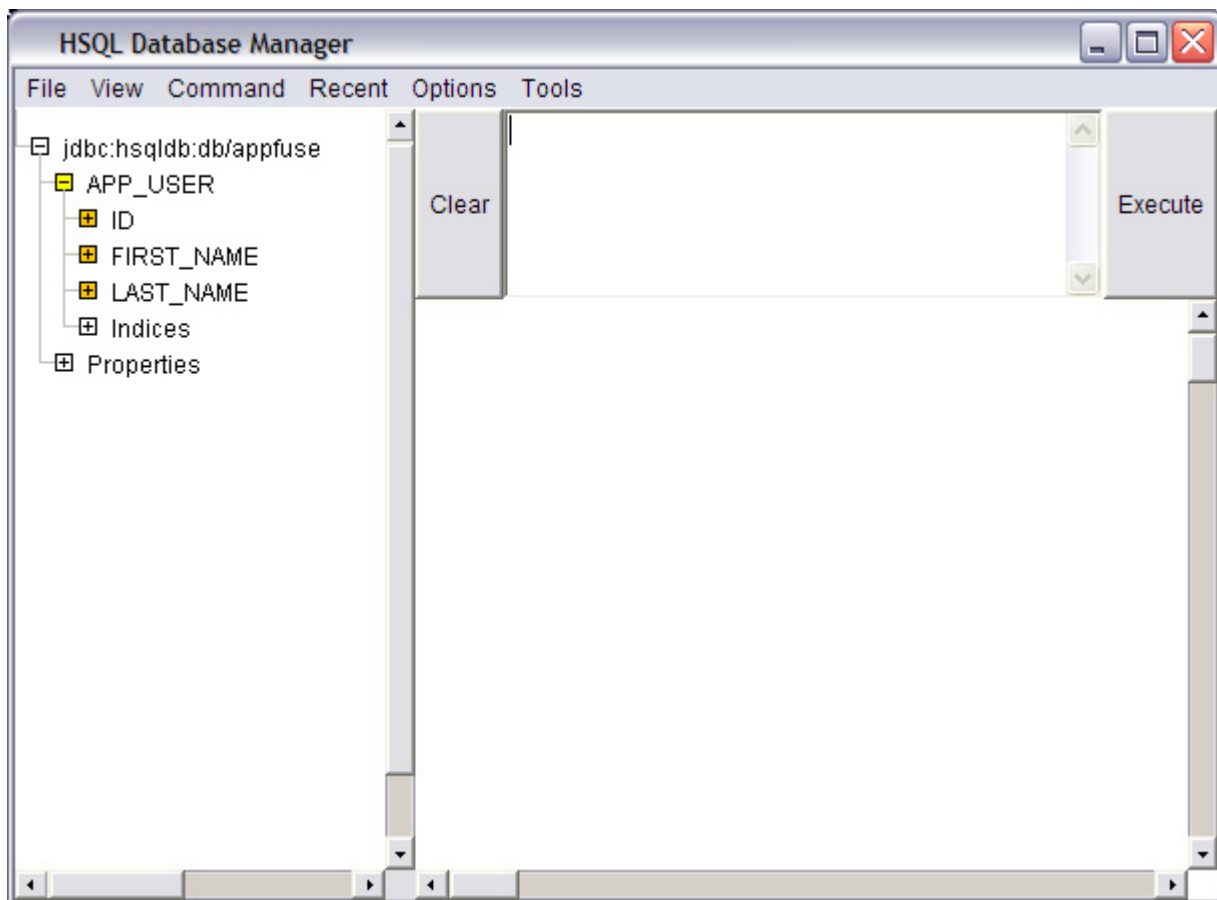


Figure 9.1: HSQL Database Manager

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>
    <!-- Make sure <value> tags are on same line - if they're not,
        authentication will fail -->
    <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.

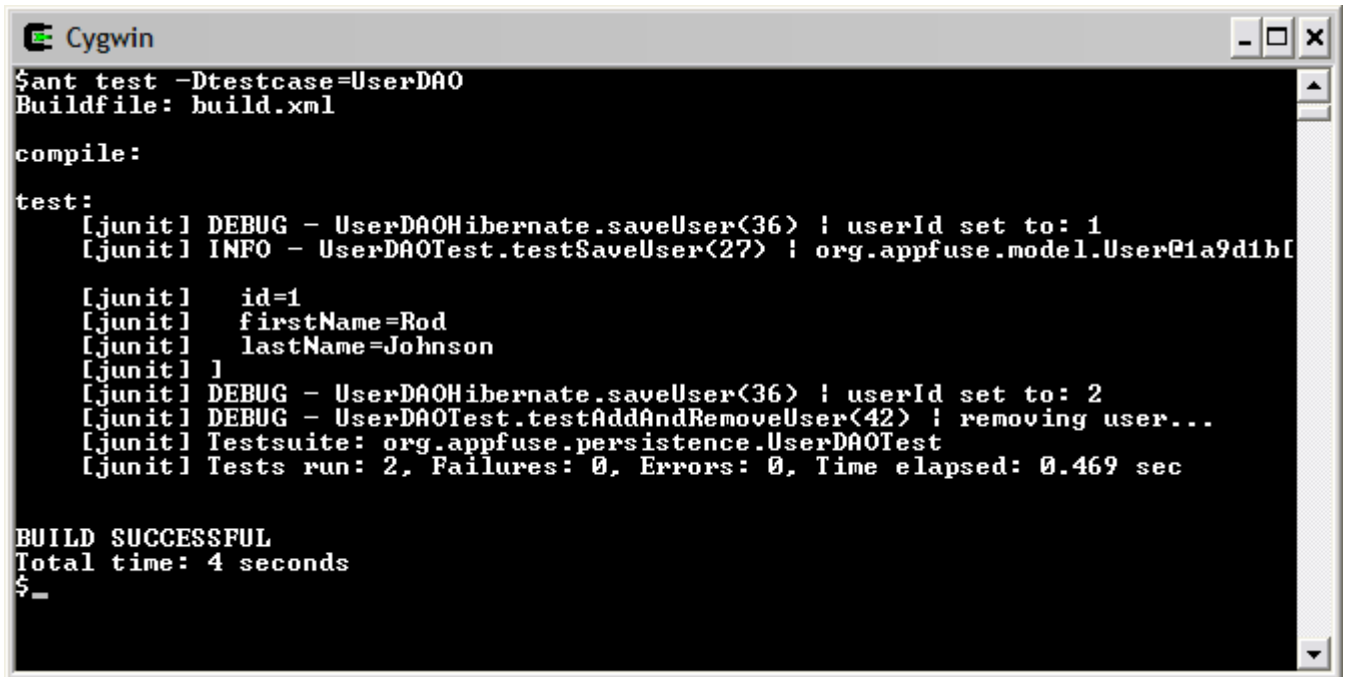
```
<bean id="userDAO"
      class="org.appfuse.dao.hibernate.UserDAOHibernate">
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>
```

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@1a9d1b1f
    [junit] id=1
    [junit] firstName=Rod
    [junit] lastName=Johnson
    [junit] 1
[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 with a business service class.

The main reasons for using a business service class 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 service class 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 service) is also a good place to put any business logic that your application requires.

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

```
package org.appfuse.service;

// use your IDE to handle imports

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

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

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

    // add testXXX methods here
}
```

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);

    assertNotNull(user.getId());

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

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

    user = mgr.getUser(userId);
    assertNull("User object found in database", user);
}
```

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 8* 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:

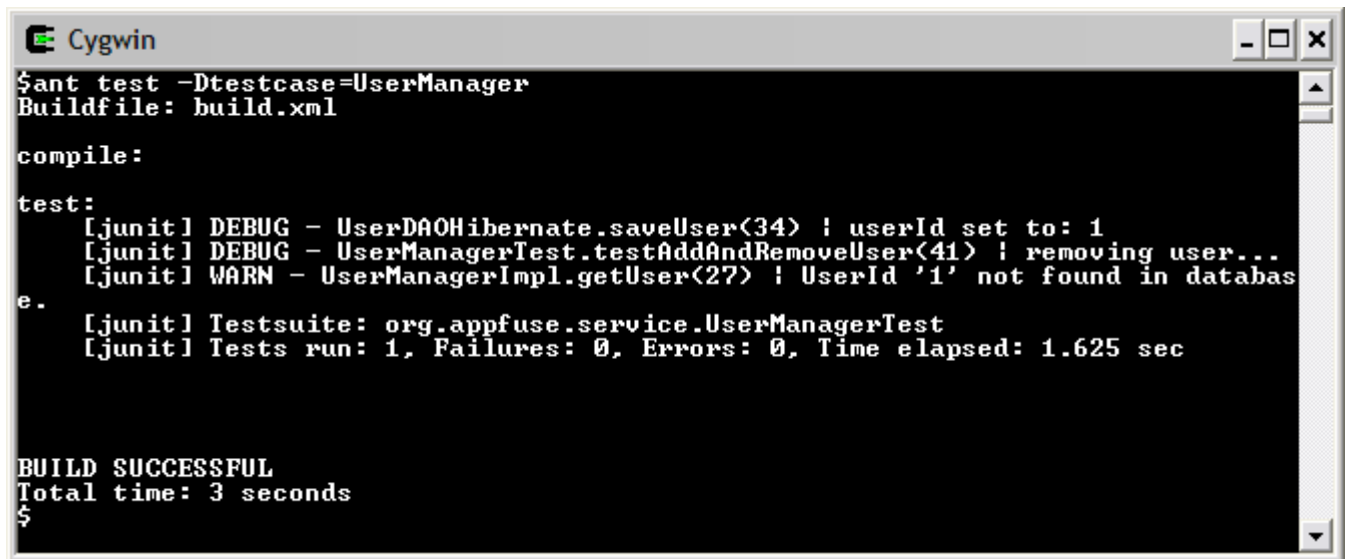
```
<bean id="userManager"
      class="org.springframework.transaction.interceptor.TransactionProxy
      FactoryBean">
  <property name="transactionManager">
    <ref local="transactionManager"/>
  </property>
  <property name="target">
    <ref local="userManagerTarget"/>
  </property>
  <property 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(27) | 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 service class 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 service 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 class extends [MockStrutsTestCase](#) and contains the following code:

```
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:

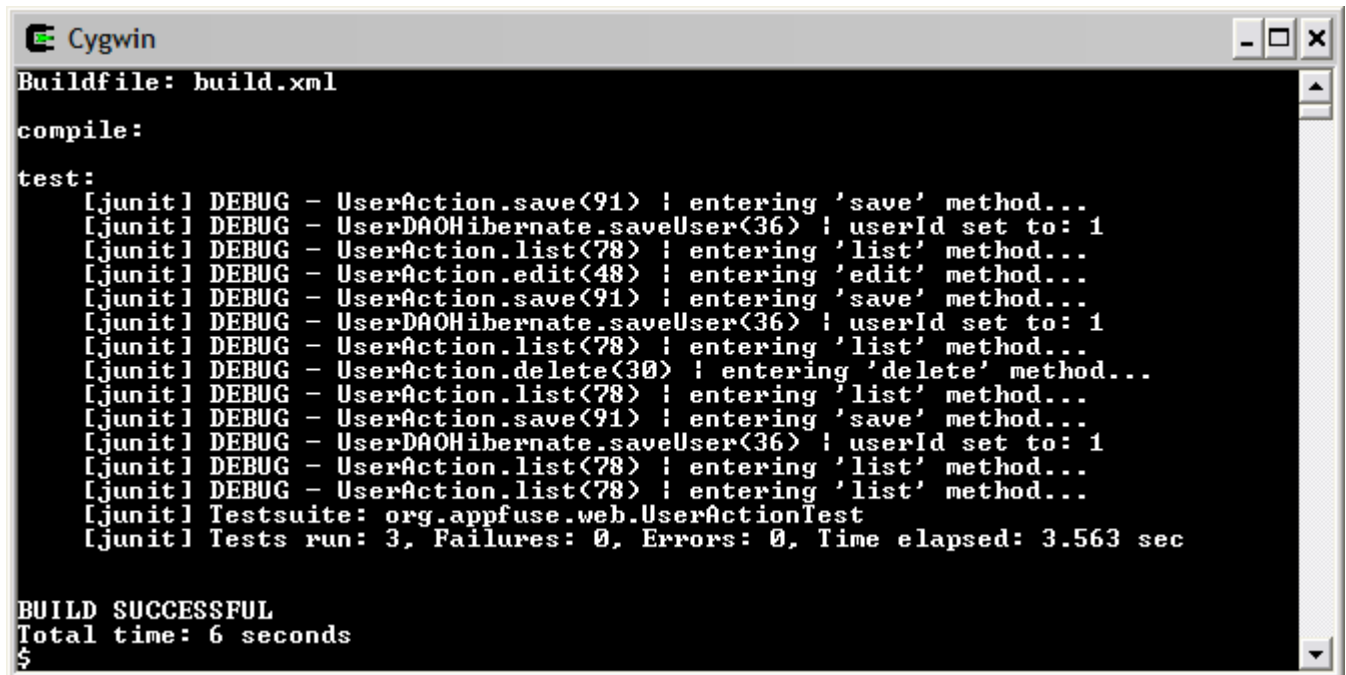
```
user.saved=User has been saved successfully.  
user.missing=No user found with this id.  
user.deleted=User successfully deleted.
```

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>
```

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:

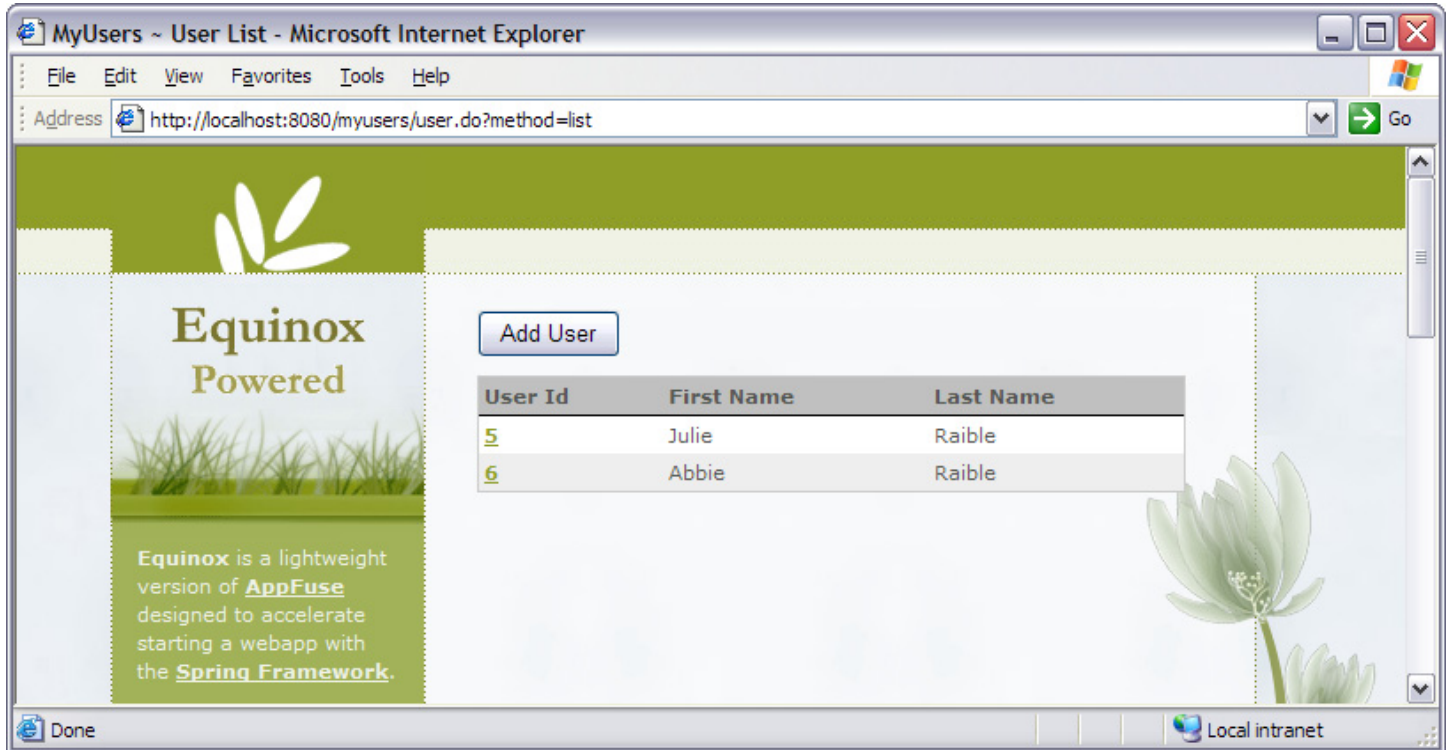


Figure 2.9: Results of `ant deploy reload` command

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").

- 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:

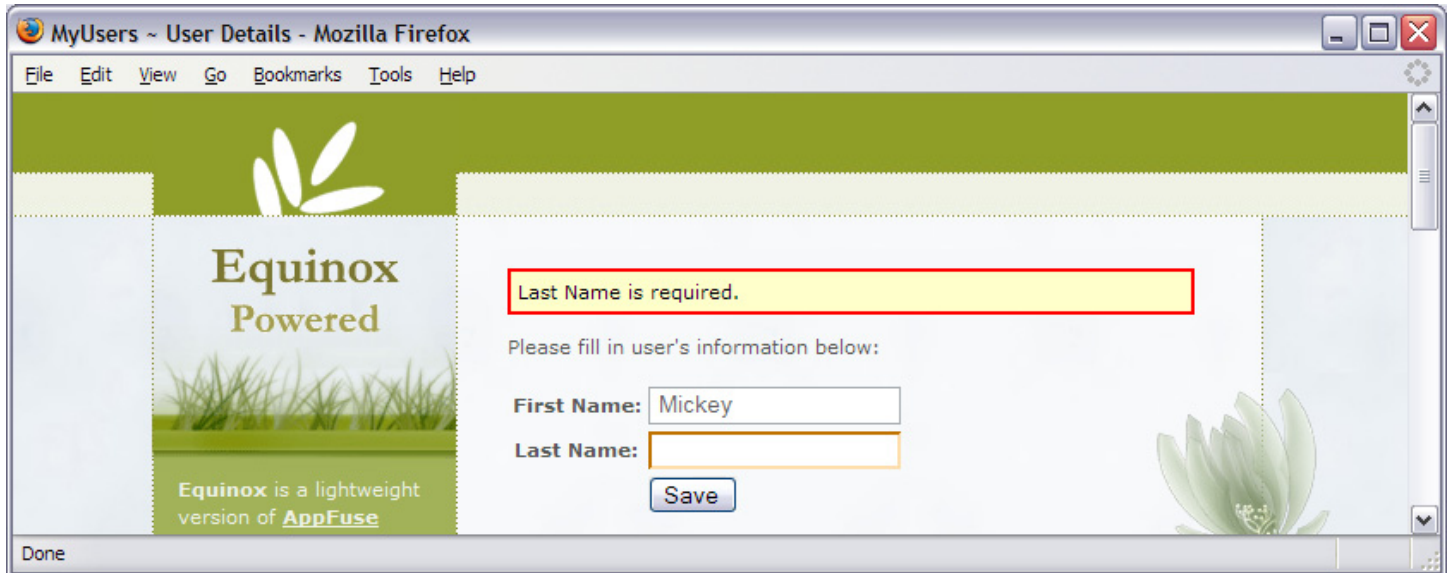


Figure 2.10: Result of the `ant deploy reload` command

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.



NOTE

For reading bean definitions from properties files, you can use [PropertiesBeanDefinitionReader](#). For more information on using this class, see R. J. Lorimer's article on [Alternative Spring Bean Mappings](#).

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.

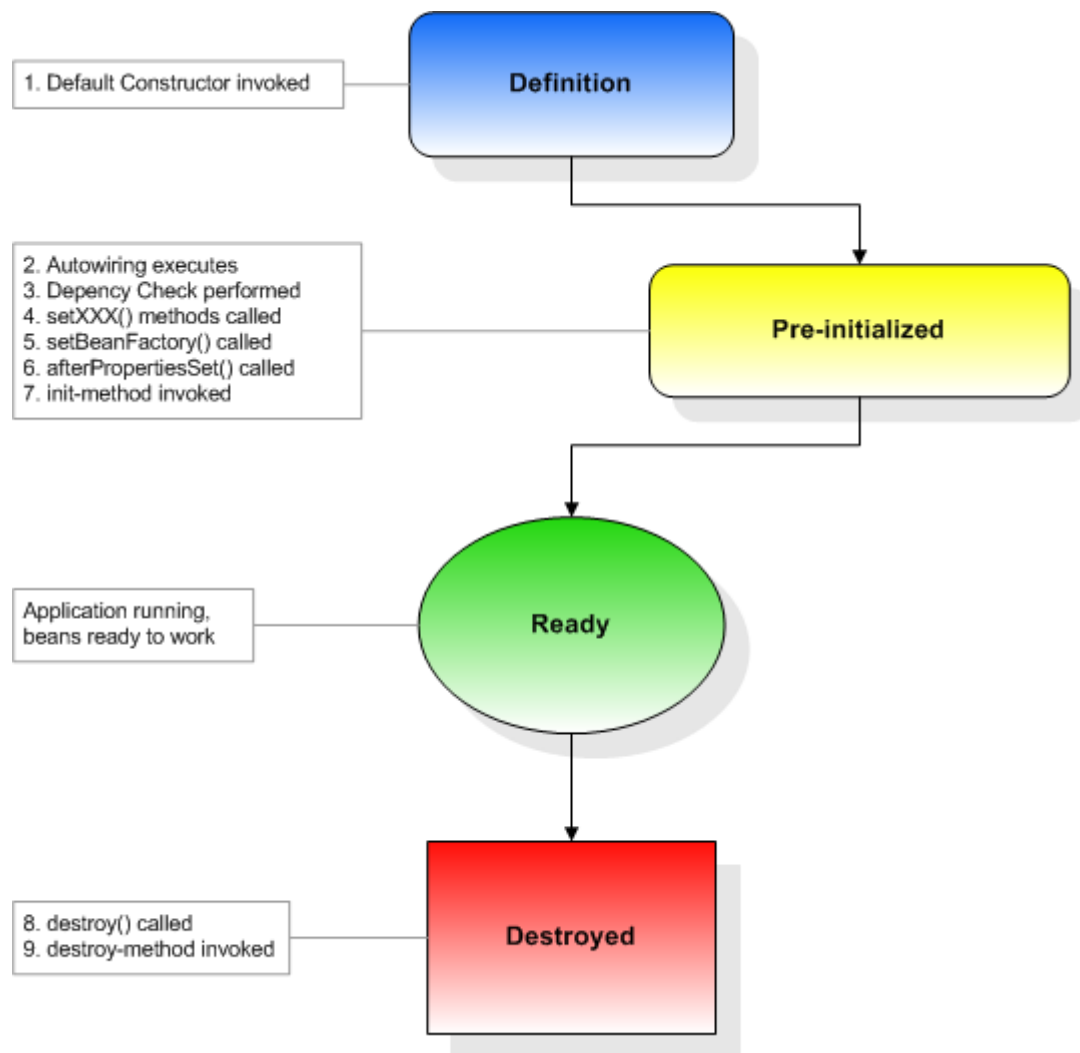


Figure 9.1: A Bean's Lifecycle in the BeanFactory

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	This attribute controls the <i>autowiring</i> of bean properties. If used, Spring automatically figures out the dependencies using one of the following modes: no: (Default) You must define bean references in the XML file using the <code><ref></code> element. Recommended to make documentation more explicit. 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. byType: Autowires if exactly one bean of the property type is in the factory. constructor: same as <i>byType</i> for constructor arguments. autodetect: chooses the <i>constructor</i> or <i>byType</i> through inspection of the bean class. 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.	Low

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

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

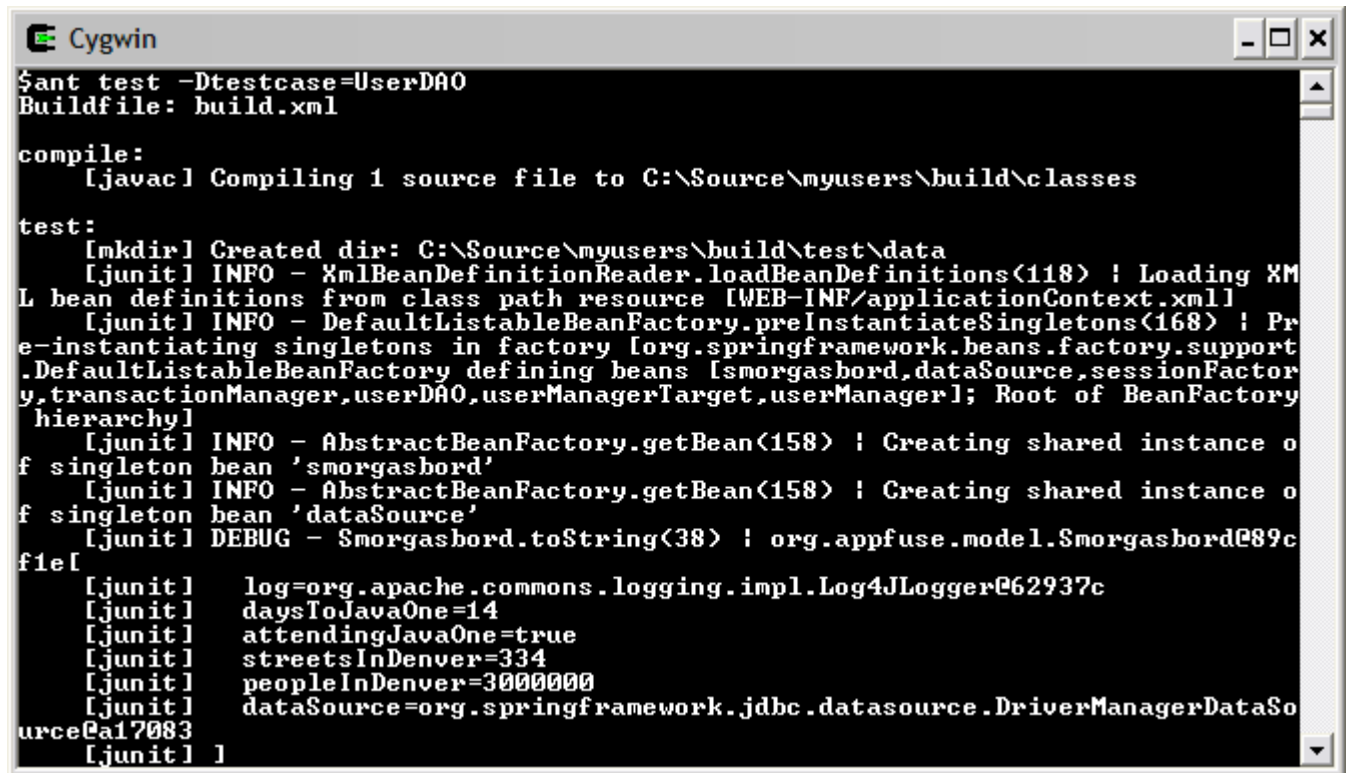
For your professional applications, I strongly recommend always writing a Unit Test.

```
<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] 1

```

Figure 9.1: 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(BeansFactory 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"/>
    </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 loading 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: Testing Spring Applications*.

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 [HierarchicalMessageSource](#), 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 [StaticMessageSource](#) (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.ResourceBundleMessageSource">
  <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
ContextRefreshedEvent	Event published when the ApplicationContext is initialized or refreshed. Initialized here means that all beans are loaded, singletons are pre-instantiated and the ApplicationContext is ready for use.
ContextClosedEvent	Event published when the ApplicationContext is closed, using the <code>close()</code> method on the ApplicationContext. Closed here means that singletons are destroyed.
RequestHandledEvent	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 was 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

Spring MVC: 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.

Chapter 3 explored Spring's BeanFactory and its lifecycle, which you can use to control how your beans are invoked and used. Spring carries this concept into the web tier and has some pretty neat concepts in its MVC Framework. In popular frameworks like Struts and WebWork, controllers usually contain a single method: `execute()`. The framework, regardless of whether a GET or POST request is sent, will call 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. Of course, 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) is 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 is designed to handle 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 that is 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 that are invoked while processing a data-entry form. is the reason one is an interface and the other super-class is primarily for flexibility. The `Controller` interface is used by all Controllers 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`, certain methods are called on GET requests, while others are called on POST requests. 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). We 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 Section in the Method Lifecycle Review towards the end.

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

- Unit Testing Spring Controllers
- Configure [DispatcherServlet](#) and [ContextLoaderListener](#)
- Create Unit Test for `UserController` (used to display a list of users)
- Create `UserController` and configure it in `action-servlet.xml`
- Create `userList.jsp` to display list of users
- Create Unit Test for `UserFormController` (edits, saves and deletes users)
- Create `UserFormController` and configure it in `action-servlet.xml`
- Create `userForm.jsp` to allow editing user's information
- Configure Commons Validator for Spring
- `SimpleFormController` - Method Lifecycle Review
- Spring's JSP Tags

As previously mentioned, Spring's MVC framework is a bit different than traditional frameworks like Struts and WebWork. With Spring, I tend to use two controllers for my master/detail screens. With Struts, I tend to use one Action for deleting, editing, saving and listing rows in a database table. By "listing," I mean the process of getting all the rows from a particular table. This satisfies most of what I tend to do in web applications. With Spring MVC, 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).



NOTE

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

Chapter 11 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 I had was that none of the recommended solutions (such as [Mock Objects](#)) had APIs to handle the stuff you normally do in a webapp: setting request parameters/attributes, grabbing stuff from application scope, etc. With Struts, you can 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.

In this chapter, like previous ones, you can follow along and do the examples as you go. The easiest way to do this is to download the **MyUsers Chapter 4** bundle from <http://sourcebeat.com/downloads>. This download is similar to the Equinox package you downloaded for *Chapter 2*. However, it has all Struts-related components removed and is designed to be a pure Spring (with Hibernate) application. It contains all the JARs you will need in this chapter in its *web/WEB-INF/lib* directory.

You can also use the application you developed in *Chapter 2*. If you go this route, download **Chapter 4 JARs** from <http://sourcebeat.com/downloads>. The next section discusses how Spring is configured in the downloaded bundle. It also shows what you need to modify if you're converting the Struts-based application from the last chapter.

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

Configure DispatcherServlet and ContextLoaderListener

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 as well, 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 than handles all requests, similar to most Java web frameworks². It's called the [DispatcherServlet](#) and is responsible for "dispatching" request to handlers, which have mappings to tell it where to go next. In the *Chapter 4* download, the DispatcherServlet is already configured in the `web/WEB-INF/web.xml` file. Its mapping is set to `*.html` - which means that any URLs ending in `.html` will be handled by this servlet.

If you're modifying the application created in *Chapter 2*, you'll need configure the MyUsers application to use the DispatcherServlet for its front controller, rather than Struts' [ActionServlet](#). Instructions for doing this are in the section below.

Modify web.xml to Use Spring's DispatchServlet

At this point, you should have the "myusers" project setup on your hard drive. To begin, open `web/WEB-INF/web.xml` and modify the "action" servlet's `<servlet-class>` from:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

to:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

2. This is a Core J2EE Pattern called [Front Controller](#).

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>
```

By default, the `DispatcherServlet` looks for an XML file named *servlet-name-servlet.xml* in the *WEB-INF* directory. In this case, it'll find and load the *action-servlet.xml* once you create it. This file contains all of the web controllers and settings used in *MyUsers*.

In *Chapter 2*, you used the Spring Plugin for Struts ([ContextLoaderPlugin](#)) to load the bean configuration files. However, the `ContextLoaderListener` was also configured in your *web.xml*. This caused the *applicationContext.xml* file to be loaded twice. This was so you could unit test your Action classes without loading any context files manually.



NOTE

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

Since the `ContextLoaderListener` is already configured in *web.xml*, no further configuration is needed on your part. If you have more than one file with bean definitions, you must add a `contextConfigLocation` context parameter to indicate the different files. For example, to do this in *MyUsers*, you would 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/applicationContext1.xml
    /WEB-INF/applicationContext2.xml
  </param-value>
</context-param>
```



NOTE

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

Those are the basic steps to configure a Java webapp to use Spring's MVC Framework. Here's a review of the steps:

1. In *web.xml*, add a `<servlet>` definition for `DispatcherServlet` and configure its `<servlet-mapping>`.
2. If you have more than one context file, define a `contextConfigLocation` `<context-param>` with the paths to your bean configuration files.
3. Add a `<listener>` definition for `ContextLoaderListener`.

Remove Struts and add Spring Files

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
```

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>
```

Download [Chapter 4 JARs](#) to your hard drive. Put the JAR files in the *web/WEB-INF/lib* directory. The *spring.jar* file contains Spring 1.1.1 (Equinox 1.0 ships with Spring 1.0.2), the *spring-mock.jar* file contains mocks for the Servlet API, and the *spring-sandbox.jar* file is used for its Commons Validator support. Another file, *validation-rule.xml* is also contained in the download - put this file in the *web/WEB-INF* directory. Configuring validation will be covered later in this chapter.

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 class returns a list of all the users from a business service class (`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.”

To create a JUnit Test for the Controller, create a `UserControllerTest.java` file in `test/org/appfuse/web` (you might need to create this directory/package). 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 be instantiated. By web-only beans, I mean those that require a `WebApplicationContext` to be present.

```
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. 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 extend. In this class's `setUp()` method, you can load the context for all child tests. If you do this, make sure to call `super.setUp()` in child classes' `setUp()` methods.

Stepping through the `testGetUsers()` method, you're grabbing the `UserController` and invoking its `handleRequest()` method – which returns a `ModelAndView` class. This method is common to all Spring Controllers, so you'll actually use this same method to test your FormControllers. The `ModelAndView` class is a unique concept in web frameworks. It basically 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 get it to compile. First, create a `UserController.java` file in `src/org/appfuse/web` (you may need to create this directory/package). This class should implement the `Controller` interface 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 you'll need to add a private `UserManager` variable and a `setUserManager()` method for Spring's IoC container to use. When you configure this Controller (a.k.a. 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 = LogFactory.getLog(UserController.class);
    private UserManager mgr = null;

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

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

Now 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 (using `ant test -Dtestcase=UserController`) it will fail.

```
[junit] Testcase: testGetUsers(org.appfuse.web.UserControllerTest): Caused
an ERROR
[junit] Line 7 in XML document from resource [/WEB-INF/action-servlet.xml]
of ServletContext is invalid; nested exception is
org.xml.sax.SAXParseException: Element "beans" requires additional
elements.
```

This error is saying that the `web/WEB-INF/action-servlet.xml` does not validate. This is because it has no beans defined in it. To make the test pass, 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.

Adding a "userController" bean definition should cause this file to look as follows:

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

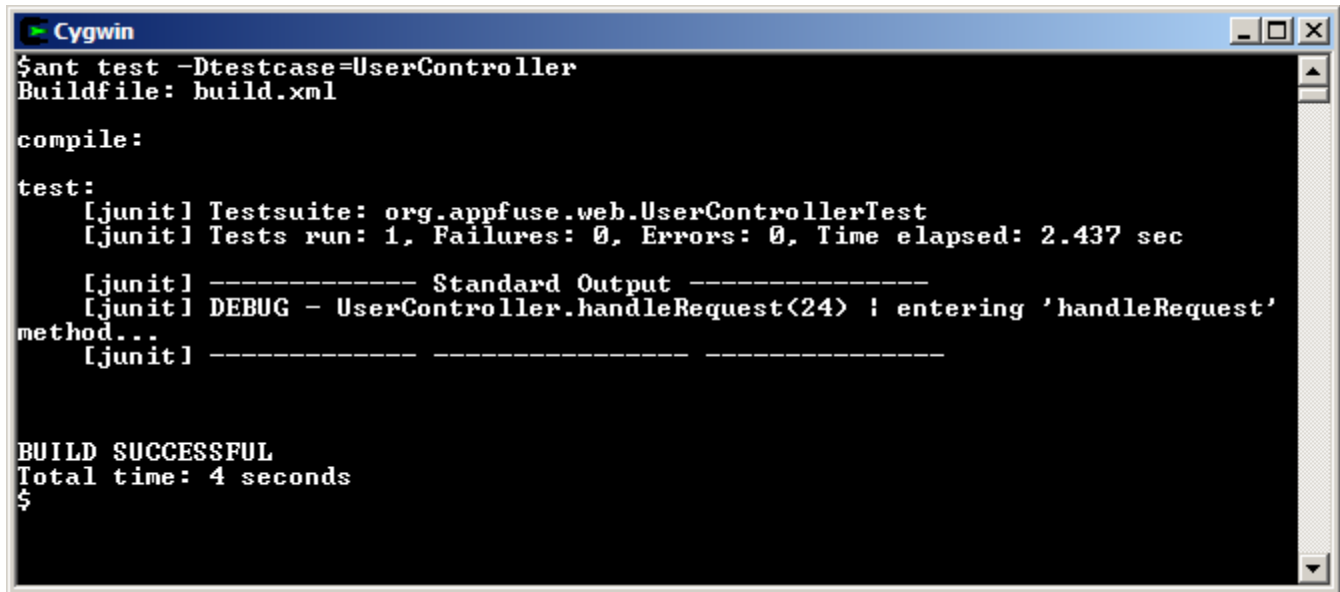
<beans>
    <bean id="userController" class="org.appfuse.web.UserController">
        <property name="userManager">
            <ref bean="userManager"/>
        </property>
    </bean>
</beans>
```

Now your `UserControllerTest` should run just fine. You can execute it by running `ant test -Dtestcase=UserController` or run it as a JUnit Test in Eclipse or IDEA.

**TIP**

Instructions for setting up MyUsers in Eclipse and IDEA are available in [this book's FAQ](#).

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` test

Create userList.jsp to Display List of Users

Now that the `UserController` is working, you must 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. Add the following XML block to *action-servlet.xml*, after the “`UserController`” 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 “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*.

Now configure URLs in the application so that the “/users.html” URL invokes the `UserController` class. To do this, Spring MVC requires you to define a [HandlerMapping](#) bean and define which URLs go to which controllers. In most cases, the [SimpleUrlHandlerMapping](#) is all you’ll need. It allows you to specify url-patterns to bean names. In order 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.Simple
        UrlHandlerMapping">
  <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.

If you’re modifying the application you created in *Chapter 2*, modify a couple of JSPs. The *web/taglibs.jsp* file should contain the following:

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

And the *web/messages.jsp* file should have the code below:

```
<%-- Success Messages --%>
<c:if test="${not empty message}">
  <div class="message">${message}</div>
  <c:remove var="message"/>
</c:if>
```

1. Create a *userList.jsp* file in the *web* directory. This file may already exist.
2. Add 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 JSTL and SiteMesh (which is used to “pretty up” the JSPs).

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

<title>MyUsers ~ User List</title>

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

<table class="list">
<thead>
<tr>
    <th><fmt:message key="user.id"/></th>
    <th><fmt:message key="user.firstName"/></th>
    <th><fmt:message key="user.lastName"/></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="editUser.html?id=${user.id}">${user.id}</a></td>
    <td>${user.firstName}</td>
    <td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```

3. To enable i18n message lookups (for the `<fmt:message>` tag), 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>
```

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, you can use the “basenames” property with a `<list>` of `<values>`.

**NOTE**

The `ResourceBundleMessageSource` depends on Java's `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`.

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**.

**NOTE**

If running `ant populate` doesn't cause users to show up in the list, see [this FAQ](#).

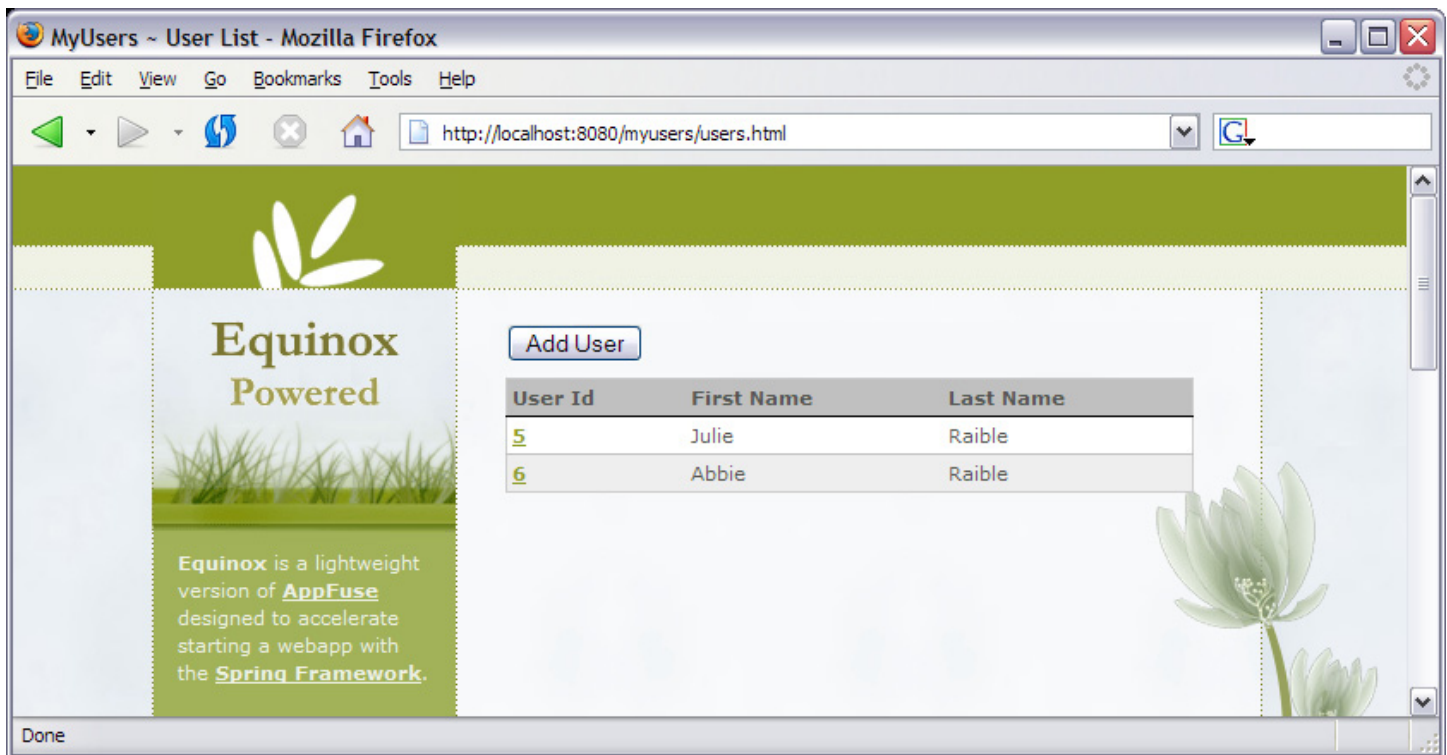


Figure 4.2: Results of the `ant populate` test

If your screen looks like the one above, congratulations! If not, send e-mail to the Equinox users' mailing list (users@equinox.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). To do this, create a `UserFormControllerTest.java` file in the `test/org/appfuse/web` directory. This file extends JUnit's `TestCase` class and has 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
}
```

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.

Now 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 MockHttpServletRequestResponse());
    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 MockHttpServletRequestResponse());
    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 MockHttpServletRequestResponse());
    assertNotNull(request.getSession().getAttribute("message"));
}
```

In the previous methods, you should probably 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](#) interface. 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` `java.lang.Long` conversion for `user.setId()`.

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 should extend *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. The `onSubmit()` method returns a `ModelAndView`, but wraps the `successView` with a `RedirectView`. This is to fix the "[double submit](#)" problem that often exists in web applications.

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*.

Now 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*.

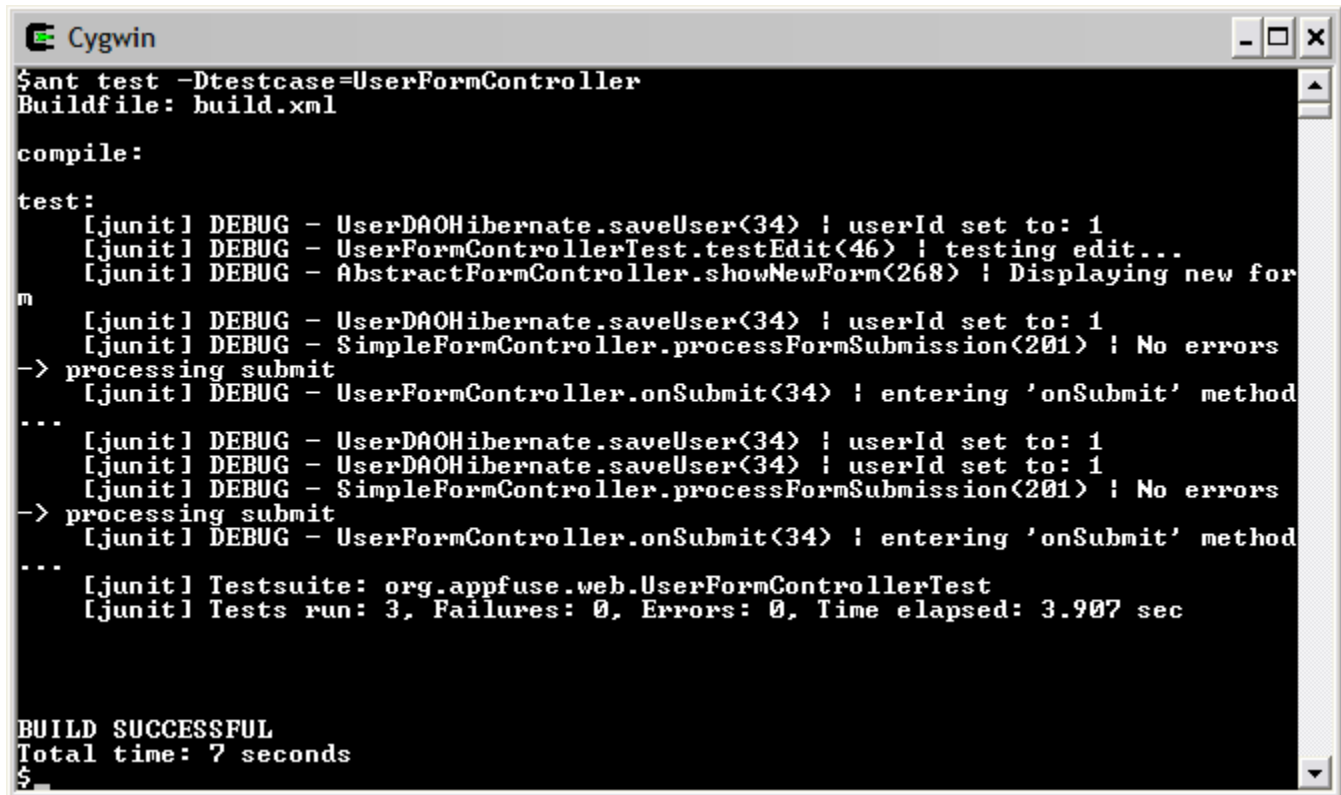
```
<bean id="userFormController"
      class="org.appfuse.web.UserFormController">
  <property name="commandName"><value>user</value></property>
  <property name="commandClass">
    <value>org.appfuse.model.User</value>
  </property>
  <property name="formView"><value>userForm</value></property>
  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

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
m
[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 9.1: Results of the `ant test -Dtestcase=UserForm` test

If you see something similar to the output above, *nice work!* Next create the JSP to interact with the `UserFormController`.

Create userForm.jsp to Allow Editing User's Information

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. At the time of this writing, an enhancement request is in Spring's JIRA to add [JSP 2.0 tag files](#) for easier form syntax.

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.

Struts will prepend the `contextPath`, and append 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. Let's 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.

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

Struts' [ActionForm](#) and Spring's 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.

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>
        </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"/>
```

The last thing you need to do is add validation.

Configure Commons Validator for Spring

At the time of this writing, Spring 1.1.1 has been released, and it does not contain a built-in declarative validation framework. The core validation framework requires that you 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. 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 (removing 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>
```

The download for this chapter has a *validation-rules.xml* file (in *web/WEB-INF*) that is different from the one that ships with Struts. This file defines all of the Spring-specific classes and methods to use for validation, as well as defining JavaScript methods for client-side validation.

At the bottom of *userForm.jsp* is a JSP tag that renders the JavaScript for client-side validation.

```
<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.

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 Plain ol' Java Object (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.

Now if you run `ant deploy reload`, 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:

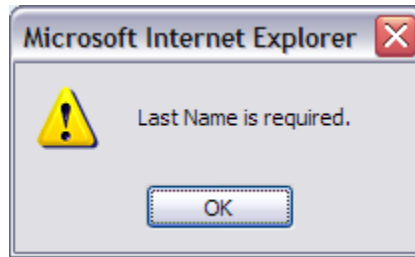


Figure 9.1: Results of the `ant deploy reload` test with JavaScript disabled

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:

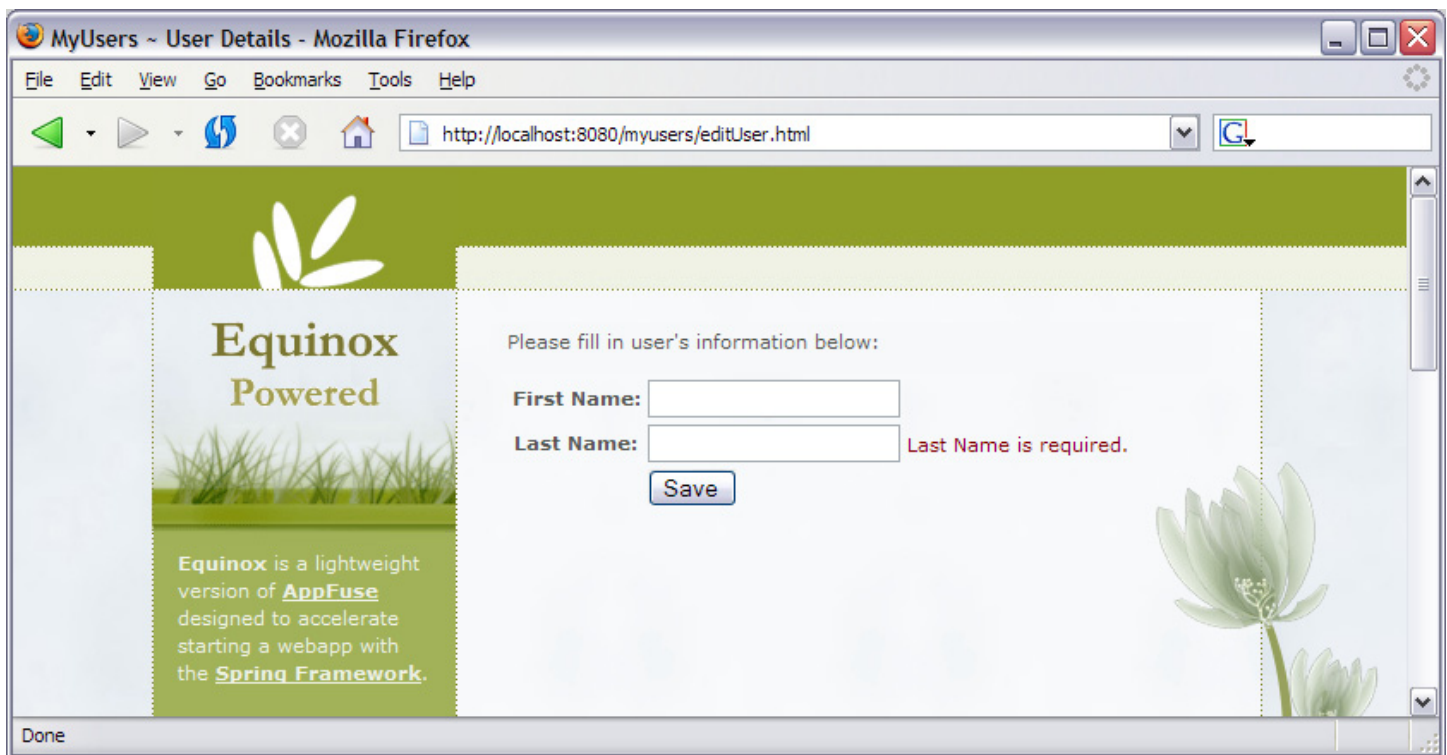


Figure 9.1: Results of the `ant deploy reload` test with JavaScript enabled

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>
```

If you add this code, run **ant deploy** and click “Save” without adding a lastName (with JavaScript turned off). Your screen should resemble the following screenshot:

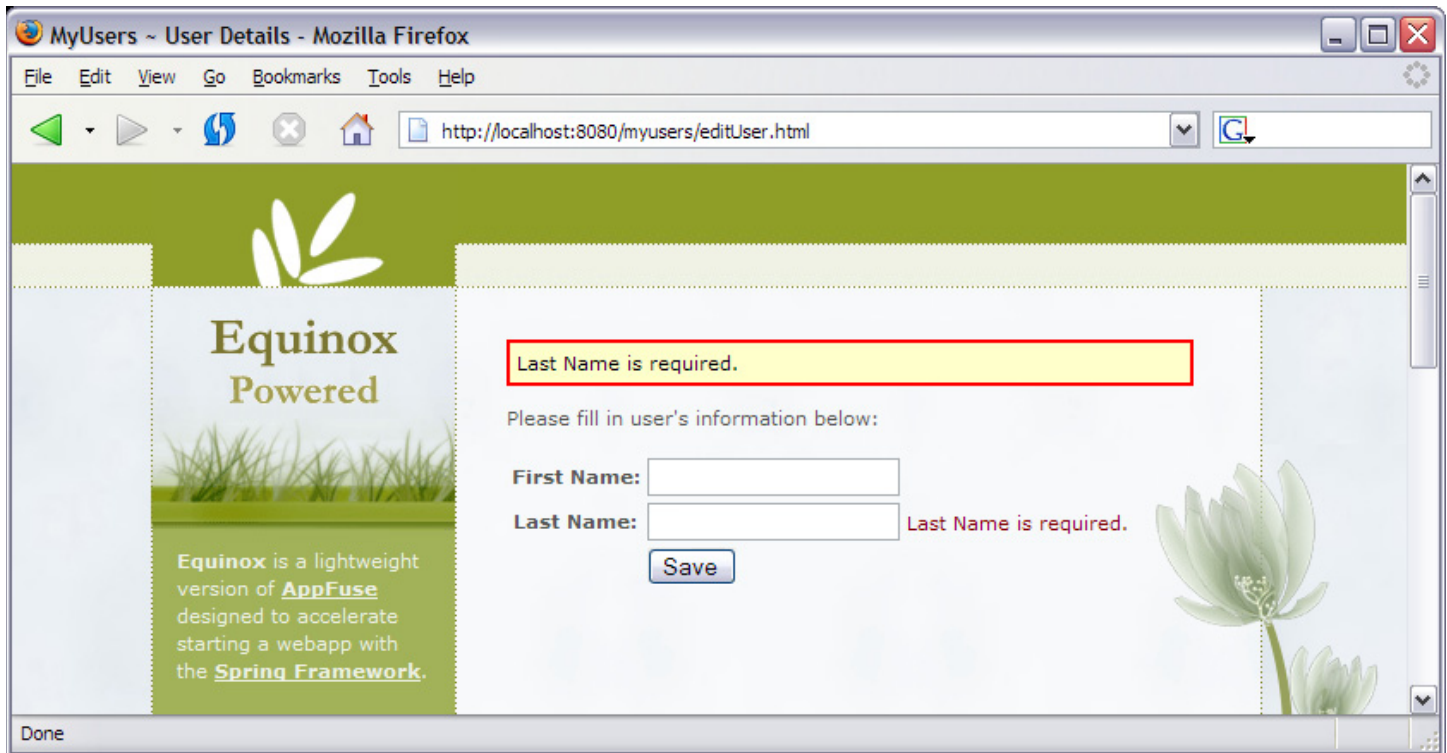


Figure 9.1: Results of the `ant deploy` test with JavaScript disabled

You’ve just created a web application that uses 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 shows a `POST` request's lifecycle.

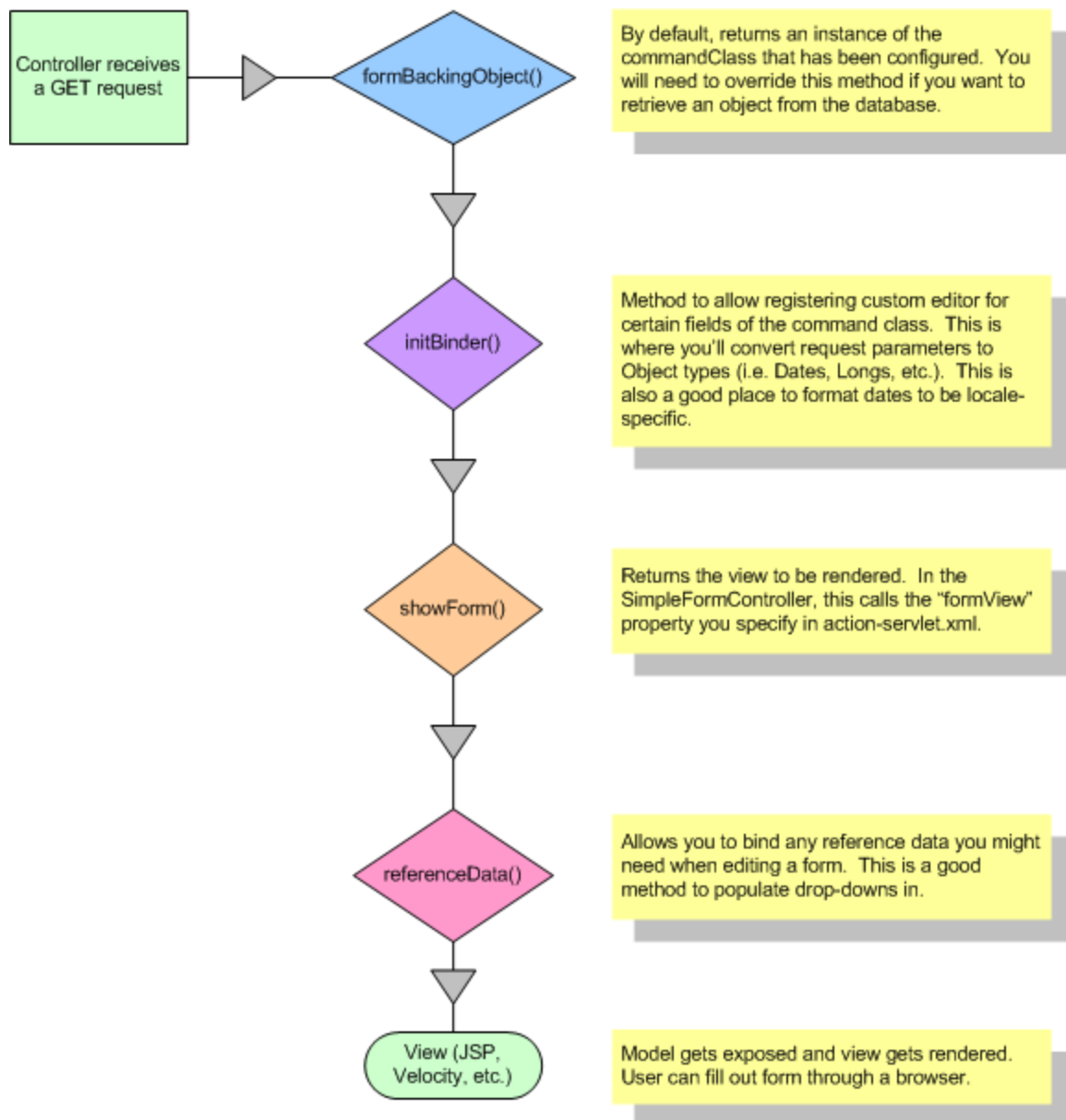


Figure 9.1: GET request lifecycle

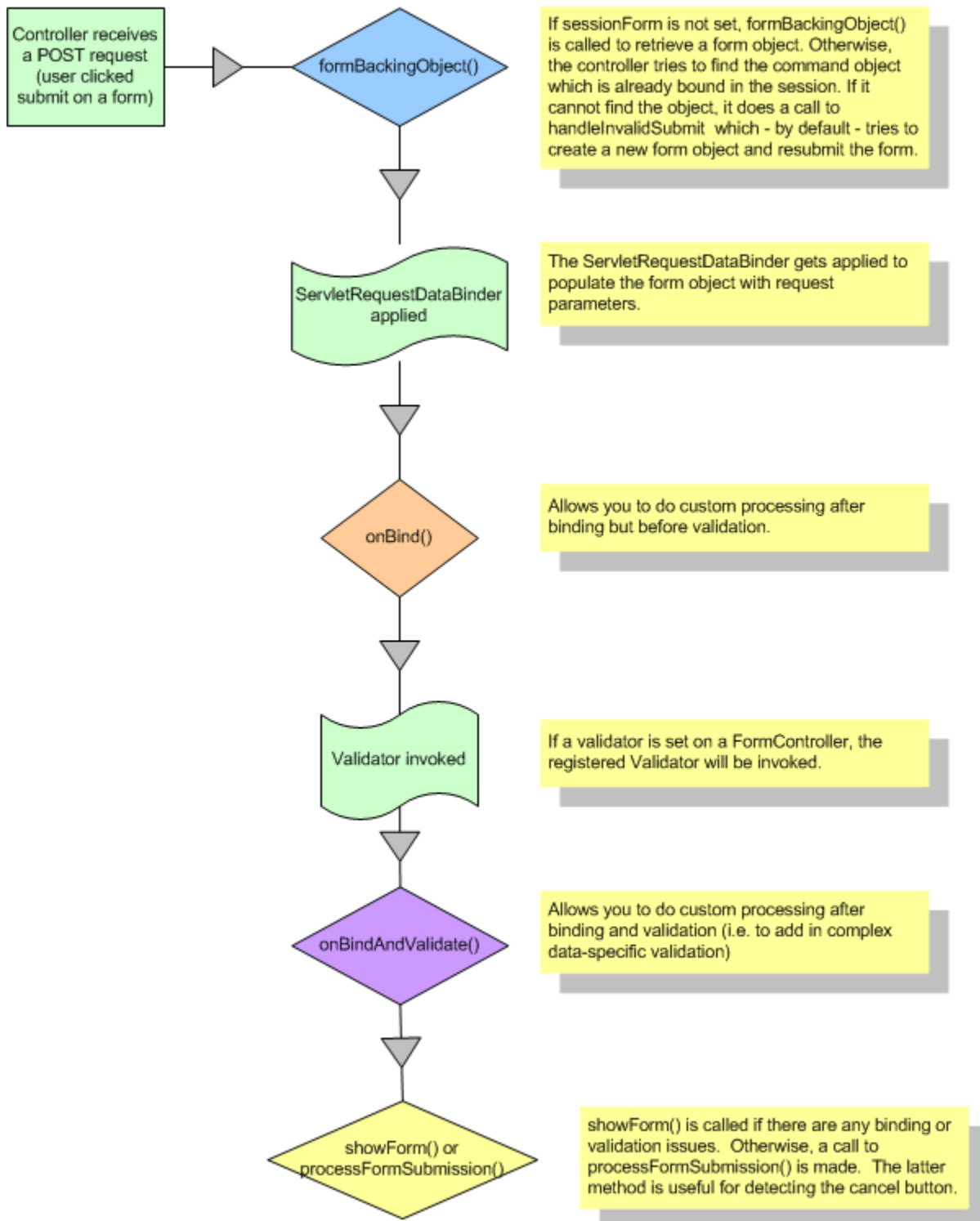


Figure 9.1: POST request lifecycle

These diagrams should give you a better understanding of the Spring's FormController's lifecycle. Knowing these methods and when they're called can be helpful when developing apps with Spring MVC.

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 if it has 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. Its 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 *page decoration* (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.

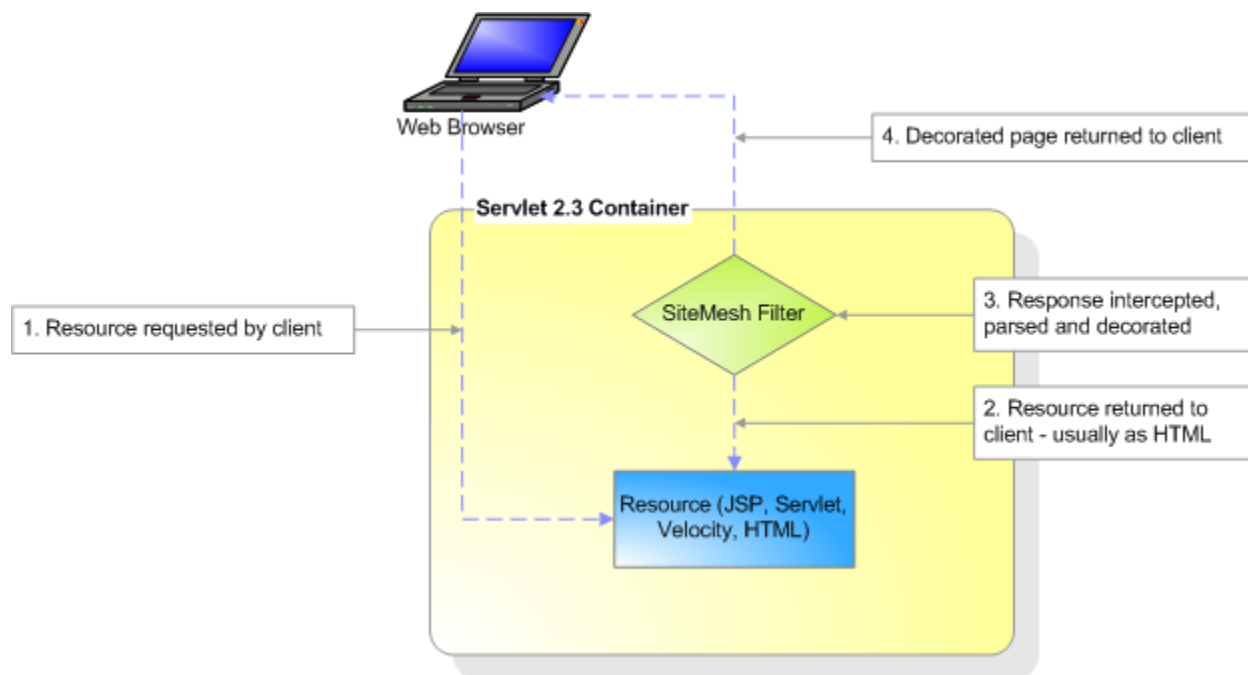


Figure 9.1: The SiteMesh process

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

If you're using the application you developed in *Chapter 4*, SiteMesh is already configured. In order to start from scratch (without SiteMesh) you must download the **MyUsers Chapter 5** bundle from <http://sourcebeat.com/downloads>. 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** while Tomcat is running; 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. Installing and configuring SiteMesh will allow you to pretty it up a bit.

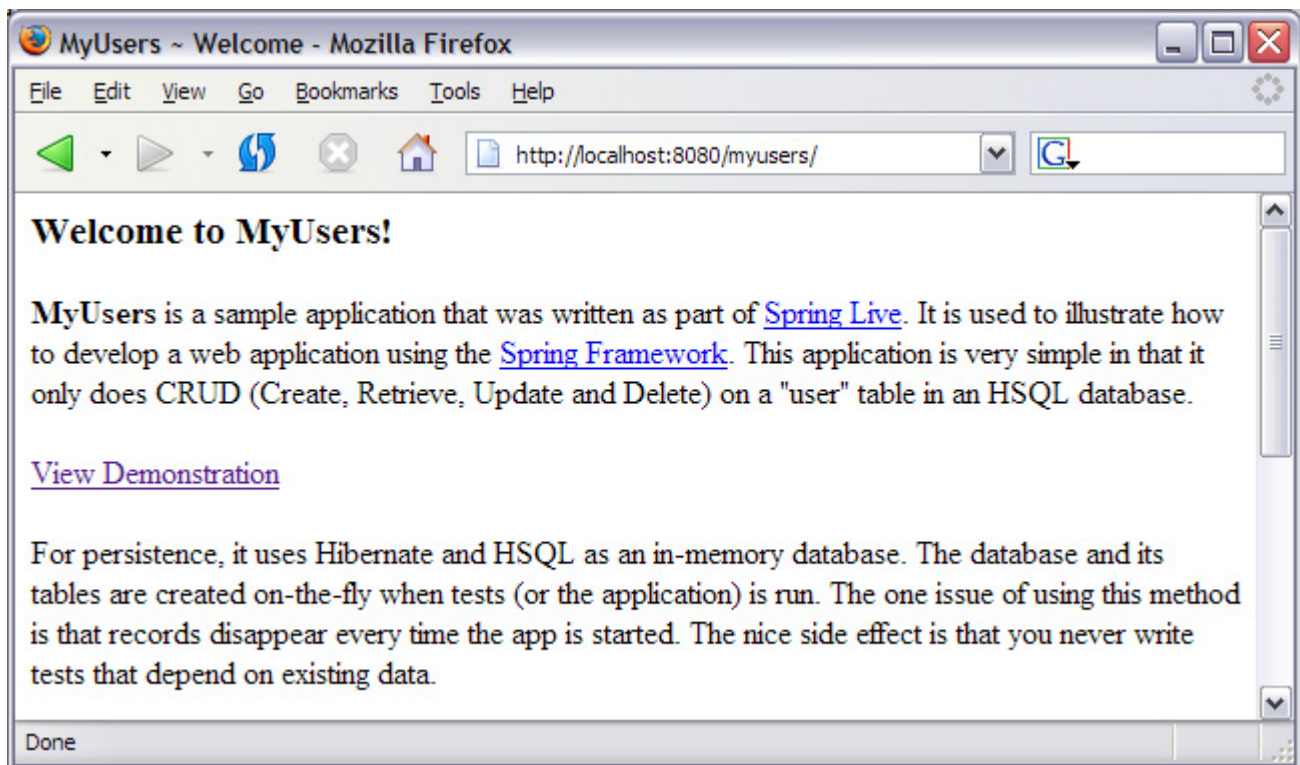


Figure 9.1: "Welcome to MyUsers" without decoration

Step 1: Configure SiteMesh in web.xml

The *sitemesh-2.1.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>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

The `<dispatcher>` elements in the `<filter-mapping>` above are new to Servlet 2.4. These dispatchers (as well as `ERROR`) allow map filters to more than just requested URLs.

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.FastPageParser"/>
    <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>
</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/urlrewritel.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;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "lastName", "errors.required", "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*. This file is included in the downloadable bundle for this chapter.
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:

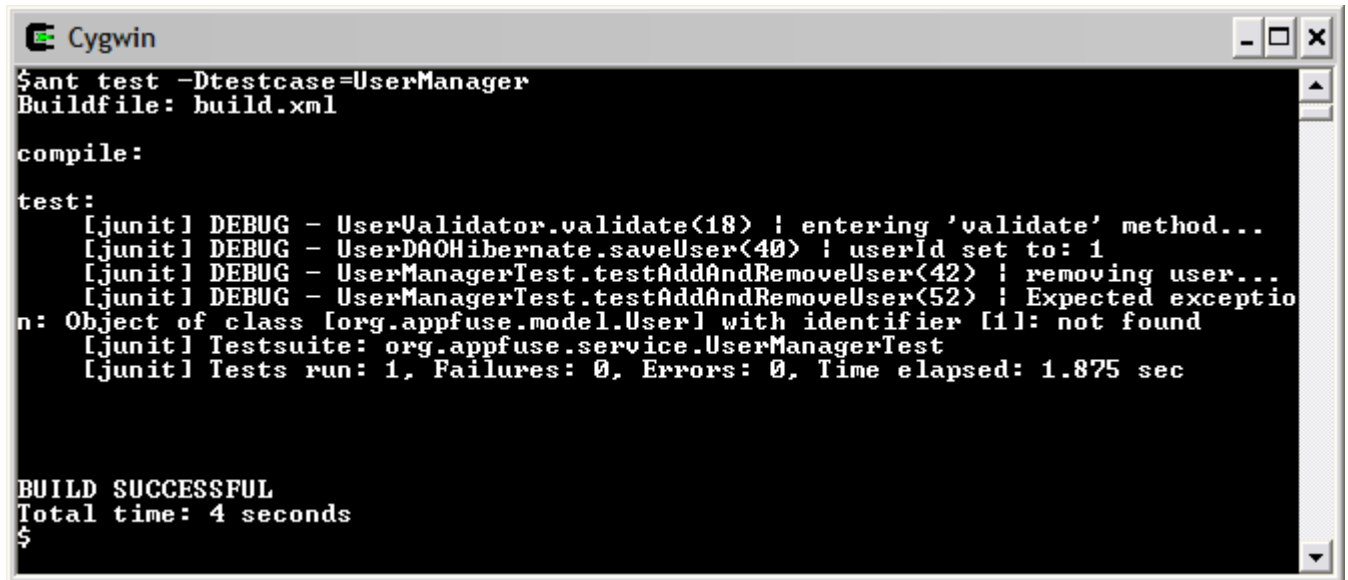
```
public User saveUser(User user) {
    BindException errors = new BindException(user, "user");
    new UserValidator().validate(user, errors);
    if (errors.hasErrors()) {
        throw new RuntimeException("validation failed!", errors);
    }
    dao.saveUser(user);
    return user;
}
```

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:
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 9.1: 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 `onSubmit()` method in a FormController. In frameworks like Struts, you can do this on your Actions and then declaratively map Exceptions to views. In other words, you can specify (in XML) that when an Exception gets thrown, the user should be forwarded to an Action, a Tiles’ definition, or a JSP.

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());
    assertNotNull(dae);
}
```

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.

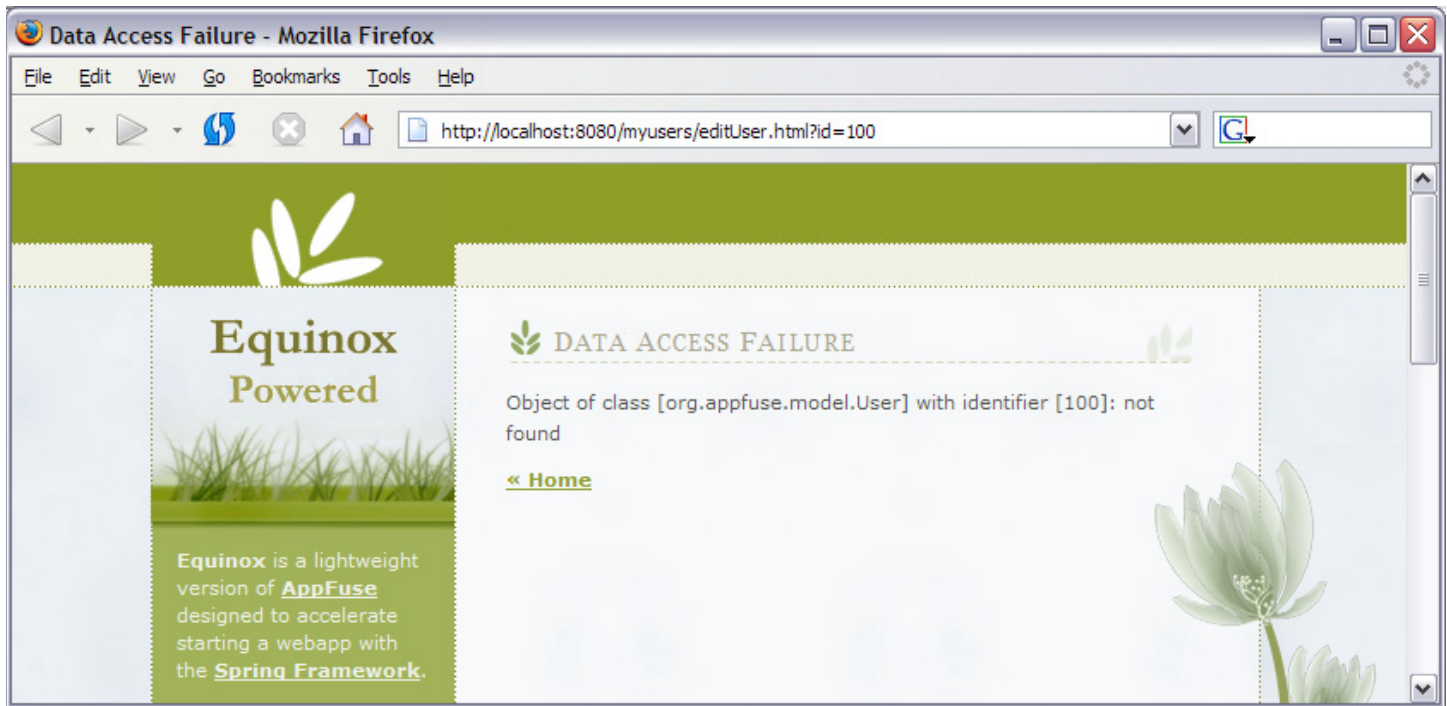


Figure 5.5: Error page stating that the user does not exist

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"/>
```

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.

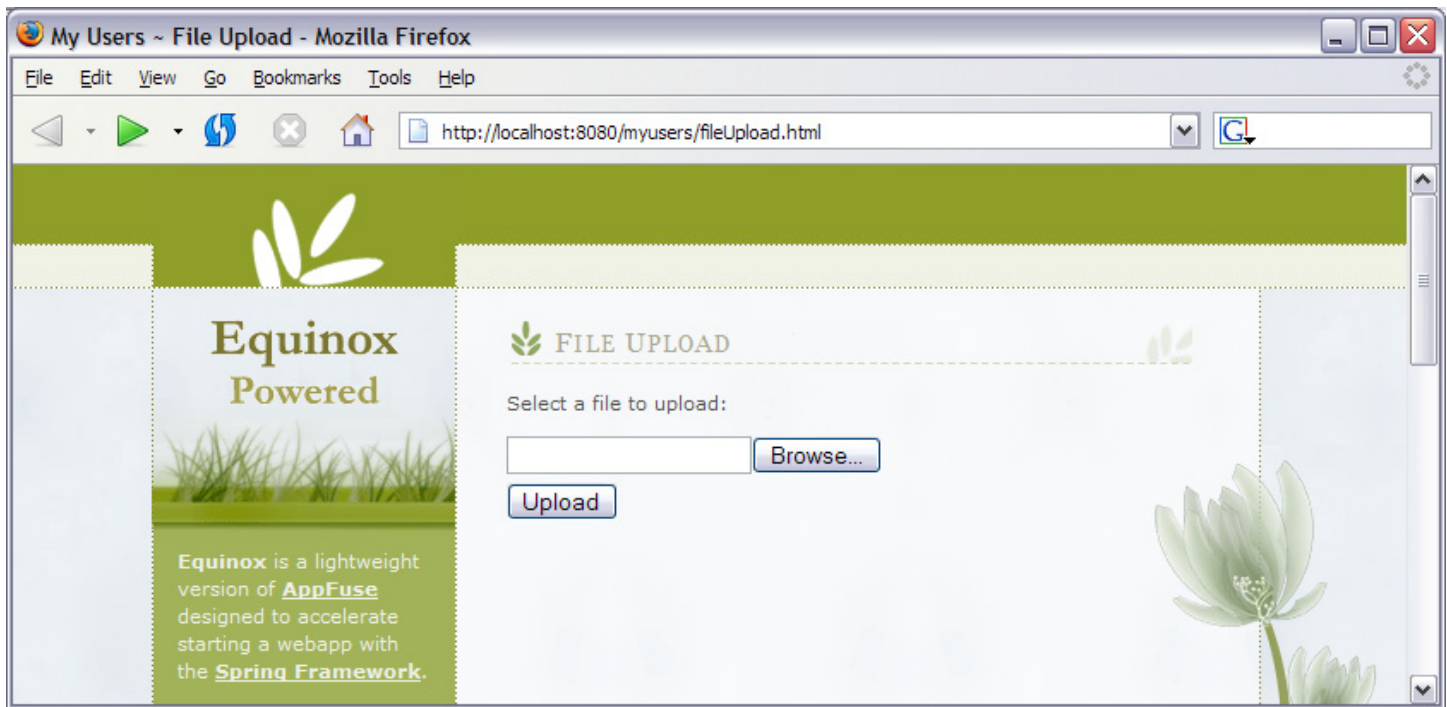


Figure 5.6: "File Upload" page

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.

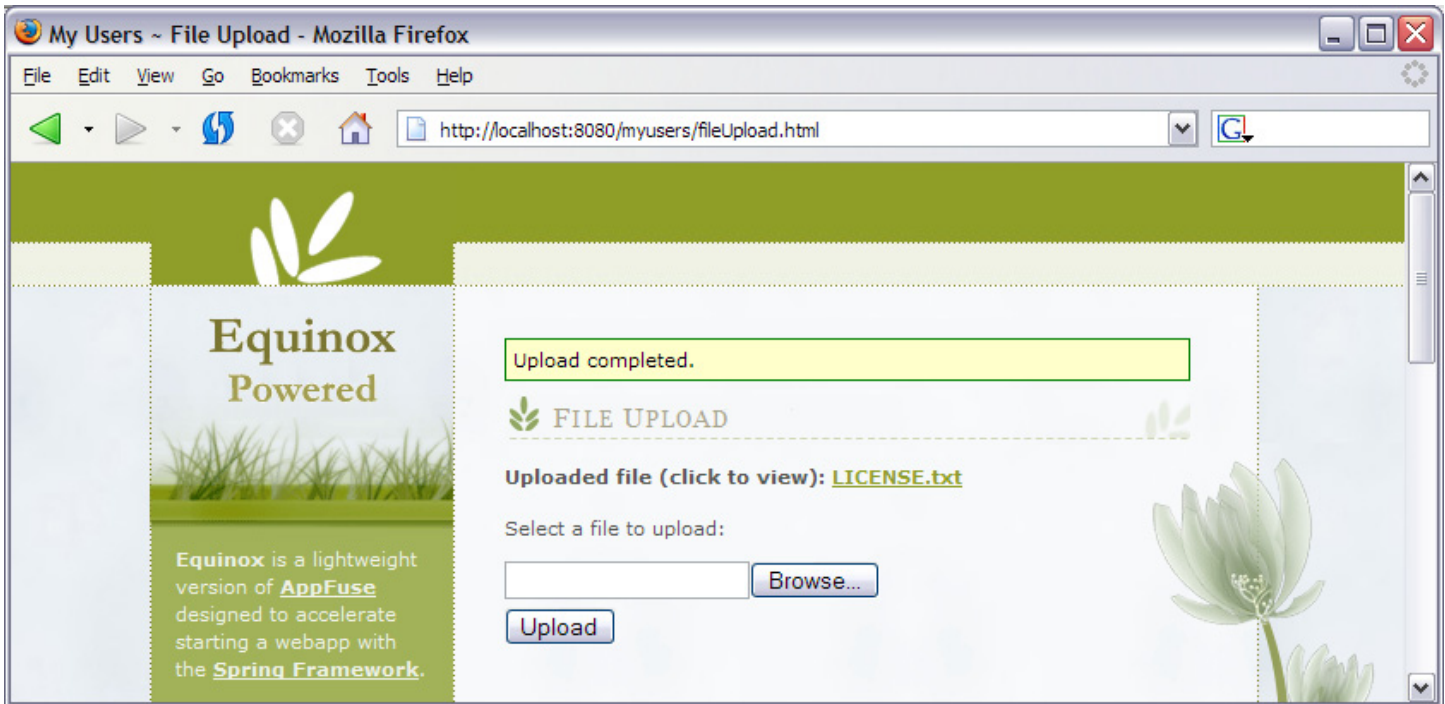


Figure 5.7: A successful file upload

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 Servlet Filters, 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

Using the [JavaMailSenderImpl](#) class requires that you have `activation.jar` and `mail.jar` in your classpath. These files are included in the *Chapter 5* download bundle.

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`.

- 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.

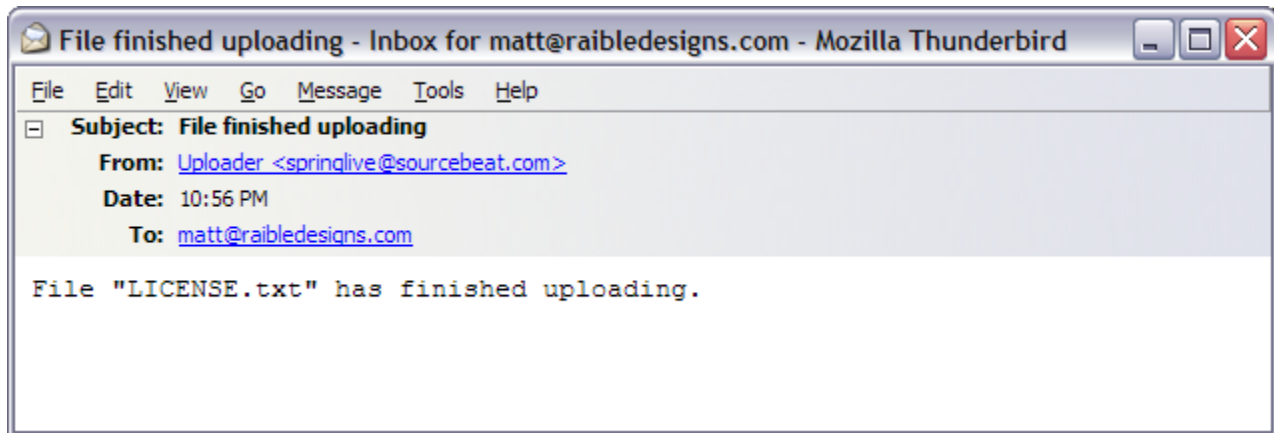


Figure 5.8: Successful e-mail message



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 a 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, Excel 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!

View Options

The Different View Options in Spring

This chapter covers the view options in Spring's MVC architecture. At the time of this writing, the options are JSP, Velocity, FreeMarker, XSLT, PDF and Excel. This chapter aims to become a reference for configuring all Spring-supported views. It also contains a brief overview how each view works and compares constructing a page in MyUsers with each option. Additionally, it focuses on internationalization for each view option.

In J2EE applications, JSPs have become the *de facto* standard for constructing the V in MVC because they're really the only page-templating option provided by the J2EE (1.4) specification. However, other options are quickly gaining recognition. Velocity and FreeMarker are templating technologies that use syntax similar to JSP 2.0. XML/XSL is a nice option if you are pulling XML data and want to transform it, or if you want to serve up different views (via XSL) on-the-fly. XSLC is similar to the templating technologies, except its syntax is plain HTML and it uses the `id` attribute to locate and replace dynamic text. Lastly, outputting data as PDF and Excel formats is great for reporting and producing portable documents. In this chapter, you will learn how to use each of these technologies with Spring's MVC framework. Before learning how to configure the different views, it's important to understand how Spring determines views.

Views and ViewResolvers

Spring ships with a number of *ViewResolvers*, which allow you to de-couple your Controllers from your View. In your Controller, you simply specify a logical name for a view and Spring resolves that name to a specific view type. The `View` interface prepares the request and hands it over to whichever view technology you have configured. In its loose-coupling spirit, Spring makes it easy to configure your view choice in one of your XML context files.

In *Chapter 4*, you saw how Controllers return a `ModelAndView`. This object contains the name of a view that the specified `ViewResolver` uses to determine what to show to the user. Throughout this chapter, you will learn how each `ViewResolver` works. Below is a list of the `ViewResolvers` currently available in Spring. The “Useful For” column indicates how one might use each resolver.

Table 6.1: View Resolvers

View Resolver	Description	Useful For
<code>AbstractCachingViewResolver</code>	Abstract <code>ViewResolver</code> that caches views. Lots of views need preparation before you can use them; extending from this view resolver enables caching of views.	Extending to implement caching in a custom resolver.
<code>ResourceBundleViewResolver</code>	Implementation of <code>ViewResolver</code> that uses bean definitions from a <code>ResourceBundle</code> , specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath.	Mixing different Views, such as JSP and Velocity.
<code>UrlBasedViewResolver</code>	Simple implementation of <code>ViewResolver</code> that allows for direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.	Resolving symbolic view names to URLs, without explicit mapping definition.
<code>InternalResourceViewResolver</code>	Convenient subclass of <code>UrlBasedViewResolver</code> that supports <code>InternalResourceView</code> (such as Servlets and JSPs), and subclasses like <code>JstlView</code> and <code>TilesView</code> . You can specify the view class for all views generated by this resolver via <code>setViewClass</code> .	Supporting JSPs and Tiles. <code>JstlView</code> and <code>TilesJstlView</code> are the most convenient viewClasses.

Table 6.1: View Resolvers

VelocityViewResolver	Convenient subclass of AbstractTemplateViewResolver that supports VelocityView (that is, Velocity templates) and custom subclasses.	Supporting Velocity Views. Allows caching and custom properties.
FreeMarkerViewResolver	Convenient subclass of AbstractTemplateViewResolver that supports FreeMarkerView (that is, FreeMarker templates) and custom subclasses of it.	Supporting FreeMarker Views.

In most cases, you won't need to choose which ViewResolver to use; you will simply use the one that your view technology requires. The exception is the `ResourceBundleViewResolver`, which allows you to choose different view classes for each view name. You will learn how to configure and use each Spring-supported view technology before using this resolver.

Each view technology section will show you how to configure the view resolver and how to implement pages for that technology. The best way to learn is to follow along with the exercises in this chapter. To do this, download the **MyUsers Chapter 6** bundle from <http://sourcebeat.com/downloads>. This project tree contains all the JARs you will use in this chapter. You can also use the application you've been developing in previous chapters. If you go this route, download **Chapter 6 JARs** from <http://sourcebeat.com/downloads>.

**TIP**

If you choose to use your existing work, you might want to remove the security you added to demonstrate *Interceptors*. It's a pain to login each time you test the view.

Before getting into view options, write a unit test to verify the basic functionality of listing, adding, saving and deleting a user.

Testing the View with jWebUnit

[jWebUnit](#) is an open-source project hosted on SourceForge. It offers a simple API (on top of HttpUnit) for testing web applications.

1. If you downloaded the *Chapter 6* bundle (or the *Chapter 6* JARs), you don't need to add any additional JARs to your application. Otherwise, download jWebUnit (version 1.2) and put the following JARs in *web/WEB-INF/lib*:
 - jwebunit-1.2.jar
 - httpunit-1.5.4.jar
 - Tidy.jar
2. If you're developing with Eclipse or IDEA, modify your project's classpath to include these JARs. For Eclipse 3.0, this is at Project → Properties → Project Build Path → Libraries Tab. In IDEA 4.5, this is at File → Settings → Project Settings:Paths → Libraries Tab.



TIP

Instructions are available for building and testing the MyUsers application in [Eclipse](#) and [IDEA](#).

3. Create a test by creating a new *UserWebTest.java* class in *test/org/appfuse/web* that extends *WebTestCase*. The code for this class is listed below:

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserWebTest extends WebTestCase {

    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
    }

    public void testWelcomePage() {
        beginAt("/");
        assertEquals("MyUsers ~ Welcome");
    }

    public void testAddUser() {
        beginAt("/editUser.html");
        assertEquals("MyUsers ~ User Details");
        setFormElement("id", "");
    }
}
```

```

        setFormElement("firstName", "Spring");
        setFormElement("lastName", "User");
        submit("save");
        assertTextPresent("saved successfully");
    }

    public void testListUsers() {
        beginAt("/users.html");

        // check that table is present
        assertTablePresent("userList");

        //check that a set of strings are present somewhere in table
        assertTextInTable("userList",
            new String[] {"Spring", "User"});
    }

    public void testEditUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertFormElementEquals("firstName", "Spring");
        submit("save");
        assertTitleEquals("MyUsers ~ User List");
    }

    public void testDeleteUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertTitleEquals("MyUsers ~ User Details");
        submit("delete");
        assertTitleEquals("MyUsers ~ User List");
    }

    /**
     * Convenience method to get the id of the inserted user
     * Assumes last inserted user is "Spring User"
     */
    public String getInsertedUserId() {
        String[] paths = {"WEB-INF/applicationContext.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(paths);
        List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
        // assumed that user inserted in testAddUser() is last user
        return "" + (User) users.get(users.size() - 1).getId();
    }
}

```

4. In order for jWebUnit to work, modify a bit of the HTML in the existing JSPs. The `testListUsers()` method contains the following line:

```
assertTablePresent("userList");
```

This line verifies that a `<table>` is present in the rendered page. In order for this to work, the table must have an `id` attribute with a value of `"userList"`.

5. Open `web/userList.jsp` and add this attribute so your HTML looks as follows:

```
<table class="list" id="userList">
```

6. Change the Hibernate settings so that the database isn't re-created every time the JVM starts. To do this, open `web/WEB-INF/applicationContext.xml` and change the `"hibernate.hbm2ddl.auto"` setting from `create` to `update`.

```
<prop key="hibernate.hbm2ddl.auto">update</prop>
```

7. Since jWebUnit performs in-container testing (it expects a server to be running), you must run **ant clean deploy** and start Tomcat (**ant deploy reload** if Tomcat is already running).
8. Run this test from your IDE or from the command line using **ant test -Dtestcase=UserWeb**.



WARNING

You may have to delete the database (stop Tomcat, delete "db" in your working directory) if this test doesn't pass on your first attempt.

9. Separate the out-of-container and in-container tests in `build.xml`. To do this, exclude classes with "WebTest" in their name from being executed in the "test" target. Locate this target in `build.xml` and add this line:

```
<batchtest todir="${test.dir}/data" unless="testcase">
  <fileset dir="${test.dir}/classes">
    <include name="**/*Test.class"/>
    <exclude name="**/*WebTest.class"/>
  </fileset>
</batchtest>
```

10. Add a new "test-web" target right after the existing "test" target.

```
<target name="test-web" depends="compile"
  description="Runs tests that required a running server">
  <property name="testcase" value="WebTest"/>
  <antcall target="test"/>
</target>
```

11. Run all your out-of-container tests using **ant test** and all the in-container tests using **ant test-web**.

JSP

The MyUsers sample application you’ve been developing has been using JSP 2.0. This version of JSP is much simpler than previous versions in that you can use its Expression Language (EL) to retrieve values for display. This is not only useful for eliminating scriptlets, but its syntax is similar to that used in [Velocity](#) and [FreeMarker](#). JavaServer Faces (JSF is the J2EE standard web application framework) uses JSPs out-of-the-box. Many folks believe that JSF will quickly become the dominant way of writing Java-based web application, so the information below is applicable for JSF applications as well.

View Resolver Configuration

In order to use JSP, and the Java Standard Tag Library (JSTL) in particular, you must define an `InternalResourceViewResolver` in *web/WEB-INF/action-servlet.xml*.

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      >
    <property name="requestContextAttribute"><value>rc</value></property>
    <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 bean definition, the “requestContextAttribute” exposes Spring’s [RequestContext](#) object as the variable `rc`. If you’re using JSPs, it’s unlikely that you’ll need to use this variable, but you can use it to grab your application’s `contextPath` (that is, `/myusers`) using `${rc.contextPath}`. This object is necessary for views that do not have access to the servlet request (that is, Velocity and FreeMarker templates). The “prefix” and “suffix” values are convenient ways to make the view name a friendly name, rather than a hard-coded path.

As an example, move the JSPs you created from the web directory to *web/WEB-INF/jsp* (you'll need to create the *jsp* directory). Below is a partial screenshot of what your project tree should look like at this point. Leave the following JSPs in the web directory since they're either error pages or used in the SiteMesh decorator:

- 404.jsp
- error.jsp
- index.jsp
- messages.jsp
- taglibs.jsp.

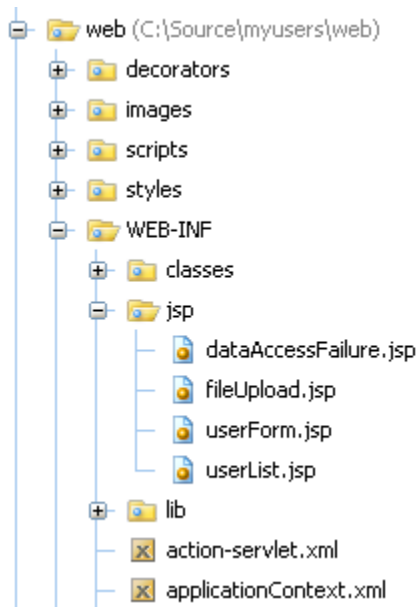


Figure 9.1: Sample project tree

In order to configure Spring to recognize the new location of your JSPs, change the "prefix" of the `viewResolver` bean to `/WEB-INF/jsp/`.

```
<property name="prefix"><value>/WEB-INF/jsp/</value></property>
```

To test this, make sure Tomcat is running and execute **ant remove**. Executing the "remove" target will delete the application from Tomcat, ensuring that your JSPs aren't left over in the root directory of your application. Then type **ant deploy** and wait for Tomcat to re-install your application. Finally, run **ant test-web**.

JSTL

In the previous `viewResolver` bean definition, a `"viewClass"` property is set.

```
<property name="viewClass">
  <value>org.springframework.web.servlet.view.JstlView</value>
</property>
```

The value of the property refers to the view class Spring uses to determine the viewable page. In this case, it's using `JstlView`. This class is helpful because it exposes Spring's local and message bundle to JSTL's formatting and message tags. If you're using JSPs, this is the view class you'll want to use.

The Display Tag

If you're using JSP and displaying lists of data, check out the [display tag library](#). The display tag is a JSP tag that provides paging and sorting tables of data. To show you its functionality, implement it in the `MyUsers` application. The JAR file for version 1.0 (RC1) is included in the download for this chapter, as well as any CSS and images. You can also [download it from SourceForge](#).

1. Change the `userList.jsp` file (in `web/WEB-INF/jsp`) to have the following contents:

```
<%@ include file="/taglibs.jsp"%>
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>

<head>
  <title>MyUsers ~ User List</title>
  <link href="styles/displaytag.css" type="text/css" rel="stylesheet"/>
</head>

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

<c:set var="idKey"><fmt:message key="user.id"/></c:set>
<c:set var="firstNameKey"><fmt:message key="user.firstName"/></c:set>
<c:set var="lastNameKey"><fmt:message key="user.lastName"/></c:set>

<display:table name="${users}" class="list" requestURI="" id="userList"
  export="true">
  <display:column property="id" sort="true" href="editUser.html"
    paramId="id" paramProperty="id" title="${idKey}"/>
  <display:column property="firstName" sort="true"
    title="${firstNameKey}"/>
  <display:column property="lastName" sort="true" title="${lastNameKey}"/>
  <display:setProperty name="basic.empty.showtable" value="true"/>
</display:table>
```

2. Run **ant deploy reload** and go to <http://localhost:8080/myusers/users.html> in your browser. You should see a table like the one below. The table below is sorted by First Name.

Add User

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

Export options: ☒ Excel | ☐ XML | ☐ CSV

Figure 6.2: Result of using the tag library

You can use the **Excel**, **XML**, and **CSV** links at the bottom of the table to export the table to the indicated format.



TIP

Use this tag library in your next demo. Most people will be impressed by the ability to sort columns in a web application.

Tiles

Tiles is a nice page decoration framework that was discussed in *Chapter 5*. The viewResolver configuration for Tiles is similar to JSTL: simply set the “viewClass” property to use [TilesView](#) or [TilesJstlView](#). I recommend the JSTL version because it gives you all the features of TilesView, and it gives you access to JSTL’s formatting tags in your pages.

```
<property name="viewClass">
<value>org.springframework.web.servlet.view.tiles.TilesJstlView</value>
</property>
```

To use Tiles, you must set up a bean in *web/WEB-INF/action-servlet.xml* to read your Tiles configuration file. The easiest way is to use Spring’s [TilesConfigurer](#) class and specify your definition files.

```
<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>
```

Using JSPs to code your view is a common way to develop J2EE applications. However, it’s not the only way. You can use other template frameworks like Velocity or FreeMarker, both of which are lightweight and fast. Better yet, Spring makes them *very* easy to use and eliminates the hard part of using them: setup and configuration.

Velocity

Velocity is an open-source project hosted at <http://jakarta.apache.org/velocity>. Its primary use is generating dynamic web pages for web browsers. However, since it is a *templating* technology, you can use it wherever you need templates (for example, sending HTML-based e-mail). Velocity runs the same in any J2EE container, giving it an advantage over JSPs, which can have quirky behavior and performance issues from one container to the next. Velocity has a templating language called [Velocity Template Language](#) (VTL) that uses syntax similar to JSP 2.0's EL. For example, to print out the value of a "lastName" property on a user object, you would use `${user.lastName}` in JSP 2.0. In Velocity, the syntax doesn't have to change at all: `${user.lastName}`. This is known as *formal reference notation*. You can also use the *shorthand* syntax: `$user.lastName`. This chapter uses the formal notation since it's the same syntax used by JSP 2.0 and FreeMarker.

Velocity works by grabbing items from a *context* and displaying them on your page. Of course, you must put items into the context first. Stuffing items into the context is easy, but Spring makes it even easier.

The next section converts the MyUsers application from JSP to Velocity. You will start by configuring Velocity with Spring. This chapter requires that you have the following Velocity JARs in your classpath (*web/WEB-INF/lib*):

- velocity-1.4.jar
- velocity-tools-view-1.1.jar

Using Velocity in MyUsers

In the MyUsers application, SiteMesh is the page decorator. SiteMesh allows you to leave the JSP decoration intact and use a JSP-based decorator to *decorate* Velocity-powered pages. However, you may want to use Velocity for its decorator template as well. In this section, you'll start by configuring Velocity in the *action-servlet.xml* file. Then you'll modify SiteMesh to use a Velocity decorator.

View Resolver Configuration

When you add Velocity to a web application, you usually have to add a *velocity.properties* file to the classpath. This file has settings in it to tell Velocity which [ResourceLoader](#) to use for loading templates. For convenience, Spring has a [VelocityConfigurer](#) class that allows you to specify a path to your templates (that is, pages) without configuring such a file.

1. Add the XML fragment below to *web/WEB-INF/action-servlet.xml*. This tells Velocity to load templates from your application's *WEB-INF/velocity* directory.

```
<bean id="velocityConfig"

class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="resourceLoaderPath">
    <value>/WEB-INF/velocity/</value>
  </property>
</bean>
```

2. If you want more control of your ResourceLoader (for example, to load templates from a database), create a *velocity.properties* file. To load the file, specify a `configLocation` property and value on your `velocityConfig` bean. You can also specify the properties directly in the bean definition by replacing `configLocation` with `properties`. For more information on using an alternative ResourceLoader, see [Spring's velocity.properties documentation](#).
3. Declare the [VelocityViewResolver](#) as the view resolver for your application.



TIP

Make sure to comment out the [InternalResourceViewResolver](#) used previously for JSPs.

```

<bean id="viewResolver" class="org.springframework.web.servlet.view.
    velocity.VelocityViewResolver">
    <property name="exposeSpringMacroHelpers"><value>true</value></property>
    <property name="requestContextAttribute"><value>rc</value></property>
    <property name="cache"><value>true</value></property>
    <property name="prefix"><value>/WEB-INF/velocity/</value></property>
    <property name="suffix"><value>.vm</value></property>
</bean>

```

The first two properties in this definition, "exposeSpringMacroHelpers" and "requestContextAttribute", are important. The first exposes macros for easy handling of forms and validation errors. The "requestContextAttribute" property is an alias to the [RequestContext](#) object. This object is useful for printing localized messages from the *messages.properties* file.

The `VelocityViewResolver` inherits a number of optional properties from its parent, the `AbstractTemplateViewResolver`. The table below illustrates these properties and their descriptions. These properties are applicable for any `AbstractTemplateViewResolver` (such as the `FreeMarkerViewResolver`).

Table 6.2: Additional TemplateViewResolver Properties

Property Name	Description	Default
exposeRequestAttributes	Sets whether all request attributes are added to the model prior to merging with the template.	false
exposeSessionAttributes	Sets whether all session attributes are added to the model prior to merging with the template.	false
exposeSpringMacroHelpers	Sets whether to expose a <code>RequestContext</code> for use by Spring's macro library, under the name <code>springBindRequestContext</code> .	false

To use Velocity with Spring, you must specify two bean definitions in your **-servlet.xml* file (`VelocityConfigurer` and `VelocityViewResolver`). Of course, there's a lot more work to do. You must create your Velocity templates and learn how to render error messages in VTL. In the *MyUsers* application, you must also learn how to do page decoration with SiteMesh's [Velocity Decorator](#).

SiteMesh and Velocity

The most difficult part of converting MyUsers to use Velocity instead of JSP has been configuring SiteMesh to use a Velocity Decorator. Fortunately, SiteMesh 2.1 added this functionality and it's [now fairly easy](#).

1. Create a decorator similar to *web/decorators/default.jsp*; however, this one should use Velocity instead of JSP tags to pull in SiteMesh content. SiteMesh ships with a [VelocityDecoratorServlet](#) that pre-populates the context with several useful variables.

Table 6.3: SiteMesh Velocity Context Objects

<code>\${request}</code>	The <code>HttpServletRequest</code> object
<code>\${response}</code>	The <code>HttpServletResponse</code> object
<code>\${base}</code>	<code>request.getContextPath()</code>
<code>\${title}</code>	Parsed page <code><title></code>
<code>\${head}</code>	Parsed page <code><head></code>
<code>\${body}</code>	Parsed page <code><body></code>
<code>\${page}</code>	SiteMesh's internal Page object

The VTL syntax is actually quite a bit cleaner than the JSP syntax (for example, `<decorator:title default="MyUsers"/>` becomes simply `${title}`).

2. Below is the *default.jsp* decorator re-written as a Velocity decorator. Use these lines as a guide to create *web/decorators/default.vm*. Strikethroughs represent text that you must delete. Underlines represent text that you must change. To save on space, ellipses represent blocks of text that don't require any changes and therefore are not displayed here.

Notice that the VTL syntax is actually quite a bit cleaner than the JSP syntax (i.e. `<decorator:title default="MyUsers"/>` becomes simply `${title}`).



TIP

Copy *web/decorators/default.jsp* to *web/decorators/default.vm* to prepare for the Velocity conversion 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>${title}</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <e:set var="ctx" value="`${pageContext.request.contextPath}`"
scope="request">
```



```

<link href="${base}/styles/global.css" type="text/css" rel="stylesheet"/
>
<link href="${base}/images/favicon.ico" rel="SHORTCUT ICON"/>
${head}
...
<h1><span>Welcome to MyUsers</span></h1>
<div id="logo" onclick="location.href='${base}'"
    onkeypress="location.href='${base}'"></div>
<h2><span>Spring Rocks!</span></h2>
...
<div id="content">
    #parse("/messages.vm")
    ${body}
</div>
</div>

<div id="supportingText">
    <div id="underground">${page.getProperty("page.underground")}</div>

    <div id="footer">
        ...
        <a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${request.requestURL}&output=Submit&gl=sec508&am
p;test=" title="Check the accessibility of this site according to U.S.
Section 508">508</a> &middot;
        <a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${request.requestURL}&output=Submit&gl=wcag1-
aaa&test=" title="Check the accessibility of this site according to
WAI Content Accessibility Guidelines 1">aaa</a>
    </div>

```

3. In the middle of this file, it parses and includes the *messages.vm* file. This file does not exist, so create it in the *web* directory.

```

## Success Messages
#if ($message)
<div class="message">${message}</div>
${request.session.removeAttribute("message")}
#end

```

4. Once you've created the Velocity template, configure SiteMesh to use it. Open *web/WEB-INF/decorators.xml* and change the default decorator's page attribute to refer to the *default.vm* file you just created.

```

<decorators defaultdir="/decorators">
    <decorator name="default" page="default.vm">
        <pattern>/*</pattern>
    </decorator>
</decorators>

```

5. Configure the application to use SiteMesh's `VelocityDecoratorServlet` to parse the Velocity templates.

Edit `web.xml`

In order for SiteMesh to properly parse the Velocity decorator, add a servlet definition and associated mapping to `web/WEB-INF/web.xml`. Add the XML below right after the `action` servlet declaration:

```
<servlet>
  <servlet-name>sitemesh-velocity</servlet-name>
  <servlet-class>com.opensymphony.module.sitemesh.velocity.VelocityDecor
    atorServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>sitemesh-velocity</servlet-name>
  <url-pattern>*.vm</url-pattern>
</servlet-mapping>
```

These instructions are also [available in SiteMesh's documentation](#).

Create Velocity Templates

The last step to converting from JSP to Velocity is to change all the JSP pages to use Velocity's VTL.

Copy the `WEB-INF/jsp` directory to `WEB-INF/velocity` and rename all the files to use a `.vm` extension. Your directory structure should be the same as the one in Figure 6.3.

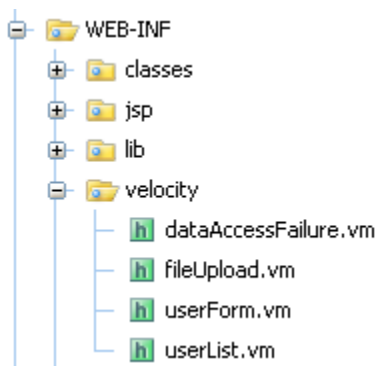


Figure 6.3: Velocity templates directory structure

Below is a list of the four Velocity templates shown above, and the changes that are necessary to make them work with Velocity. Special notes for each file are directly after the code. Any VTL code that you'll need in your templates is underlined.

userList.vm

```
<title>MyUsers ~ User List</title>

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

<table class="list" id="userList">
<thead>
<tr>
  <th>${rc.getMessage("user.id")}</th>
  <th>${rc.getMessage("user.firstName")}</th>
  <th>${rc.getMessage("user.lastName")}</th>
</tr>
</thead>
<tbody>
  #foreach ($user in $users)
  #if ($velocityCount % 2 == 0) <tr class="even">
  #else <tr class="odd">
  #end
    <td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
    <td>${user.firstName}</td>
    <td>${user.lastName}</td>
  </tr>
  #end
</tbody>
</table>
```

Notice the `${rc.getMessage() }` call to get localized messages from `web/WEB-INF/classes/messages.properties`. The `$velocityCount` variable is an internal Velocity value that's exposed when iterating.

userForm.vm

```
<title>MyUsers ~ User Details</title>

#springBind("user.*")
#if ($status.error)
<div class="error">
  #foreach ($error in $status.errorMessages)
    ${error}<br/>
  #end
</div>
#end
```

```

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

<form method="post" action="editUser.html">
#springBind("user.id")
<input type="hidden" name="id" value="${status.value}"/>
<table>
<tr>
<th>${rc.getMessage("user.firstName")}:</th>
<td>
#springBind("user.firstName")
<input type="text" name="firstName" value="${status.value}"/>
<span class="fieldError">${status.errorMessage}</span>
</td>
</tr>
<tr>
<th>${rc.getMessage("user.lastName")}:</th>
<td>
#springBind("user.lastName")
<input type="text" name="lastName" value="${status.value}"/>
<span class="fieldError">${status.errorMessage}</span>
</td>
</tr>
<tr>
<td></td>
<td>
<input type="submit" class="button" name="save" value="Save"/>
#if ($user.id)
<input type="submit" class="button" name="delete" value="Delete"/>
#end
</td>
</tr>
</table>
</form>

```

Notice the `#springBind` macro call that exposes variables for each field. The exclamation point after the dollar sign (`${!{...}}`) indicates that nothing should be printed if no value is found.

These macros don't have a closing tag or ending statement like the `<spring:bind>` JSP tags require. The `#springBind` macro becomes available when you set the `exposeSpringMacroHelpers` property to `true` on the `viewResolver` bean.

Client-side validation with JavaScript and Commons Validator is only supported with JSPs.

fileUpload.vm

```

<h3>File Upload</h3>

#if ($model.filename)
  <p style="font-weight: bold">
    Uploaded file (click to view): <a
href="${model.url}">${model.filename}</a>
  </p>
#end

<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>

```

dataAccessFailure.vm

```

<h3>Data Access Failure</h3>
<p>
  ${exception}
</p>

<a href="${rc.contextPath}">&#171; Home</a>

```

This file does not have the Exception's stack trace printed out in a comment like the JSP. I tried putting `${exception.printStackTrace() }` in a comment, but the stack trace is never printed.

Deploy and Test

Now that you've told Spring to configure and use Velocity, configured SiteMesh to use a Velocity Decorator and converted all the JSPs to Velocity, it's time to test that everything worked. Start Tomcat and run **ant clean deploy reload**. Once the new context has restarted, run **ant test-web** to verify everything works.

**WARNING**

If the test fails because a user is not found in the database, delete the database (`rm -r db`), restart Tomcat and try again. You can also add `<delete dir="db"/>` to the "delete" target.

Summary of Velocity

In this section, you learned how to use Velocity as an alternate view technology to JSP. Velocity tends to have a much cleaner syntax than JSP (though JSP has improved with version 2.0). Compilation is much faster than JSP when you first access a Velocity-backed template. With JSP, the initial load time can be a couple seconds, which is slightly painful when developing since you constantly have to wait. It's not something you notice until you've developed with Velocity and then go back to JSPs. The one downside to Velocity is that you can't use the rich set of tag libraries (for example, `displaytag`, `oscache`, etc.) that are available. However, a fast templating solution that allows you to use JSP tags is available: FreeMarker.

FreeMarker

FreeMarker is an open-source project hosted at <http://freemarker.sourceforge.net>. It is similar to Velocity in that it's a *template engine*, which generates text output based on templates. The major difference between the two libraries is their template language syntax. FreeMarker is more robust, yet easier to work with. It also supports using JSP tag libraries in your templates (so you can use `displaytag!`). For more information on the differences between the two, see the [FreeMarker vs. Velocity](#) web page.

The next section converts the MyUsers application from Velocity to FreeMarker. Start by configuring FreeMarker with Spring. This chapter requires you to have FreeMarker's *freemarker.jar* in your classpath (*web/WEB-INF/lib*).



NOTE

Spring requires FreeMarker version 2.3 or higher.

View Resolver Configuration

1. Add a *configurer* bean definition to *web/WEB-INF/action-servlet.xml*. Comment out the *velocityConfig* bean as part of this exercise.

```
<bean id="freemarkerConfig"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigur
er">
    <property name="templateLoaderPath"><value>/</value></property>
</bean>
```

Like the *VelocityConfigurer*, you can configure this class with a properties file (with a *configLocation* property pointing to the file). You can also set properties on the bean itself by specifying a *freemarkerSettings* list of properties.

2. Comment out the *velocityConfig* bean and the *viewResolver* bean, and add one for FreeMarker. The properties on this resolver are very similar to the Velocity version, the only difference being the "prefix" and the "suffix".

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.
freemarker.FreeMarkerViewResolver">
    <property name="exposeSpringMacroHelpers"><value>true</value><
property>
    <property name="requestContextAttribute"><value>rc</value></property>
    <property name="prefix"><value>/WEB-INF/freemarker/</value></property>
    <property name="suffix"><value>.ftl</value></property>
</bean>
```

Keep in mind that you can also expose request and session attributes using the `exposeRequestAttributes` and `exposeSessionAttributes` on this template resolver. This tutorial doesn't require them, but they have value.

SiteMesh and FreeMarker

Just like Velocity, SiteMesh has a *decorator servlet* specifically for FreeMarker called [FreeMarkerDecoratorServlet](#). It pre-populates FreeMarker's data model with several context attributes, which are listed in the table below.

Table 6.4: SiteMesh FreeMarker Context Attributes

<code>\${base}</code>	<code>request.getContextPath()</code>
<code>\${title}</code>	Parsed page <code><title></code>
<code>\${head}</code>	Parsed page <code><head></code>
<code>\${body}</code>	Parsed page <code><body></code>
<code>\${page}</code>	SiteMesh's internal Page object

FreeMarker supports getting request parameters, request attributes and session variables, which is why you don't see `$req` and `$res` like you do with Velocity. These values are available in the templates through the variables `Request`, `RequestParameters`, `Session` and `Application` (for example, `${Session["user"]}`).

- Below is the Velocity decorator from the previous section re-written using FreeMarker. The listing displays only the changed lines, so you don't have to copy the entire contents of the file below. Use these lines as a guide to create `web/decorators/default.ftl`. The variable references are the same as Velocity, but the conditional logic syntax is different.



TIP

Copy `web/decorators/default.vm` to `web/decorators/default.ftl` to prepare for the conversion.

```

<div id="content">
    <#include "/messages.ftl"/>
    ${body}
</div>
</div>

<div id="supportingText">
    <div id="underground">
        <#if page.getProperty("page.underground")?exists>
            ${page.getProperty("page.underground")}
        </#if>
    </div>
    ...

```



```

    <a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${Request.requestURL}&output=Submit&gl=sec508&am
p;test=" title="Check the accessibility of this site according to U.S.
Section 508">508</a> &middot;
    <a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${Request.requestURL}&output=Submit&gl=wcag1-
aaa&test=" title="Check the accessibility of this site according to
WAI Content Accessibility Guidelines 1">aaa</a>
    ...

```

2. Create the *web/messages.ftl* file with the contents below:

```

<!-- Success Messages -->
<#if message?exists>
    <div class="message">${message}</div>
</#if>

```

3. Change *web/WEB-INF/decorators.xml* to use the FreeMarker template as the default.

```

<decorators defaultdir="/decorators">
    <decorator name="default" page="default.ftl">
        <pattern>/*</pattern>
    </decorator>
</decorators>

```

Notable differences between Velocity and FreeMarker are listed below:

- To check for a null value in Velocity, you simply write `#if (object.property)` to test for the existence of a value. With FreeMarker, you must append `?exists` to test for nulls.
- Velocity exposes the actual `HttpServletRequest` object, so you can call `${request.contextPath}` and `${request.requestURL}`. FreeMarker only exposes scoped attributes. Because of this, you can't simply remove messages from the session in the *messages.ftl* file.

FreeMarker has some limitations, which arguably make it a cleaner MVC implementation. The easiest and cleanest way to solve the issues above is to create a `ServletFilter` that searches for messages in the session, and if it finds them, it stuffs them in the request. This way, you can forget about removing them in a view page and let the filter do the work.

Below is a *MessageFilter.java* class to put in *src/org/appfuse/web*. The logic to set the "requestURL" in the request is to provide full links back to your application as part of the project's footer page.

```
package org.appfuse.web;

// use your IDE to organize imports

public class MessageFilter implements Filter {
    private static Log log = LogFactory.getLog(MessageFilter.class);

    public void doFilter(ServletRequest req, ServletResponse res,
                        FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;

        // grab messages from the session and put them into request
        // this is so they're not lost in a redirect
        Object message = request.getSession().getAttribute("message");
        if (message != null) {
            request.setAttribute("message", message);
            request.getSession().removeAttribute("message");
        }

        // set the requestURL as a request attribute for templates
        // particularly freemarker, which doesn't allow
        // request.getRequestURL()
        request.setAttribute("requestURL", request.getRequestURL());

        chain.doFilter(req, res);
    }

    public void init(FilterConfig filterConfig) {}

    public void destroy() {}
}
```

- Using this filter, you can eliminate any session removal logic in the all the messages templates (at *web/messages.**). To enable it, add a `<filter>` and `<filter-mapping>` to *web/WEB-INF/web.xml*. Put the following `<filter>` declaration just above the *sitemesh* filter.

```
<filter>
    <filter-name>messageFilter</filter-name>
    <filter-class>org.appfuse.web.MessageFilter</filter-class>
</filter>
```

5. Add the `<filter-mapping>` just above SiteMesh's filter-mapping.

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

Edit web.xml

To edit *web.xml* for SiteMesh, comment out the `sitemesh-velocity` servlet and its mapping and add the following `sitemesh-freemarker` servlet and mapping.

```
<servlet>
  <servlet-name>sitemesh-freemarker</servlet-name>
  <servlet-
class>com.opensymphony.module.sitemesh.freemarker.FreemarkerDecoratorServ
let</servlet-class>
  <init-param>
    <param-name>TemplatePath</param-name>
    <param-value></param-value>
  </init-param>
  <init-param>
    <param-name>default_encoding</param-name>
    <param-value>ISO-8859-1</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>sitemesh-freemarker</servlet-name>
  <url-pattern>*.ftl</url-pattern>
</servlet-mapping>
```

Create FreeMarker Templates

The last step for implementing FreeMarker is to create page templates using FreeMarker's template language.

1. Copy the *WEB-INF/velocity* directory to *WEB-INF/freemarker* and rename all the files to use an FTL extension. Your directory structure should be the same as the one in Figure 6.4.

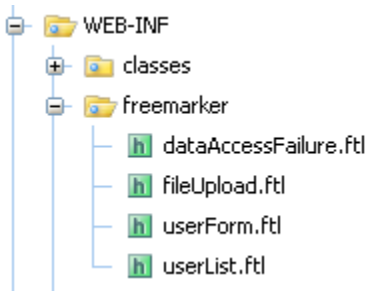


Figure 6.4: FreeMarker Templates directory structure

2. Below is a list of the four FreeMarker templates shown above, and the changes that are necessary to make them work with Velocity. Special notes for each file are directly after the code. Any lines where the FreeMarker code is different from Velocity's VTL are underlined.

userList.ftl

```

<title>MyUsers ~ User List</title>

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

<table class="list" id="userList">
<thead>
<tr>
  <th>${rc.getMessage("user.id")}</th>
  <th>${rc.getMessage("user.firstName")}</th>
  <th>${rc.getMessage("user.lastName")}</th>
</tr>
</thead>
<tbody>
<#list users as user>
<#if user index % 2 == 0> <tr class="even">
<#else> <tr class="odd">
</#if>
  <td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
  <td>${user.firstName}</td>
  <td>${user.lastName}</td>
</tr>
</#list>
</tbody>
</table>

```

Similar to Velocity, the `${rc.getMessage() }` call gets localized messages from *web/WEB-INF/classes/messages.properties*.

userForm.ftl

```

<#import "/spring.ftl" as spring/>

<title>MyUsers ~ User Details</title>

<@spring.bind "user.*"/>
<#if spring.status.error>
<div class="error">
  <#list spring.status.errorMessages as error>
    ${error}<br/>
  </#list>
</div>
</#if>

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

<form method="post" action="editUser.html">

```

```

<@spring.bind "user.id"/>


```

The most important thing to notice in this file is the first line where *spring.ftl* is imported. This file contains the `spring.bind` macro needed to expose a properties value.

You may also notice that each field is given a default by adding `?default('')` to the end of its expression.

These macros don't have a closing tag or ending statement like the `<spring:bind>` JSP tags require. The `<@spring.bind>` macro becomes available when you set the `exposeSpringMacroHelpers` property to `"true"` on the `viewResolver` bean.

Client-side validation with JavaScript and Commons Validator is only supported with JSPs.

fileUpload.ftl

```
<h3>File Upload</h3>

<#if model?exists>
  <p style="font-weight: bold">
    Uploaded file (click to view): <a
  href="{model.url}">${model.filename}</a>
</p>
</#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>
```

dataAccessFailure.ftl

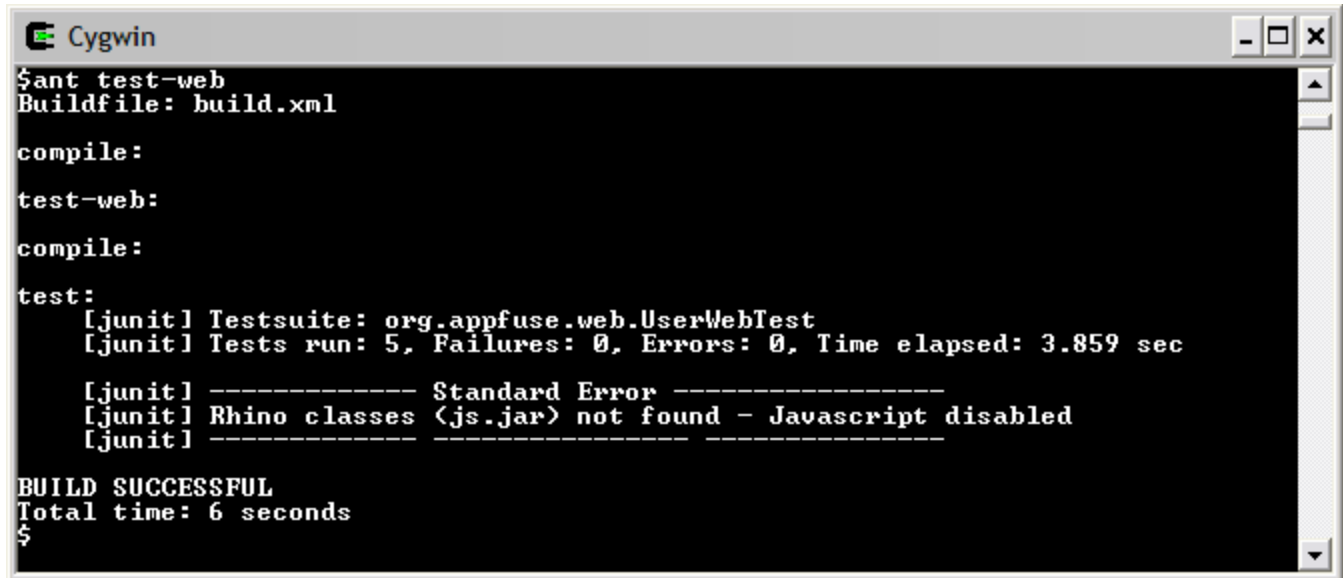
```
<h3>Data Access Failure</h3>
<p>
  ${exception}
</p>

<a href="{rc.contextPath}">? Home</a>
```

The code for this file is the same for FreeMarker and Velocity.

Deploy and Test

Now that you've told Spring to configure and use FreeMarker, configured SiteMesh to use a FreeMarker Decorator and converted all the pages to use FreeMarker, it's time to test that everything worked. Start Tomcat and run **ant clean deploy**. Once everything has started, run **ant test-web** to verify everything works.



```
$ant test-web
Buildfile: build.xml

compile:
test-web:
compile:
test:
[junit] Testsuite: org.appfuse.web.UserWebTest
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 3.859 sec

[junit] ----- Standard Error -----
[junit] Rhino classes <js.jar> not found - Javascript disabled
[junit] -----

BUILD SUCCESSFUL
Total time: 6 seconds
$
```

Figure 6.5: FreeMarker test results

JSP, Velocity and FreeMarker are the dominant view choices when using Spring MVC. However, a few others can be quite useful. They are XSTL (for displaying and transforming XML), PDF and Excel. In the next sections, you'll implement these to generate reports of the user list screen.

XSLT

XSLT describes the process of combining XML and XSL to *transform* XML to another output. In most cases, this output is text-based, but you can also use [XSL-FO](#) to generate PDFs. Spring's XSLT views can be helpful if you're loading and presenting XML documents, or you can easily convert your model to XML.

Before creating an XSLT View, prepare the UserController class to handle rendering report views.

1. Open *UserControllerTest.java* (in *test/org/appfuse/web*) and add the following test:

```
public void testGetUsersAsXML() throws Exception {
    UserController c =
        (UserController) ctx.getBean("userController");
    MockHttpServletRequest request = new MockHttpServletRequest();
    request.addParameter("report", "XML");
    ModelAndView mav =
        c.handleRequest(request, (HttpServletResponse) null);
    assertEquals(mav.getViewName(), "userListXML");
}
```

2. Open *UserController.java* (in *src/org/appfuse/web*) and add some logic to render a different view name if a report parameter is passed in. This logic will also render the Excel and PDF views covered later in this chapter.

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

    String viewName = "userList";
    if (request != null &&
        request.getParameter("report") != null) {
        viewName += request.getParameter("report");
    }
    return new ModelAndView(viewName, "users", mgr.getUsers());
}
```

3. The reports you're about to produce are simply going to contain the user's full name in a list format. To make this easier, add a *getFullName()* method to the *User* class (in *src/org/appfuse/model*).

```
public String getFullName() {
    return firstName + ' ' + lastName;
}
```

Create the View Class

Before creating the class to convert the list of users to XML, create a class to test it.

1. In *test/org/appfuse/web*, create a `UserXMLViewTest` class that extends JUnit's `TestCase`.

```
package org.appfuse.web;

// user your IDE to organize imports

public class UserXMLViewTest extends TestCase {
    private static Log log = LogFactory.getLog(UserXMLViewTest.class);

    public void testXMLCreation() throws Exception {
        // setup a user to print out as XML
        User user = new User();
        user.setFirstName("James");
        user.setLastName("Strachan");
        List users = new ArrayList();
        users.add(user);
        Map model = new HashMap();
        model.put("users", users);

        // invoke the XsltView and call its 'createDomNode' method
        UserXMLView feed = new UserXMLView();
        org.w3c.dom.Node node = feed.createDomNode(model, "users",
            new MockHttpServletRequest(),
            new MockHttpServletResponse());
        assertEquals(node.getFirstChild().toString(),
            "<users><user>James Strachan</user></users>");
    }
}
```

2. In *src/org/appfuse/web*, create a `UserXMLView` class that extends [AbstractXsltView](#) and has the following contents.



NOTE

The XML classes used are from [dom4j](#), which should be in your classpath. Also, the `Node` return type (from `createDomNode()`) is from the W3C's DOM (`org.w3c.dom.Node`).

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserXMLView extends AbstractXsltView {
    protected Node createDomNode(Map model, String rootName,
```

```

        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {
        Document doc = DocumentHelper.createDocument();
        Element root = doc.addElement(rootName);
        doc.setRootElement(root);

        List users = (List) model.get("users");
        for (Iterator it = users.iterator(); it.hasNext();) {
            User user = (User) it.next();
            root.addElement("user").addText(user.getFullName());
        }

        response.setContentType("text/xml");

        return new DOMWriter().write(doc);
    }
}

```

3. Run **ant test -Dtestcase=UserXML** to verify this class is working as expected.



NOTE

In this example, you're going to modify this XML document to produce another XML document (that's why the response is set to be content type "text/xml"). If you were going to produce HTML in your XSL stylesheet, you could eliminate this line.

4. Create a *users.xsl* file in *web/WEB-INF/xsl*. Add the following XSL to this document.

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" omit-xml-declaration="no"/>

  <xsl:template match="/">
    <users>
      <xsl:for-each select="users/user">
        <user><xsl:value-of select="."/></user>
      </xsl:for-each>
    </users>
  </xsl:template>

</xsl:stylesheet>

```

5. Now that you have created all the view files, configure Spring to know about the `userListXML` view name. The easiest way to do this is to create a second `viewResolver` that uses [ResourceBundleViewResolver](#) to resolve its views. Since you already have a `viewResolver` bean for FreeMarker, you must give the new resolver a different id. Using `reportViewResolver` is a good solution:

```
<bean id="reportViewResolver"
      class="org.springframework.web.servlet.view.ResourceBundleView
        Resolver">
  <property name="order"><value>1</value></property>
</bean>
```

6. The "order" property specifies priorities of view resolvers. Add this same property to the `viewResolver` bean with a value of 0. The `ResourceBundleViewResolver` allows you to configure your view names, their classes, and their properties in a `ResourceBundle` (or properties file). By default, this file name is *views.properties*, and it should exist in your *WEB-INF/classes* directory. If you want to override the name of this file, specify a "basename" property on the `reportViewResolver` bean.
7. Create a *view.properties* file in *web/WEB-INF/classes* and fill it with the following text:

```
userListXML.class=org.appfuse.web.UserXMLView
userListXML.stylesheetLocation=/WEB-INF/xsl/users.xsl
userListXML.root=users
```

Deploy and Test

You are now ready to deploy and test the application. Run **ant deploy reload populate** and load <http://localhost:8080/myusers/users.html?report=XML> into your browser. You should see something similar to the screenshot below.

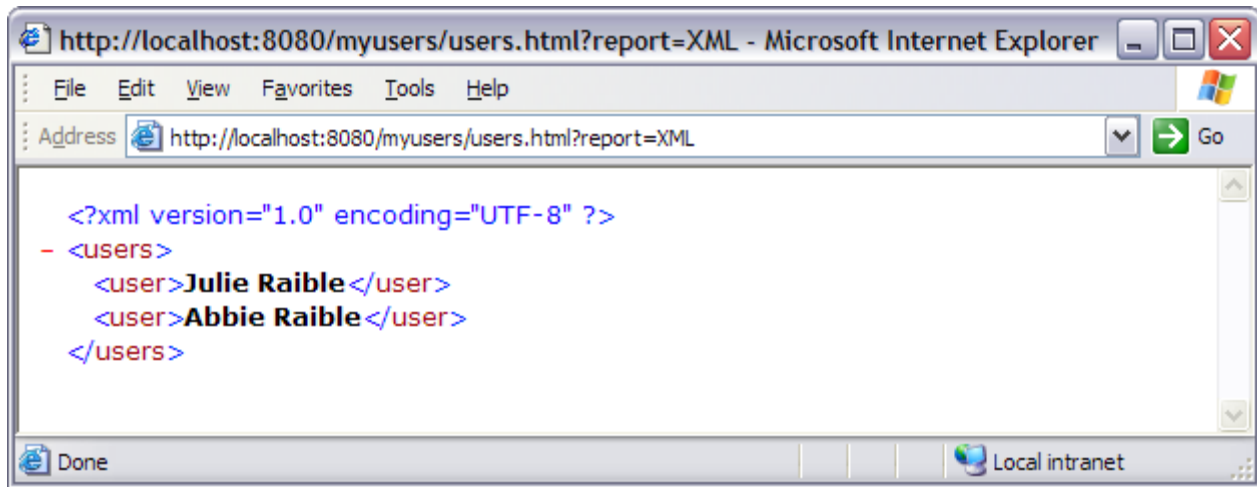


Figure 6.6: View class test result

One report is working; add reports for PDF and Excel too. To make things easier, add a set of links to the current userlist page. If you're still using FreeMarker, the filename is *userList.ftl* and it's located in *web/WEB-INF/freemarker*. Open it and add the following HTML to the top of the file, between the `<button>` and the `<table>`:

```
<p style="text-align: right; margin-bottom: -10px">
<strong>Export Options:</strong>
  <a href="?report=XML">XML</a> &middot;
  <a href="?report=Excel">Excel</a> &middot;
  <a href="?report=PDF">PDF</a>
</p>
```

Excel

Excel documents are a useful way to export data for manipulation by users. If you simply need to output a list screen in Excel, I recommend using the Display Tag Library (described earlier). If you need something more robust, then this section is for you. The next few steps show you how to build and send an Excel spreadsheet using Jakarta's [POI](#) library.



NOTE

If you [downloaded](#) and installed the *chapter6-jars.zip* file, you will already have *poi-2.5-final-20040302.jar* in your classpath. This JAR contains classes that are necessary to complete this exercise.

Create the View Class

1. Create a `UserExcelView` class in `src/org/appfuse/web`. This class should extend `AbstractExcelView` and contain the following code:

```
package org.appfuse.web;

//use your IDE to organize imports

public class UserExcelView extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook wb,
                                       HttpServletRequest req,
                                       HttpServletResponse resp)
        throws Exception {

        HSSFSheet sheet = wb.createSheet("My Users");
        sheet.setDefaultColumnWidth((short) 12);

        List users = (List) model.get("users");

        for (int i = 0; i < users.size(); i++) {
            HSSFCell cell = getCell(sheet, i, 0);
            setText(cell, ((User) users.get(i)).getFullName());
        }
    }
}
```

Notifying Spring that the `userListExcel` view exists is quite easy using the `ResourceBundleViewResolver`.

2. Open `web/WEB-INF/classes/views.properties` and add the following line:

```
userListExcel.class=org.appfuse.web.UserExcelView
```

Deploy and Test

1. Run `ant deploy reload` and point your browser to <http://localhost:8080/myusers/users.html>.

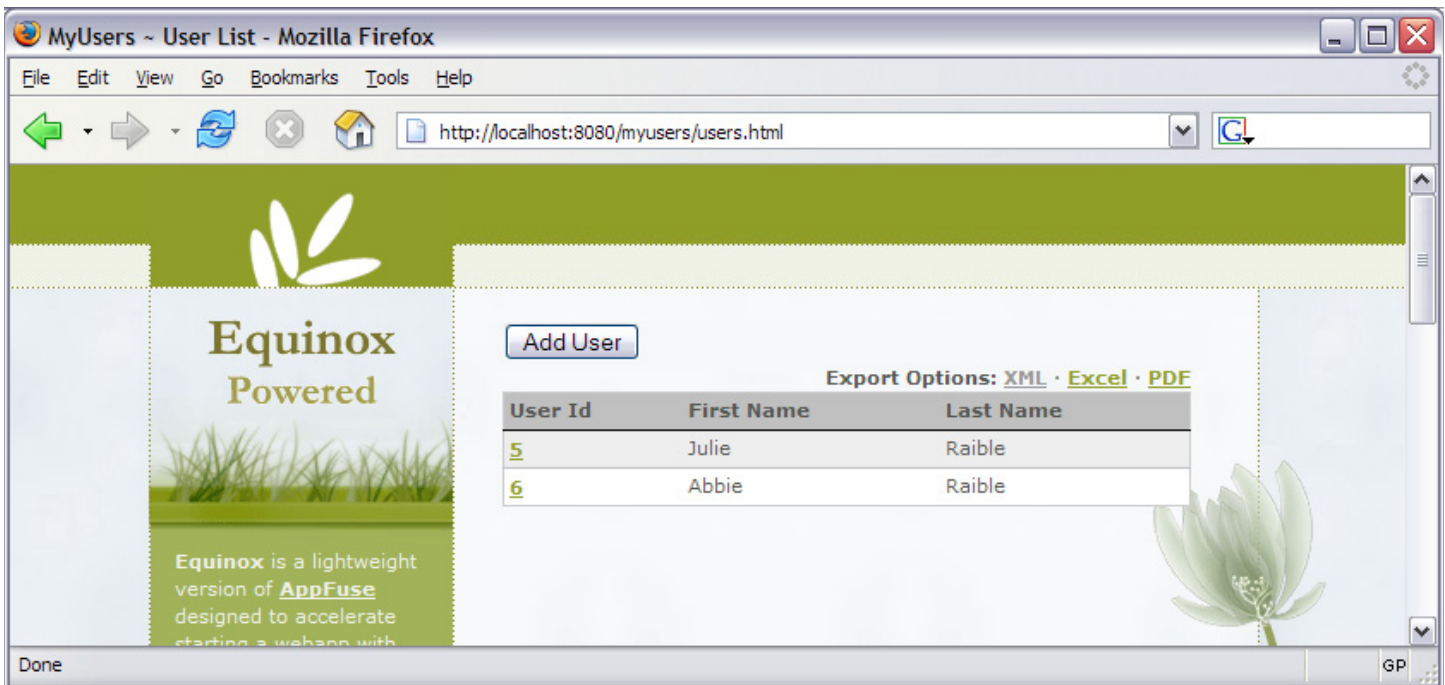


Figure 6.7: Excel view class test results

2. Click on the **Excel** link at the top-right of the table. If you have Excel installed, it starts Excel and brings up a list of the current users.

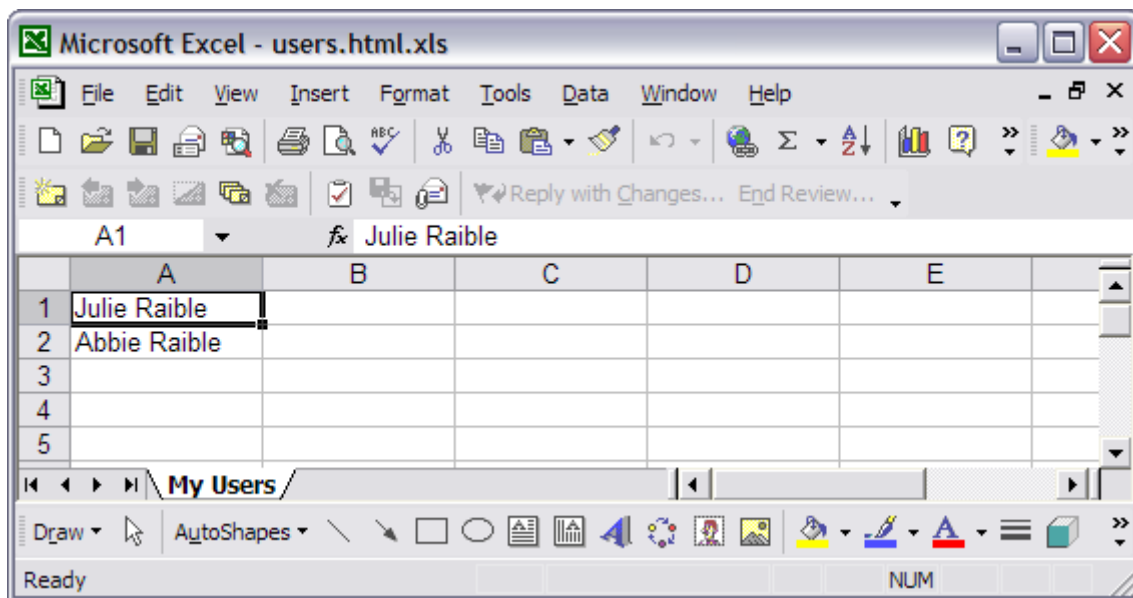


Figure 6.8: Results displayed in Excel

PDF

PDF documents are an excellent way to produce printable reports. In most MVC frameworks, you're required to use something like [JasperReports](#) to produce PDF output. With Spring, you simply create a subclass of [AbstractPdfView](#) and override the `buildPdfDocument()` method. This method returns an [iText](#) document that can be easily rendered in your browser.



NOTE

If you [downloaded](#) and installed the *chapter6-jars.zip* file, you will already have *itext-1.02b.jar* in your classpath. This JAR contains classes that are necessary to complete this exercise.

Create the View Class

1. Create a `UserPDFView` class in `src/org/appfuse/web`. Populate this class with the code below.

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserPDFView extends AbstractPdfView {
    protected void buildPdfDocument(Map model, Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {
        List users = (List) model.get("users");

        for (int i = 0; i < users.size(); i++) {
            String fullName = ((User) users.get(i)).getFullName();
            doc.add(new Paragraph(fullName));
        }
    }
}
```

2. To notify Spring where the `userListPDF` view is, open `web/WEB-INF/classes/views.properties` and add the following line:

```
userListPDF.class=org.appfuse.web.UserPDFView
```


Deploy and Test

1. Run `ant deploy reload` and point your browser to <http://localhost:8080/myusers/users.html>.
2. Click on the **PDF** link at the top-right of the table. This opens a PDF containing a list of the user's names.

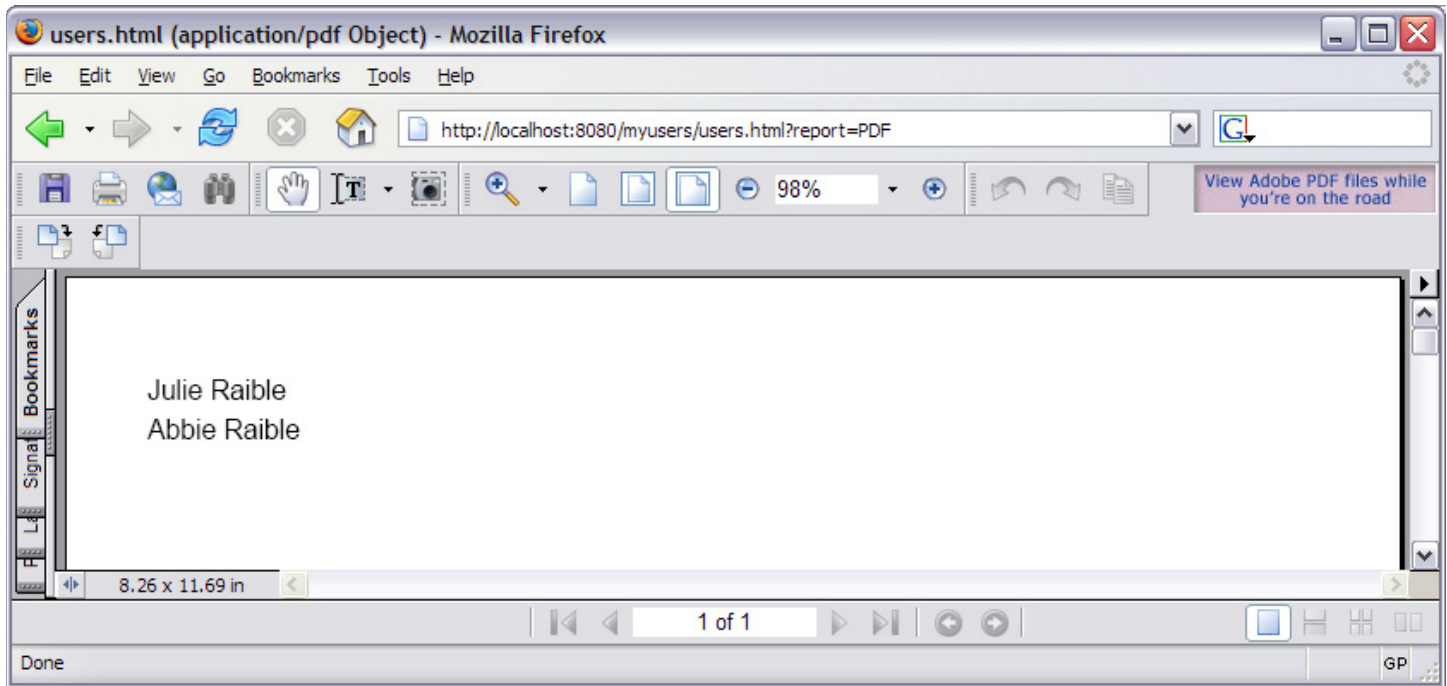


Figure 6.9: Results displayed in PDF

Summary

This chapter explored the rich functionality offered in Spring's MVC framework. It showed you how to easily change from using JSP to Velocity without changing a single line of Java code. Then it showed you how to change that to FreeMarker by simply altering some XML files and re-working the templates a bit. You also learned how to use SiteMesh with Velocity and FreeMarker. The ability to use all of these J2EE templating engines with a highly configurable page decoration engine like SiteMesh is a *very* powerful solution for quickly developing web applications. Spring makes it easy to switch from one to the other, so your templating technology of choice is not a *hard-and-fast* decision.

The ability to produce reports in PDF and Excel is another powerful and easy-to-use feature of Spring MVC. It's even nicer that these features are powered by solid and well-supported open-source projects like iText and POI. The reporting examples in this chapter are simple, but the main goal is to show you how to get the ball rolling - not how to win the game.

Persistence Strategies: Hibernate, iBATIS, JDBC, JDO and OJB

How to Integrate 5 Popular Persistence Frameworks into Your Application

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). This chapter refactors the MyUsers application to support both JDBC and iBATIS as persistence framework options. It also implements the UserDAO using JDO and OJB to showcase Spring's excellent support for these frameworks.

Overview

Most modern applications talk to a database to load and store their information. Persistence is the process of retrieving, saving and deleting data from a data store (usually a relational database). Persistence is a critical feature in web applications, if only for loading and displaying information. For many years, this has been the ugly part of Java. You *could* use JDBC, and it usually works across database vendors. After all, the point of the JDBC API is to provide a *standard* for accessing databases from Java. However, JDBC is difficult to write, especially for a newcomer. It requires developers to catch exceptions and close connections in a final block, which many Java rookies forget to do. Additionally, the exceptions thrown by different JDBC Driver vendors are not standard, so an error code on one server might mean something completely different on another server.

With the advent of Spring's persistence support, many of the issues with JDBC disappear. Its JDBC framework converts JDBC's checked exceptions to a common hierarchy of `RuntimeExceptions`. These exceptions provide precise information about what went wrong, which is much better than [SQLException](#) reports. It uses closures to handle closing database connections, and it includes a set of common SQL error codes for numerous database types.



NOTE

A *closure* is an object that's represented as a block of code (within a method). You can use this object like any Java object, such as parameter, variable, etc. For more information, see the [Java Glossary](#). Charles Miller also has a [tutorial on Closures and Java](#).

Spring provides support classes for numerous other persistence frameworks, including Hibernate, iBATIS, JDO and OJB. It even uses a common methodology in its support, further simplifying the learning curve from one framework to the other. If you learn how to use Spring's Hibernate support, it's very easy to use Spring's iBATIS or JDO APIs.

In this chapter, you will learn more about Spring's Hibernate support and how MyUsers uses Hibernate. From there, you will implement DAOs using iBATIS, Spring JDBC, JDO and OJB. By the end of this chapter, you will know how to use these technologies and configure them with the Spring Framework. The purpose of this chapter is to briefly introduce persistence options with Spring and how to configure them. You will not learn how to solve complicated persistence problems with each framework. Please consult the framework's perspective project page for that information.



NOTE

Transactions will be covered in *Chapter 10*.

Spring has a common theme when it comes to persistence:

- Provide easy configuration using dependency injection.
- Provide base DaoSupport classes for the popular persistence frameworks, including Hibernate, JDBC, iBATIS, JDO and OJB.
- Convert the checked exceptions for the different frameworks to a common exception hierarchy, relieving your upper-level classes of the specifics of the persistence layer. Figure 7.1¹ illustrates this hierarchy.
- Provide easy-to-use *Template* classes that reduce many DAO methods to one-liners. These templates can be used standalone or obtained from the DaoSupport parent classes.
- Open and close any necessary resources (such as database connections) for you.

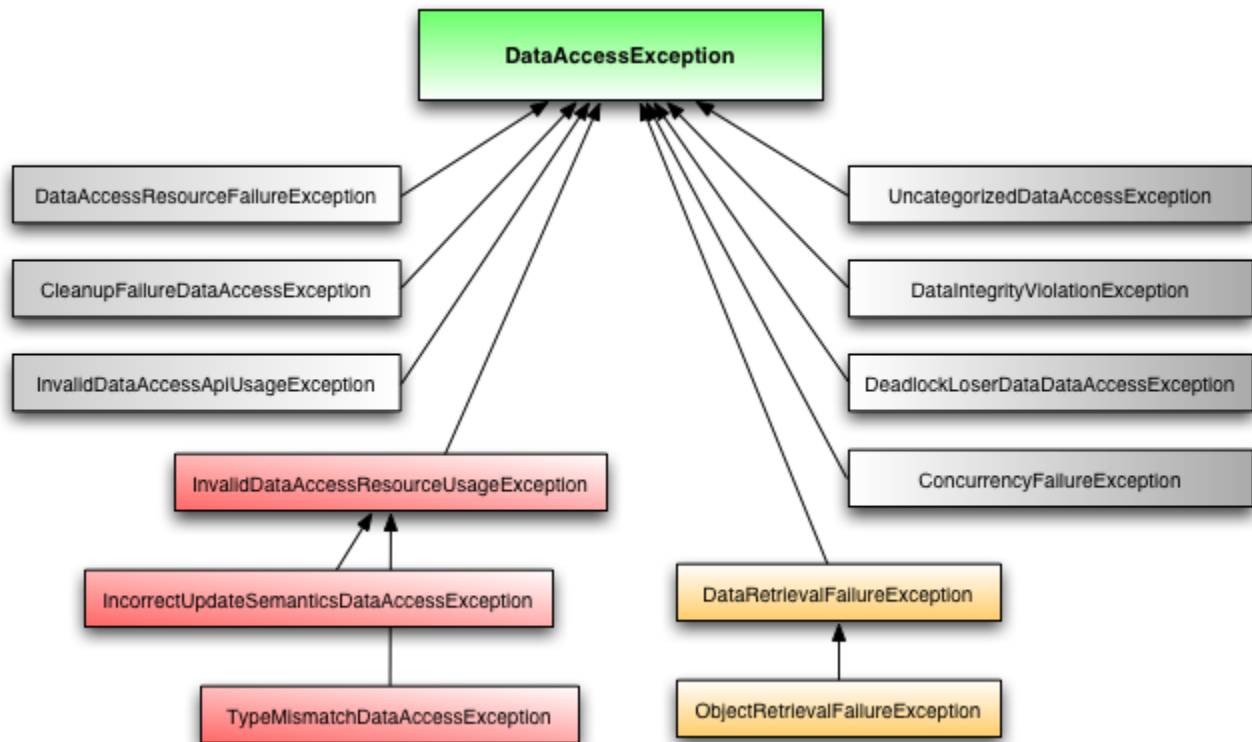


Figure 9.1: Common Exceptions Hierarchy

The code in this chapter should be easy to test since you already have a `UserDAOTest` that does not contain any Hibernate-specific code. This test will verify your implementations. Each persistence framework section will show you how to configure the respective framework with Spring, and how to implement the `UserDAO`.

1. Diagram modeled from the one found at <http://www.springframework.org/docs/reference/dao.html>.

Preparing for the Exercises

The best way to learn is to follow along with the exercises in this chapter. The easiest way to do this is to download the **MyUsers Chapter 7** bundle from <http://sourcebeat.com/downloads>. This project tree is the result of *Chapter 6* exercises, and contains all the JARs you will use in this chapter. You can also use the application you've been developing in previous chapters. If you go this route, download **Chapter 7 JARs** from <http://sourcebeat.com/downloads>. The *Configuration* sub-topic of each section describes the dependencies each project needs. You can use this chapter as a reference for integrating these principles into your own applications.

If you're using a MyUsers application from a previous chapter, you must change the way that it loads context files. Rather than just loading *applicationContext.xml*, change tests and *web.xml* to load *applicationContext*.xml*. This is for convenience. When creating context files for each persistence option, it's easier to specify a wildcard to match the filename. Otherwise, you'd have to change the tests and *web.xml* each time you switched persistence layers.

The following files are affected by this change:

- test/org/appfuse/dao/BaseDAOTestCase.java
- test/org/appfuse/service/UserManagerTest.java
- test/org/appfuse/web/UserControllerTest.java
- test/org/appfuse/web/UserFormControllerTest.java
- web/WEB-INF/web.xml

Because you'll be isolating all the database configurations to one file, you must also change any `<ref local>` directives in *applicationContext.xml* to `<ref bean>`, so other files can contain bean definitions. The baseline *applicationContext.xml* file for this chapter should not contain references to the following beans: `dataSource`, `sessionFactory`, `transactionManager`, and `userDAO`. If they exist in your version, please delete them. Making the above changes will allow you to create different context files for each DAO implementation.

Hibernate

[Hibernate](#) is an open-source Object/Relational Mapping (ORM) solution. ORM is the technique of mapping an Object model to a Relational model (usually represented by a SQL database). Hibernate was created in late 2001 by Gavin King and a handful of other developers. Since then, Hibernate has become a very popular persistence framework in the Java Community. It's become so popular in fact, that the next versions of EJB and JDO are using Hibernate as a source of good ideas. The reasons for its popularity are mainly due to its good documentation, ease of use, excellent features and [smart project management](#). Hibernate's license is [LGPL](#), which means you can use it for free as long as you don't modify its source code. More information on its license is available on the [Hibernate website](#).

Hibernate frees you from hand-coding JDBC. Rather than using SQL and JDBC, you can use domain objects (which are usually POJOs) and simply create XML-based *mapping files*. These files indicate which fields (in an object) map to which columns (in a table). Hibernate has a powerful query language called Hibernate Query Language (HQL). This language allows you to write SQL, but also use object-oriented semantics. One of the best parts about its query language is you can literally guess at its syntax and get it right.

Hibernate's [Session](#) interface is similar to a database connection in that it must be opened and closed at appropriate times to avoid errors and memory leaks. In my opinion, the biggest advantage of using Spring with Hibernate is that you don't have to manage opening and closing these Sessions – everything just works.



NOTE

Spring's Hibernate support classes are located in the [org.springframework.orm.hibernate](#) and [org.springframework.orm.hibernate.support](#) packages.

Although *Chapter 2* covers integrating Hibernate into MyUsers, this chapter covers it as well to provide a single chapter describing Spring's persistence support. If you downloaded the *myusers-ch7* bundle, the Hibernate configurations are eliminated so you can start with a clean slate.

Dependencies

Hibernate has a number of third-party libraries it depends on. All of these are available as part of the [Hibernate download](#). Below are the JARs included in the MyUsers download as part of Hibernate 2.1.6. They are all required, except for the one marked optional. Spring requires Hibernate 2.1 or above for its support classes.

- **hibernate2.jar** – Hibernate core
- **c3p0-0.8.4.5.jar** – Basic connection pool for running unit tests
- **cglib-full-2.0.2.jar** – Code Generation Library for generating proxies for persistent classes
- **dom4j-1.4.jar** – XML library for parsing configuration and mapping files
- **ehcache-0.9.jar** – A pure Java, in-process cache; the default cache for Hibernate
- **jta.jar** – Java Transaction API
- **odmg-2.0.1.jar** – Standard for object-relational mapping products; [superseded by JDO](#)
- (optional) **oscache-2.0.1.jar** and (cluster-aware) **swarmcache-1.0rc2.jar** – Alternative caching implementations



TIP

While Spring does not support Hibernate 3 (now in beta), a [patch is available](#).

Configuration

To make your objects persistable with Hibernate, first create a mapping file. (This chapter assumes you have already created a User POJO in `src/org/appfuse/model`, with `id`, `firstName` and `lastName` properties.

1. In the `src/org/appfuse/model` directory, create a `User.hbm.xml` file 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>
```

In the above mapping, the `<id>` element uses `"increment"` to indicate *max value + 1* for the generated primary key. The generator type of `"increment"` is not recommended for a cluster. Fortunately, Hibernate has [many other options](#).

2. Create an *applicationContext-hibernate.xml* file in the *web/WEB-INF* directory and add a bean definition for a [DataSource](#). You can use an existing *applicationContext*.xml* as a template. This file should have the *spring-beans.dtd* and root `<beans>` element defined before the bean definition.

The "dataSource" bean in this example uses an [HSQL](#) database, which is a pure Java database that runs from a simple *hsqldb.jar* file in the *web/WEB-INF/lib* directory. Later, you'll change this to use [MySQL](#) to see how easy it is to change databases.

```
<?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>

    <!-- Add additional bean definitions here -->
</beans>
```

The [DriverManagerDataSource](#) is a simple `DataSource` that configures a JDBC Driver via bean properties. You can also configure a JNDI `DataSource` if you'd rather use your container's pre-configured `DataSource`. For example, a common strategy is to use the `DriverManagerDataSource` for testing, and the JNDI `DataSource` (below) for production.



NOTE

Chapter 8 shows how to mock the JNDI `DataSource` and use the configuration below for both testing and production.

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

3. Add a "sessionFactory" bean definition, which depends on the previous "dataSource" bean and the mapping file. The "dialect" property will change based on the database, and the "hibernate.hbm2ddl.auto" property creates the database on-the-fly when the application starts. You might notice that the "dataSource" reference has changed (from previous chapters) to use `<ref bean>` rather than `<ref local>`. This is so you can configure the "dataSource" bean in separate file, possibly switching a testing version with an in-container version.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref bean="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">update</prop>
        </props>
    </property>
</bean>
```

As an alternative to mapping each individual .hbm.xml file, use the "mappingJarLocations" or "mappingDirectoryLocations" properties to refer to a JAR file or directory. You can also use a [hibernate.cfg.xml](#) file to specify your settings and point to it using a "configLocation" property.

4. Add a "transactionManager" bean definition that uses Spring's `HibernateTransactionManager` class. This class's [javadocs](#) explain it best: "This class binds a Hibernate Session from the specified factory to the thread, potentially allowing for one thread Session per factory." [SessionFactoryUtils](#) and [HibernateTemplate](#) are aware of thread-bound Sessions and will participate in such transactions automatically. Using either is required for Hibernate access code that needs to support a transaction handling mechanism.

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

The `UserHibernateDAO` or `UserDAOTest` do not use this bean specifically, but the "userManager" bean's definition references it. *The JDO section* shows you how to utilize transactions in your DAO Tests.

5. Create a `UserDAOHibernate.java` class in `src/org/appfuse/dao/hibernate` (you may need to create this directory/package). This file extends [HibernateDaoSupport](#) and implements `UserDAO`.



NOTE

If you're continuing development on a `MyUsers` application from the previous chapters, change the `saveUser()` method in `UserDAO` to take a `User` (rather than an object) as a parameter.

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

// use your IDE to organize imports

public class UserDAOHibernate extends HibernateDaoSupport implements
UserDAO {

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

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

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

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

6. Add a bean definition for the "userDAO" to `applicationContext-hibernate.xml`.

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

In the `UserDAOHibernate` class, `HibernateTemplate` does most of the work. Using templates to handle persistence calls is a common theme among Spring DAO support classes. Also note the following items in the `UserDAOHibernate.java` class:

- The `getUser()` method uses `HibernateTemplate().get()`, which returns null if it finds no matching objects. The alternative is to use `HibernateTemplate().load()`, which throws an exception if it finds no objects. `HibernateTemplate.load()` is used in the `removeUser()` method, but you could easily use `get()` instead.
- The `getUser()` method throws an `ObjectRetrievalFailureException` when it finds no user.
- It has no checked exceptions. You will likely end up writing a fair amount of try/catch statements with Hibernate.

A `logger` variable is already defined in `HibernateDaoSupport`, allowing for easy logging in subclasses. For example, add the following to the end of the `saveUser()` method.

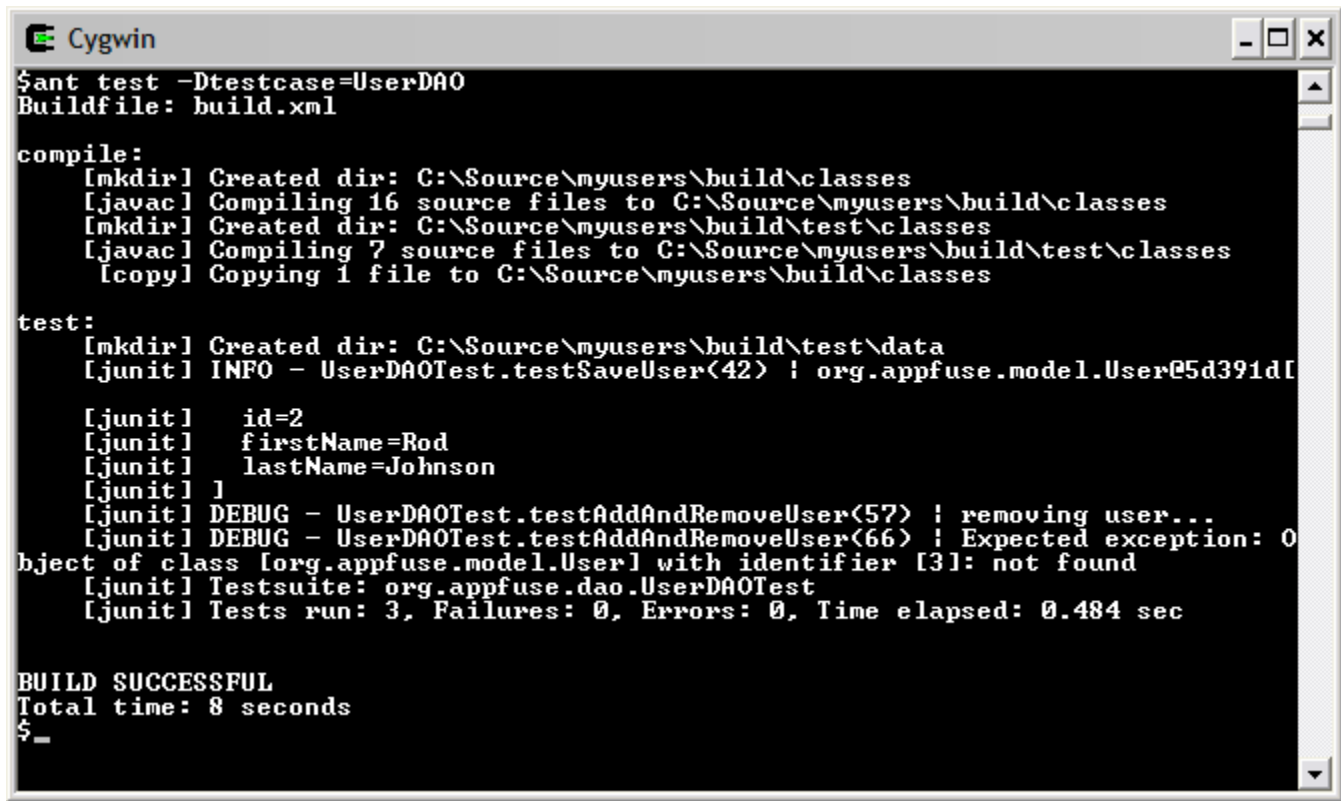
```
if (logger.isDebugEnabled()) {  
    logger.debug("User's id set to: " + user.getId());  
}
```

Test It!

If you're developing from a previous `MyUsers` application, you must make a couple of changes to the `UserDAOTest.testGetUsers()` method before you can successfully run it. The new method is below. It removes assertions that checked for an empty database table before the test and a single record after the test. The reason for this is that you set the `"hibernate.hbm2ddl.auto"` property (on the `"sessionFactory"` bean) to `update`. This changes Hibernate's behavior and updates the schema when the JVM starts (rather than creating it each time). The primary motivation for this is other frameworks don't easily create/delete tables, and you want to have a consistent unit test.

```
public void testGetUsers() {  
    // add a record to the database so we have something to work with  
    user = new User();  
    user.setFirstName("Rod");  
    user.setLastName("Johnson");  
    dao.saveUser(user);  
  
    List users = dao.getUsers();  
    assertTrue(users.size() >= 1);  
    assertTrue(users.contains(user));  
}
```

Run `ant test -Dtestcase=UserDAO`. Your results should be similar to Figure 7.2.



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

compile:
[mkdir] Created dir: C:\Source\myusers\build\classes
[javac] Compiling 16 source files to C:\Source\myusers\build\classes
[mkdir] Created dir: C:\Source\myusers\build\test\classes
[javac] Compiling 7 source files to C:\Source\myusers\build\test\classes
[copy] Copying 1 file to C:\Source\myusers\build\classes

test:
[mkdir] Created dir: C:\Source\myusers\build\test\data
[junit] INFO - UserDAOTest.testSaveUser(42) ! org.appfuse.model.User@5d391d[
[junit]     id=2
[junit]     firstName=Rod
[junit]     lastName=Johnson
[junit] ]
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(57) ! removing user...
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(66) ! Expected exception: 0
bject of class [org.appfuse.model.User] with identifier [3]: not found
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.484 sec

BUILD SUCCESSFUL
Total time: 8 seconds
$_
```

Figure 7.2: Results of the `ant test -Dtestcase=UserDAO` test

MySQL Configuration

Switching from HSQL to MySQL is quite easy. Thanks to Spring, it's just a matter of configuration settings.

1. Make sure [MySQL](#) is installed and running. If you're using MyUsers from a previous chapter, install the additional JARs for this chapter or download MySQL's [JDBC Driver](#) and copy it to *web/WEB-INF/lib*.
2. In *applicationContext-hibernate.xml*, change the "dataSource" bean's properties to the following:

```
<property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
</property>
<property name="url">
    <value>jdbc:mysql://localhost/myusers</value>
</property>
<property name="username"><value>root</value></property>
<property name="password"><value></value></property>
```

A username of `root` and empty password are the default installation settings. You may need to adjust them for your installation.



TIP

You can use a [PropertyPlaceholderConfigurer](#) bean to set the above property values from a properties file. This is covered in [Chapter 8](#).

3. Change the "sessionFactory" bean's "hibernate.dialect" property to MySQL:

```
<prop key="hibernate.dialect">
    net.sf.hibernate.dialect.MySQLDialect
</prop>
```

4. Create the myusers database before running unit tests or starting Tomcat:

```
mysqladmin -u root -p create myusers
```

Optionally, change the `driver`, `url` and `userid` properties in the "browser" target of *build.xml* if you want to use that target with MySQL.

Run `ant test -Dtestcase=UserDAO`. The results should be the same as in Figure 7.2.

Caching

A powerful feature in persistence frameworks is the ability to *cache* data and avoid constant trips to the database. The Hibernate `Session` object is a transaction-level cache of persistent data, but it doesn't handle per-class or collection-by-collection caching at the JVM or cluster level. However, it does support plugging in both JVM or clustered caches (also called *second level* caches). For a complete list of supported caches, see [Hibernate's Second Level Cache documentation](#).

The following example shows how to configure the EHCACHE for JVM-level caching.



NOTE

EHCACHE is the default cache, so you don't need to configure a `hibernate.cache.provider_class` setting in `applicationContext-hibernate.xml`.

1. The simplest way to enable caching for an object is to add a `<cache>` tag to its mapping file. To do this with the `User` object, add a `<cache>` element to the `User.hbm.xml` file in `src/org/appfuse/model`. Optional values are "read-write" and "read-only". You should use the second option only if you're referring to an object or table that rarely changes.

```
<class name="org.appfuse.model.User" table="app_user">
  <cache usage="read-write"/>
  <id name="id" column="id" unsaved-value="0">
```

2. (Optional) Create settings in EHCACHE's configuration file for this class. Create an `ehcache.xml` file in `web/WEB-INF/classes` and fill it with the following XML:

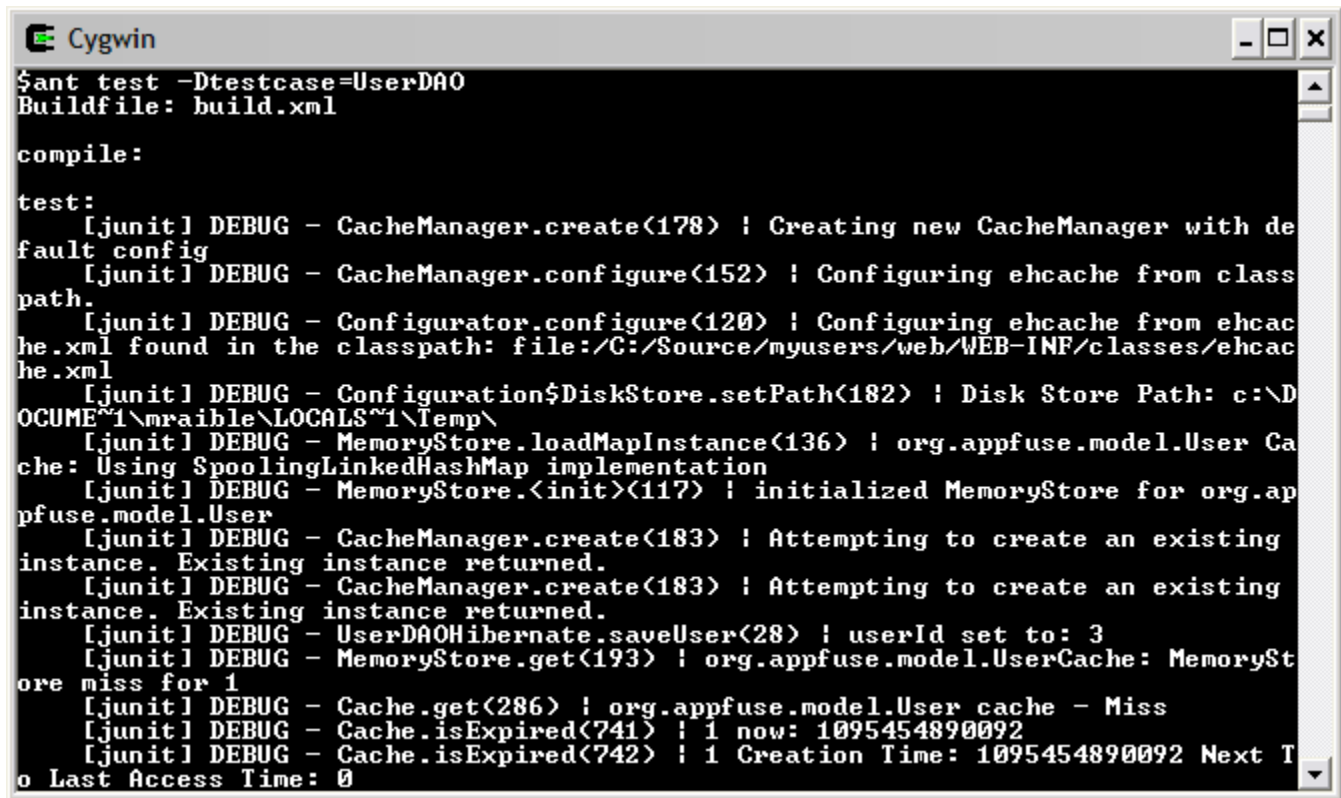
```
<ehcache>
  <!-- Only needed if overflowToDisk="true" -->
  <diskStore path="java.io.tmpdir"/>
  <!-- Required element -->
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>

  <!-- Cache settings per class -->
  <cache name="org.appfuse.model.User"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"/>
</ehcache>
```

3. To prove that your User object is being cached, turn on debug logging for EHCACHE in *web/WEB-INF/classes/log4j.xml*:

```
<logger name="net.sf.ehcache">
  <level value="DEBUG"/>
</logger>
```

4. Run `ant test -Dtestcase=UserDAO`. Your results should be similar to Figure 7.3.



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

compile:
test:
[junit] DEBUG - CacheManager.create(178) ! Creating new CacheManager with de
fault config
[junit] DEBUG - CacheManager.configure(152) ! Configuring ehcache from class
path.
[junit] DEBUG - Configurator.configure(120) ! Configuring ehcache from ehcac
he.xml found in the classpath: file:/C:/Source/myusers/web/WEB-INF/classes/ehcac
he.xml
[junit] DEBUG - Configuration$DiskStore.setPath(182) ! Disk Store Path: c:\D
OCUME~1\mraible\LOCALS~1\Temp\
[junit] DEBUG - MemoryStore.loadMapInstance(136) ! org.appfuse.model.User Ca
che: Using SpoolingLinkedHashMap implementation
[junit] DEBUG - MemoryStore.<init>(117) ! initialized MemoryStore for org.ap
pfuse.model.User
[junit] DEBUG - CacheManager.create(183) ! Attempting to create an existing
instance. Existing instance returned.
[junit] DEBUG - CacheManager.create(183) ! Attempting to create an existing
instance. Existing instance returned.
[junit] DEBUG - UserDAOHibernate.saveUser(28) ! userId set to: 3
[junit] DEBUG - MemoryStore.get(193) ! org.appfuse.model.UserCache: MemorySt
ore miss for 1
[junit] DEBUG - Cache.get(286) ! org.appfuse.model.User cache - Miss
[junit] DEBUG - Cache.isExpired(741) ! 1 now: 1095454890092
[junit] DEBUG - Cache.isExpired(742) ! 1 Creation Time: 1095454890092 Next T
o Last Access Time: 0
```

Figure 7.3: Results of the `ant test -Dtestcase=UserDAO` test

The Hibernate reference documentation has more information on [using and configuring Second Level Caches](#). In general, caching is something that you shouldn't configure for your application until you've tuned your database (that is, with indexes). This cache implementation is for demonstration purposes only.

Lazy-Loading Dependent Objects

One of Hibernate's many features is the ability to *lazy-load* dependent objects. For example, if a list of users refers to a collection of roles objects, you probably don't need the roles loaded to display the list of users. By marking the roles collection with `lazy-load="true"`, they won't be loaded until you try to do something with them (usually in the UI).

To use this feature with Spring, configure the [OpenSessionInViewFilter](#) in your application. This will open a session when a particular URL is first requested, and close it when the page finishes loading. To enable this feature, add the XML below to *web.xml*:

```
<filter>
  <filter-name>hibernateFilter</filter-name>
  <filter-class>org.springframework.orm.hibernate.support.OpenSession
    InViewFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>hibernateFilter</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>
```

Using this feature may cause the following error when running DAO unit tests:

```
[junit] net.sf.hibernate.LazyInitializationException: Failed to lazily
initialize a collection - no session or session was closed
```

To fix this, add the following code to the `setUp()` and `tearDown()` methods of your test.

```
protected void setUp() throws Exception {
    // the following is necessary for lazy loading
    sf = (SessionFactory) ctx.getBean("sessionFactory");
    // open and bind the session for this test thread.
    Session s = sf.openSession();
    TransactionSynchronizationManager
        .bindResource(sf, new SessionHolder(s));

    // setup code here
}

protected void tearDown() throws Exception {
    // unbind and close the session.
    SessionHolder holder = (SessionHolder)
        TransactionSynchronizationManager.getResource(sf);
    Session s = holder.getSession();
    s.flush();
    TransactionSynchronizationManager.unbindResource(sf);
    SessionFactoryUtils.closeSessionIfNecessary(s, sf);

    // teardown code here
}
```

While using this open-session-in-view pattern is a well-known Hibernate session handling idiom, some Spring developers advocate initializing all necessary data in the service layer. Loading everything at once is more efficient and guarantees the same data for all clients.

Other tips and tricks for Hibernate are available on the [Hibernate website](#).

Community and Support

The Hibernate community is a vibrant one. In addition to having [good documentation](#) and well-supported [user forums](#), it is very popular with tens of thousands of developers. It releases early and often, and downloads average 30,000 per release.

The Hibernate website has a list of [Who Uses Hibernate](#). [Commercial Support and Training](#) are available. Additionally, Gavin King and Christian Bauer recently authored [Hibernate in Action](#). I recommend this book for learning Hibernate.

iBATIS

[iBATIS SQL Maps](#) is an open-source persistence framework that allows you to use your model objects with a relational database. In contrast to Hibernate, you write SQL, much like you would with JDBC. You do this in a very simple XML file, allowing abstraction of SQL from Java classes. iBATIS is not an O/R Mapper (ORM). Rather, it is a *Data Mapper*. In Martin's Fowler's [Patterns of Enterprise Application Architecture](#), he describes two patterns: [Data Mapper](#) and [Metadata Mapping](#). The difference is that ORMs (Metadata Mappers) map classes to tables; iBATIS (Data Mapper) maps inputs and outputs to an interface (for example, SQL interface to an RDBMS). An ORM solution works well when you have control of your database. A Data Mapper like iBATIS works well when the database is heavily normalized and you need to pull from several tables to populate an object.

iBATIS is the name of a open-source project started by Clinton Begin in 2001. Clinton had a few products, but none of them gained much recognition until the .NET Pet Shop was released, claiming that .NET was superior to Java in developer productivity and performance. Microsoft published a paper claiming that .NET's version of Sun's PetStore was [10 times faster and 4 times more productive](#). Knowing this wasn't the case, Clinton responded with [JPetStore 1.0](#) in July 2002. Not only did this application have fewer lines of code and a better design than its .NET counterpart, but Clinton implemented it over a few weeks in his spare time!

Clinton's goals while writing JPetStore were to argue the points of 1) Good Design, 2) Code Quality and 3) Productivity. The original .NET Pet Shop had a horrible design with much of the business logic contained in stored procedures, whereas JPetStore had a clean and efficient persistence framework.

This framework quickly drew the attention of the open-source community. Today, iBATIS refers to the "iBATIS Database Layer," which consists of a DAO framework and a SQL Map framework. Spring supports the iBATIS SQL Maps by providing helper classes to easily configure and use them. Furthermore, the Spring project includes JPetStore as one of its sample applications, rewriting many of its pieces to use Spring features.

In my opinion, iBATIS is a "sleeping project" in the open-source community. Not many folks know about it, but those who do, really like it. Perhaps the Spring supporting classes will boost its popularity.

iBATIS's license is [Apache](#), which means you can use it freely as long as your end-user documentation states that your product contains software developed by the Apache Software Foundation. You can modify the code, but then you can no longer distribute it under the Apache name without permission.



NOTE

iBATIS [recently applied](#) to become an Apache top-level project. If accepted, it will likely be renamed iBATIS Data Mapper.

iBATIS is an excellent persistence framework to use with existing or legacy databases. You can easily migrate a JDBC-based application to iBATIS (most of the work involves extracting the SQL out of Java classes and into Java files). Not only is iBATIS fast and efficient, but it doesn't hide SQL, which is one of the most powerful and oldest languages today. Using iBATIS's *SQL Maps*, developers write SQL in XML files and populates objects based on the results of those queries. Much like the Spring/Hibernate combination, iBATIS DAOs require very few lines of code in each method.

In my experience, I've found the following qualities to be true of iBATIS:

- Easy to learn
- Queries are extremely efficient
- Easy transition because of pre-existing SQL
- Just as fast (if not faster) than Hibernate
- Writing iBATIS DAOs is similar to writing Hibernate DAOs



NOTE

iBATIS's [SqlMapClient](#) is similar to Hibernate's *Session* and JDBC's *Connection*. Spring's iBATIS support classes are located in the [org.springframework.orm.ibatis](#) and [org.springframework.orm.ibatis.support](#) packages.

Dependencies

For J2SE 1.4, iBATIS has only one third-party dependency: Commons Logging. For J2SE 1.3, see the *jar-dependencies.txt* file included in [the download](#). Below are the JARs included in the MyUsers download as part of iBATIS 2.0.5. Spring supports both iBATIS 1.x and 2.x; the main difference is the word "Client" on the 2.x classes. This section only covers 2.x.

- **ibatis-common-2.jar** – common classes
- **ibatis-sqlmap-2.jar** - required for SQL Maps 2.0
- **commons-logging.jar** – Logging framework that provides ultra-thin bridge between different logging libraries
- (optional) **cglib-full-2.0.2.jar** – Code Generation Library that enables runtime byte-code enhancement for optimized JavaBean property access and enhanced lazy loading.

Configuration

To integrate iBatis you must create a SQL Map for the `User` object. A SQL Map is an XML file that contains SQL statements to *map* a query's inputs and outputs to objects. In order to avoid conflicts among bean names, rename the `applicationContext-hibernate.xml` file (in `web/WEB-INF`) to `applicationContext-hibernate.xml.txt`. This will prevent Spring from loading it.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="UserSQL">
    <insert id="addUser" parameterClass="org.appfuse.model.User">
        insert into app_user (id, first_name, last_name)
        values (#id#, #firstName#, #lastName#);
        <selectKey resultClass="java.lang.Long" keyProperty="id" >
            select last_insert_id();
        </selectKey>
    </insert>

    <update id="updateUser" parameterClass="org.appfuse.model.User">
        update app_user set first_name = #firstName#,
        last_name = #lastName#
        where id = #id#;
    </update>

    <select id="getUser" parameterClass="java.lang.Long"
        resultClass="org.appfuse.model.User">
        select id, first_name as firstName, last_name as lastName
        from app_user where id=#id#;
    </select>

    <select id="getUsers" resultClass="org.appfuse.model.User">
        select id, first_name as firstName, last_name as lastName
        from app_user;
    </select>

    <delete id="deleteUser" parameterClass="java.lang.Long">
        delete from app_user where id = #id#;
    </delete>
</sqlMap>
```


In this file, you can see that different elements (<insert>, <update>, <select>, <delete>) indicate database operations. Note that each element can optionally specify a `parameterClass` or `resultClass` attribute. Dynamic variables are enclosed in `#value#` and indicate properties in the `parameterClass`.

2. Create a `sql-maps-config.xml` file in `src/org/appfuse/dao/ibatis` (you must create this directory) that indicates the location of the `UserSQL.xml` SQL Map.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC
    "-//iBatis.com//DTD SQL Map Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
    <settings enhancementEnabled="true" maxTransactions="5"
        maxRequests="32" maxSessions="10"/>

    <sqlMap resource="org/appfuse/dao/ibatis/UserSQL.xml" />
</sqlMapConfig>
```

See [iBatis's Documentation](#) (PDF) for more information on the <settings> element. The values shown here should be sufficient for most applications.

3. Create an `applicationContext-ibatis.xml` file (using `applicationContext-empty.txt` as a template) in the `web/WEB-INF` directory and add a bean definition for a `javax.sql.DataSource`. The "dataSource" bean in this example uses the MySQL database you configured in the last section.

```
<?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>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost/myusers</value>
        </property>
        <property name="username"><value>root</value></property>
        <property name="password"><value></value></property>
    </bean>
    <!-- Add bean definitions here -->
</beans>
```

4. Create a "sqlMapClient" bean definition to integrate iBatis with Spring.

```
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation">
    <value>classpath:/org/appfuse/dao/ibatis/sql-map-config.xml
  </value>
  </property>
</bean>
```

5. Add a "transactionManager" bean definition that uses a [DataSourceTransactionManager](#). This [PlatformTransactionManager](#) implementation binds a JDBC connection from the specified DataSource to the thread, potentially allowing for one thread connection per data source.



NOTE

The primary reason you need a bean named "transactionManager" is that the "userManager" bean references a bean with that name.

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.
      DataSourceTransactionManager">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
```

6. Create a UserDAOiBatis.java class in *src/org/appfuse/dao/ibatis*. This file extends [SqlMapClientDaoSupport](#) and implements UserDAO.

```
package org.appfuse.dao.ibatis;

// use your IDE to organize imports

public class UserDAOiBatis extends SqlMapClientDaoSupport
    implements UserDAO {

    public List getUsers() {
        return getSqlMapClientTemplate()
            .queryForList("getUsers", null);
    }

    public User getUser(Long id) {
        User user = (User) getSqlMapClientTemplate()
            .queryForObject("getUser", id);

        if (user == null) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
    }
}
```

```

        return user;
    }

    public void saveUser(User user) {
        if (user.getId() == null) {
            // Use iBatis's <selectKey> feature, which is db-specific
            Long id = (Long) getSqlMapClientTemplate()
                .insert("addUser", user);
            user.setId(id);
            logger.info("new User id set to: " + id);
        } else {
            getSqlMapClientTemplate().update("updateUser", user);
        }
    }

    public void removeUser(Long id) {
        getSqlMapClientTemplate().update("deleteUser", id);
    }
}

```

7. Add a "userDAO" bean definition to *applicationContext-ibatis.xml*. The example below uses the `autowire` attribute rather than specifying `dataSource` and `sqlMapClient` properties. When using `autowire`, it's a good idea to specify the injected dependencies in a comment.

```

<bean id="userDAO" autowire="byName"
    class="org.appfuse.dao.ibatis.UserDAOiBatis"/>
    <!-- injected properties: dataSource, sqlMapClient -->

```

8. The `app_user` table created by Hibernate does not allow nulls in the `id` column. With iBATIS, it's easiest to insert nulls for primary keys and retrieve the generated id back from the database. To do this, drop and recreate the `app_user` table.
 - a. Login to MySQL by typing `mysql -u root -p myusers` from the command line.
 - b. Execute `drop table app_user;`.
 - c. Execute the following SQL statement:

```

create table app_user (id bigint not null auto_increment,
    first_name varchar(50), last_name varchar(50),
    primary key (id));

```

Test It!

If you're developing MyUsers from a previous application, you may need to change *build.xml* before the tests will pass. The `compile` target has the following XML that copies Hibernate mapping files to the build directory.

```
<!-- Copy hibernate mapping files to ${build.dir}/classes -->
<copy todir="${build.dir}/classes">
  <fileset dir="${src.dir}" includes="**/*.hbm.xml"/>
</copy>
```

Change this to the following:

```
<!-- Copy XML files to ${build.dir}/classes -->
<copy todir="${build.dir}/classes">
  <fileset dir="${src.dir}" includes="**/*.xml"/>
</copy>
```

Run `ant clean test -Dtestcase=UserDAO`. The output from this test should be similar to the `UserDAOHibernate` class.

Caching

iBATIS supports many caching strategies for SQL Maps. To add a caching strategy to the *UserSQL.xml* file, add the following `<cacheModel>` element:

```
<sqlMap namespace="UserSQL">
  <cacheModel id="userCache" type="LRU">
    <flushInterval hours="24"/>
    <property name="size" value="1000"/>
  </cacheModel>
  <insert id="addUser" parameterClass="org.appfuse.model.User">
```

Add a `cacheModel` attribute to any `<select>` statements. For example, add it to the `"getUser"` statement:

```
<select id="getUser" parameterClass="java.lang.Long"
  resultClass="org.appfuse.model.User"
  cacheModel="userCache">
  select id, first_name as firstName, last_name as lastName
  from app_user where id=#id#;
</select>
```

Retrieving Generated Primary Keys

The "addUser" statement uses a database-specific means to retrieve the generated primary key. The `<selectKey>` element allows you to retrieve generated primary keys quite easily.

```
<insert id="addUser" parameterClass="org.appfuse.model.User">
    insert into app_user (id, first_name, last_name)
    values (#id#, #firstName#, #lastName#);
    <selectKey resultClass="java.lang.Long" keyProperty="id" >
        select last_insert_id();
    </selectKey>
</insert>
```

The problem with this approach is that it's database-specific. Using `select last_insert_id()` will only work with MySQL. This shows how the SQL standard allows variations and is not a *true* standard. A good ORM tool like Hibernate or JDO does this transparently for you.



NOTE

If you'd like to try switching back to HSQLDB at this point, simply change the `<selectKey>` statement to call `identity()`. Also change the `dataSource` bean appropriately. Run `ant browse` and drop/recreate the `app_user` table. The following SQL script will help you with the table:

```
create table app_user (id integer identity, first_name varchar(50),
last_name varchar(50));
```

As an alternative, use a Spring class to generate the primary key for you. For example, to use the [MySQLMaxValueIncrementer](#), perform the following steps:

1. Comment out the `<selectKey>` element in *UserSQL.xml*.
2. Change the `save()` method to the following:

```
public void saveUser(User user) {
    if (user.getId() == null) {
        MySQLMaxValueIncrementer incrementer =
            new MySQLMaxValueIncrementer(
                getDataSource(), "user_sequence", "value");
        Long id = new Long(incrementer.nextLongValue());
        user.setId(id);

        getSqlMapClientTemplate().insert("addUser", user);
        logger.info("new User id set to: " + id);
    } else {
        getSqlMapClientTemplate().update("updateUser", user);
    }
}
```

3. Create the `user_sequence` table to allow the retrieval of primary keys.

```
create table user_sequence (value bigint auto_increment,
    primary key (value));
insert into user_sequence values (0);
```

If you clean and run the `UserDAOTest`, all your tests should pass.

The problem with putting the `MySQLMaxValueIncrementer` into your class is now you've hard-coded your DAO to depend on MySQL. Having the `select last_insert_id()` statement seems more configurable since it was in XML. Refactor this class to use dependency injection.

1. Add an `incrementer` variable and setter to the `UserDAOHibernate` class.

```
private DataFieldMaxValueIncrementer incrementer;

public void setIncrementer(DataFieldMaxValueIncrementer incrementer) {
    this.incrementer = incrementer;
}
```

2. Change the `saveUser()` method to use this `incrementer`. Just delete the three lines that initialize the `incrementer`.
3. Configure the `incrementer` as a bean in `web/WEB-INF/applicationContext-ibatis.xml`. Since `userDAO` autowires by name, this property will get injected by Spring.

```
<bean id="incrementer"

class="org.springframework.jdbc.support.incrementer.MySQLMaxValueIncremen
ter">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="incrementerName">
        <value>user_sequence</value>
    </property>
    <property name="columnName"><value>value</value></property>
</bean>
```

4. Run `ant test -Dtestcase=UserDAO`.

Unfortunately, iBATIS doesn't support JDBC 3.0's `getGeneratedKeys()` method, so you must use one of the above methods to generate primary keys. Of course, you can also `select max(id)` and increment the primary key yourself.

Community and Support

Developers who use iBATIS tend to be very happy with it. It has clean and concise [documentation and tutorials](#). The documentation is a mere 53 pages and the tutorial is only 9! In my experience, this is a framework you can learn and use in the same day. iBATIS doesn't have a user mailing list, but it does have [help forums](#). Its application to be a top-level project at Apache is another favorable indicator.

Spring JDBC

If you've written JDBC code before, you know that it can be tedious. Not only do you have to set up Connections, Statements and ResultSets, but you have to close them after you've retrieved your data. Closing resources in JDBC code is an area that many new developers don't know how to do properly. They tend to forget to use the *finally* block, even though it's is a [basic JDBC code pattern](#).

As you've seen in the previous examples, Spring removes the open/close resource responsibility from the developer. It manages these operations for you, allowing you to write application code rather than infrastructure code. Spring's JDBC core is an abstraction on top of J2SE's JDBC that allows you to write a minimal amount of code to retrieve, save and delete your data.

Four packages make up the JDBC abstraction framework: `core`, `datasource`, `object` and `support`. The functionality of each package is below:

- [org.springframework.jdbc.core](#): core classes, `JdbcTemplate` and `JdbcDaoSupport` convenience class
- [org.springframework.jdbc.datasource](#): classes for easy `DataSource` access and basic `DataSource` implementations
- [org.springframework.jdbc.object](#): classes for corresponding to database queries, updates and stored procedures as thread-safe reusable objects
- [org.springframework.jdbc.support](#): `SQLException` translation, `DataFieldMaxValueIncrementer` implementations and support classes for `jdbc.core` and `jdbc.object`

In this section, you will use [JdbcTemplate](#) and [JdbcDaoSupport](#). The names are similar to the `Template` and `DaoSupport` classes you used with Hibernate and iBATIS. This section will not cover `SQLException` translation because you rarely have to do it yourself. The classes are used internally and a number of default translations are built-in for the following databases: DB2, HSQL, SQL Server, MySQL, Oracle, Informix, PostgreSQL and Sybase. You can find a complete list of error code to exception mappings in the [jdbc/support/sql-error-codes.xml](#) file.

Mapping ResultsSets to Objects

Similar to Hibernate and iBATIS, Spring JDBC requires you to map results from a query to your POJOs. With Hibernate and iBATIS, you did this using XML. With Spring JDBC, you must do this programmatically. You don't necessarily *have* to map results to objects; you can simply return maps of your data. For instance, the following code returns a map with each entry as a row:

```
public List getList() {
    return getJdbcTemplate().queryForList("select * from tablename");
}
```

In most cases, however, you'll want to map these results to a List of objects. To do this with Spring JDBC, use a [MappingSqlQuery](#) class to convert each row of the JDBC ResultSet to an object. The following is an example of this type of class retrieving a list of users in the MyUsers application:

```
public class UsersQuery extends MappingSqlQuery {
    public UsersQuery(DataSource ds) {
        super(ds, "SELECT * FROM app_user");
        compile();
    }
    protected Object mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        User user = new User();
        user.setId(new Long(rs.getLong("id")));
        user.setFirstName(rs.getString("first_name"));
        user.setLastName(rs.getString("last_name"));
        return user;
    }
}
```

When extending `MappingSqlQuery`, you must override the `mapRow(ResultSet rs, int rowNum)` method. Use the `UsersQuery` class to get a list of `User` objects:

```
public List getUsers() {
    return new UsersQuery(getDataSource()).execute();
}
```

Declare parameters on your query by using the `declareParameter()` method of `MappingSqlQuery`.

```
public UserQuery(DataSource ds) {
    super(ds, "SELECT * FROM app_user WHERE id = ?");
    super.declareParameter(new SqlParameter("id", Types.INTEGER));
    compile();
}
```

Pass in the `id` parameter using an `Object` array:

```
public User getUser(Long id) {
    List users = new UserQuery(getDataSource())
        .execute(new Object[]{id});
    if (users.isEmpty()) {
        throw new ObjectRetrievalFailureException(User.class, id);
    }
    return (User) users.get(0);
}
```

Executing Update Queries

Mapping `ResultSet` to objects helps you retrieve data, but it doesn't help you update it. For that, use a `SqlUpdate` class to create a reusable object for updating rows. Here is an example:

```
public class UserUpdate extends SqlUpdate {
    public UserUpdate(DataSource ds) {
        super(ds, "INSERT INTO app_user (id, first_name, last_name) values"
            "(?, ?, ?)");
        declareParameter(new SqlParameter("id", Types.BIGINT));
        declareParameter(new SqlParameter("first_name", Types.VARCHAR));
        declareParameter(new SqlParameter("last_name", Types.VARCHAR));
        compile();
    }
}
```

Call this class using this example code:

```
Object[] params =
    new Object[] {user.getId(), user.getFirstName(),
        user.getLastName()};
new UserUpdate(getDataSource()).update(params);
```

Another, less verbose option is to use the `update()` method of `JdbcTemplate`:

```
getJdbcTemplate().update(
    "UPDATE app_user SET first_name = ?, last_name = ? WHERE id = ?",
    new Object[] {user.getFirstName(), user.getLastName(), user.getId()});
```

Finally, you don't need to implement a full `SqlUpdate` class; simply declare one in-line within a method:

```
String sql = "INSERT INTO app_user (id, first_name, last_name) ";
    sql += "values (?, ?, ?)";
SqlUpdate su = new SqlUpdate(getDataSource(), sql);
su.declareParameter(new SqlParameter("id", Types.BIGINT));
su.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
su.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
su.compile();

Object[] params = new Object[]
    {user.getId(), user.getFirstName(), user.getLastName()};

su.update(params);
```

Retrieving Generated Primary Keys

JDBC 3.0, which is part of J2SE 1.4, specifies that a JDBC 3.0-compliant driver has to implement the [`java.sql.Statement.getGeneratedKeys\(\)`](#) method. This method is to retrieve generated keys from databases (for example, calling `select last_insert_id();` from MySQL). The MySQL driver in `MyUsers` is JDBC 3.0-compliant.

To take advantage of retrieving generated keys with Spring JDBC, use a [`KeyHolder`](#) as a second parameter to the [`SqlUpdate.update\(\)`](#) method.

```
KeyHolder keys = new GeneratedKeyHolder();
su.update(params, keys);
user.setId(new Long(keys.getKey().longValue()));
```

For non-compliant JDBC Drivers, use the [`DataFieldMaxValueIncrementer`](#) strategy as [described earlier](#).

Dependencies

Since Spring JDBC is part of Spring, it has no third-party dependencies like the other libraries. If you only want to use the JDBC functionality of Spring, use *spring-dao.jar* (rather than the all-encompassing *spring.jar*) in your application.

Configuration

Spring JDBC has no *mapping files* or *sql maps* so it's simple to configure in the MyUsers application.

1. Create an *applicationContext-jdbc.xml* file in the *web/WEB-INF* directory and add a bean definition for "dataSource". Be sure to rename the previous *applicationContext-ibatis.xml* to *applicationContext-ibatis.xml.txt*.



NOTE

You may want to duplicate the iBATIS version of this file and remove the `sqlMapClient` bean.

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost/myusers</value>
  </property>
  <property name="username"><value>root</value></property>
  <property name="password"><value></value></property>
</bean>
```

2. Add a transaction manager:

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
```

3. Create a `UserDAOJdbc.java` class in `src/org/appfuse/dao/jdbc`. This file extends `JdbcDaoSupport` and implements `UserDAO`.

```
package org.appfuse.dao.jdbc;

// use your IDE to organize imports

public class UserDAOJdbc extends JdbcDaoSupport implements UserDAO {

    public List getUsers() {
        return new UsersQuery(getDataSource()).execute();
    }

    public User getUser(Long id) {
        List users = new UsersQuery(getDataSource())
```

```

        .execute(new Object[]{id});
    if (users.isEmpty()) {
        throw new ObjectRetrievalFailureException(User.class, id);
    }
    return (User) users.get(0);
}

public void saveUser(User user) {
    if (user.getId() == null) {
        String sql = "INSERT INTO app_user (id, first_name, ";
        sql += " last_name) values (?, ?, ?)";
        SqlUpdate su = new SqlUpdate(getDataSource(), sql);
        su.declareParameter(new SqlParameter("id", Types.BIGINT));
        su.declareParameter(new SqlParameter("first_name",
Types.VARCHAR));
        su.declareParameter(new SqlParameter("last_name",
Types.VARCHAR));
        su.compile();

        Object[] params = new Object[]
            {user.getId(), user.getFirstName(),
            user.getLastName()};

        KeyHolder keys = new GeneratedKeyHolder();
        su.update(params, keys);
        user.setId(new Long(keys.getKey().longValue()));

        if (logger.isDebugEnabled()) {
            logger.info("user's id is: " + user.getId());
        }
    } else {
        getJdbcTemplate().update("UPDATE app_user SET first_name = ?,
last_name = ? WHERE id = ?",
            new Object[] {user.getFirstName(), user.getLastName(),
            user.getId()});
    }
}

public void removeUser(Long id) {
    getJdbcTemplate().update("DELETE FROM app_user WHERE id = ?",
        new Object[] {id});
}

// Query to get a single User
class UserQuery extends MappingSqlQuery {
    public UserQuery(DataSource ds) {
        super(ds, "SELECT * FROM app_user WHERE id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }
}

```

```

        protected Object mapRow(ResultSet rs, int rowNum)
        throws SQLException {
            User user = new User();
            user.setId(new Long(rs.getLong("id")));
            user.setFirstName(rs.getString("first_name"));
            user.setLastName(rs.getString("last_name"));
            return user;
        }
    }

    // Query to get a list of User objects
    class UsersQuery extends MappingSqlQuery {
        public UsersQuery(DataSource ds) {
            super(ds, "SELECT * FROM app_user");
            compile();
        }
        protected Object mapRow(ResultSet rs, int rowNum)
        throws SQLException {
            User user = new User();
            user.setId(new Long(rs.getLong("id")));
            user.setFirstName(rs.getString("first_name"));
            user.setLastName(rs.getString("last_name"));
            return user;
        }
    }
}

```

This class has a lot more lines of code than the previous two. However, it does not require any configuration files. The `UserQuery` and `UsersQuery` classes are inner classes, but could just as easily be refactored into their own .java files.

4. Add a `userDAO` bean to the `applicationContext-jdbc.xml` file in the `web/WEB-INF` directory.

Test It!

To ensure a clean database, drop and recreate the `app_user` table.

1. Login to MySQL by typing `mysql -u root -p myusers` from the command line.
2. Execute `drop table app_user;.`
3. Execute the following statement:

```

create table app_user (id bigint not null auto_increment,
    first_name varchar(50), last_name varchar(50),
    primary key (id));

```

4. Run `ant test -Dtestcase=UserDAO`.

Community and Support

Spring's JDBC framework is a clean abstraction on top of JDBC. It frees you from opening and closing resources and works great for batch processing large amounts of data. Spring's [reference documentation](#) is a good resource for learning more about it, and Spring's [Data Access Support Forum](#) is a great place to get your questions answered.

Commercial support for Spring and its JDBC Framework is available through [Interface21](#).

JDO

JDO is a Java standard, which means it was developed as part of the [Java Community Process](#). As a standard, it's not an actual product, but a specification of how the product should be built. The goals of JDO are described best in section 1.1 (Overview) of the [JDO 2.0 specification](#).

"There are two major objectives of the JDO architecture: first, to provide application programmers a transparent Java-centric view of persistent information, including enterprise data and locally stored data; and second, to enable pluggable implementations of datastores into application servers."

This section covers how to use Spring to configure and use your JDO implementation, which may be from your app server vendor, or from an open-source provider like [JPOX](#) or [ObjectWeb's Speedo](#). These examples use JPOX because it is the most popular open-source implementation.



NOTE

Spring's JDO support classes are located in the [org.springframework.orm.jdo](#) and [org.springframework.orm.jdo.support](#) packages.

JDO and Primary Keys

JDO handles primary keys (also called *object ids*) very differently than other persistence options. This is because it's designed for both object databases and relational databases. In most cases, you'll be working with a relational database and will therefore want to use *application* identity. However, if you're prototyping an application, it might be easier to use *database* identity and not worry about primary keys. Below is a list of the different identity types in the JDO specification.

JDO Identity Types

Identity Type	JDO-Controlled Schema	Existing Schema
Datastore	#1. Generates tables; adds a primary key and defines its value	#2. Allows you to configure classes to map to existing tables and define the primary key column name
Application	#3. Generates tables; allows you to define classes for primary keys	#4. Maps to existing tables; allows you to define classes for primary keys

The purpose of the *datastore* identity type is to let the JDO completely handle the primary key. For example, to use the `User` object with type #1, remove the `id` field and its access method, as JDO does not use them. This creates its own column (`user_id`) in the database and its own table (`user`). The assumption is that you don't need to know the primary key in your action.

With type #2, you can define a column name that matches the `id` property in `User`, but the database won't populate it when you persist the object. In other words, it won't tell you what the generated primary key is.

If you want to know the primary key, you must use the *application* identity type. Using this option, you can still have your tables generated for you (type #3), or you can use `MetaData` (which is really a mapping file similar to Hibernate's) to specify a table name (type #4). To use application identity, you must specify a primary key class. In `MyUsers`, the `UserDAOTest`'s `testSaveUser()` method verifies that the primary key was assigned.

```
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);
}
```

Because of this, the JDO implementation of `UserDAO` will use *application* identity.

Dependencies

This example uses JPOX version 1.1.0-alpha-2, which is an implementation of JDO 2.0. Below are the JARs included in the `MyUsers` download as part of JPOX.

- **bcel-5.1.jar:** Byte-Code Engineering Library enhances the classes with byte-code
- **jpox-enhancer-1.1.0-alpha-2.jar:** JPOX byte-code Enhancer enhances classes before running a JDO-enabled application with JPOX
- **jpox-1.1.0-alpha-2.jar:** Core JPOX classes

Configuration

JDO requires a little more setup than the other frameworks. You must add an extra step to the build process that *enhances* your persistable objects. To avoid conflicts with the previous section on iBATIS, rename the *applicationContext-jdbc.xml* file (in *web/WEB-INF*) to *applicationContext-jdbc.xml.txt*. This will prevent it from being loaded.



NOTE

The JPOX implementation for this example is an alpha release of JDO 2.0. The 2.0 specification has an early draft available and will soon have a community review. After that, you a lot more vendors will be implementing JDO 2.0 solutions. The *enhancement* step described here will likely be replaced by a byte-code enhancing solution like [cglib](#).

1. Open the *build.xml* file in MyUsers and add the following "enhance" target just before the "test" target.

```
<target name="enhance" description="JPOX enhancement"
  depends="compile">
  <taskdef name="jpoxenhancer"
    classname="org.jpox.enhancer.tools.EnhancerTask"
    classpathref="classpath"/>

  <jpoxenhancer dir="${build.dir}/classes" failonerror="true"
    fork="true" verbose="true" check="true">
    <classpath>
      <path refid="classpath"/>
      <path location="${build.dir}/classes"/>
      <path location="web/WEB-INF/classes"/>
    </classpath>
  </jpoxenhancer>
</target>
```

Change the "test" and "deploy" targets to depend on "enhance".

```
<target name="test" depends="enhance"
  description="Runs JUnit tests">
  ...
<target name="war" depends="enhance"
  description="Packages app as WAR">
```

2. Create a `MetaData` file to map the `User` object to the `app_user` table. In the `org/appfuse/model` directory, create a file named `package.jdo` with the following contents:

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="org.appfuse.model">
        <class name="User" identity-type="application"
            objectid-class="org.appfuse.model.PrimaryKey"
            table="app_user">
            <field name="id" primary-key="true"/>
            <field name="firstName" persistence-modifier="persistent">
                <column name="first_name" jdbc-type="VARCHAR" length="50"/>
            </field>
            <field name="lastName" persistence-modifier="persistent">
                <column name="last_name" jdbc-type="VARCHAR" length="50"/>
            </field>
            <extension vendor-name="jpo" key="use-poid-generator"
value="true"/>
        </class>
    </package>
</jdo>
```

Most of the items in this class are self-explanatory. You can see where the table name, columns and sizes are set, and how each field is defined as a column in the `User` class. Note that the `package.jdo` file can hold metadata for all the classes in the `org.appfuse.model` package (defined by the `<package>` element). You may also notice the following line towards the bottom:

```
<extension vendor-name="jpo" key="use-poid-generator" value="true"/>
```

This is an indicator to use a sequence table for generating primary keys. There are many ways to control how primary keys are generated. For more information, please see [JPOX's documentation](#).



NOTE

When JPOX has fully implemented JDO 2.0, this extension will likely be replaced.

3. In order to copy and package the `package.jdo` file with the war, modify the `"compile"` target in `build.xml` to include `*.jdo` files:

```
<!-- Copy XML files to ${build.dir}/classes -->
<copy todir="${build.dir}/classes">
    <fileset dir="${src.dir}" includes="**/*.xml"/>
    <!-- Copy JDO mapping files -->
    <fileset dir="${src.dir}" includes="**/*.jdo"/>
</copy>
```

4. The `<class>` element of the *package.jdo* file has an `objectid-class` attribute. This specifies a primary key class and is required with an *application* identity. Create a `PrimaryKey.java` class in *src/org/appfuse/model* and populate it with the code below:

```
package org.appfuse.model;

// organize imports with your IDE

public class PrimaryKey extends BaseObject implements Serializable {
    public Long id;

    public PrimaryKey() {}

    public PrimaryKey(String value) {
        id = Long.valueOf(value);
    }
}
```

A primary key class has [many requirements](#) as defined by the JDO spec. They are as follows:

- The class must have a public no-args constructor.
- The class must be serializable.
- The class must provide a String constructor to return an instance that is equal to an instance returned by the `toString()` method.
- The class must implement `equals()`, `hashCode()` and `toString()` methods. In the previous `PrimaryKey` class, this is handled via reflection in the `BaseObject` class.

5. Run `ant clean enhance` to see the metadata enhance the `org.appfuse.model.User` class.

```

Cygwin
$ant clean enhance
Buildfile: build.xml

clean:
    [delete] Deleting directory C:\Source\myusers\build

compile:
    [mkdir] Created dir: C:\Source\myusers\build\classes
    [javac] Compiling 20 source files to C:\Source\myusers\build\classes
    [mkdir] Created dir: C:\Source\myusers\build\test\classes
    [javac] Compiling 7 source files to C:\Source\myusers\build\test\classes
    [copy] Copying 4 files to C:\Source\myusers\build\classes

enhance:
    [jpxenhancer] JPOX Enhancer : Processing JDO Meta-Data C:\Source\myusers\build\
classes\org\appfuse\model\package.jdo
    [jpxenhancer] JPOX Enhancer : Processing class org.appfuse.model.User
    [jpxenhancer] ENHANCED: org.appfuse.model.User

BUILD SUCCESSFUL
Total time: 10 seconds
$

```

Figure 7.4: Results of the `ant clean enhance` test

6. Create an `applicationContext-jdo.xml` file (using `applicationContext-empty.txt` as a template) in `web/WEB-INF`. Add the following "persistenceManagerFactory" bean definition that uses the [LocalPersistenceManagerFactoryBean](#).

```

<bean id="persistenceManagerFactory"
class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="jdoProperties">
        <props>
            <prop key="javax.jdo.PersistenceManagerFactoryClass">
                org.jpox.PersistenceManagerFactoryImpl
            </prop>
            <prop key="javax.jdo.option.ConnectionDriverName">
                com.mysql.jdbc.Driver
            </prop>
            <prop key="javax.jdo.option.ConnectionUserName">root</prop>
            <prop key="javax.jdo.option.ConnectionPassword"></prop>
            <prop key="javax.jdo.option.ConnectionURL">
                jdbc:mysql://localhost/myusers
            </prop>
            <prop key="org.jpox.autoCreateSchema">true</prop>
            <prop key="org.jpox.validateTables">false</prop>
            <prop key="javax.jdo.option.NontransactionalRead">

```

```

        true
    </prop>
</props>
</property>
</bean>

```

7. Add a "transactionManager" bean definition that uses a [JdoTransactionManager](#). This binds a JDO PersistenceManager from the specified factory to the thread, potentially allowing for one thread PersistenceManager per factory.

```

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

```

8. Create a `UserDAOJdo.java` class in `src/org/appfuse/dao/jdo` (you must create this directory). This class extends [JdoDaoSupport](#) and implements `UserDAO`.

```

package org.appfuse.dao.jdo;

// user your IDE to organize imports

public class UserDAOJdo extends JdoDaoSupport implements UserDAO {

    public List getUsers() {
        Collection users = getJdoTemplate().find(User.class);
        users = getPersistenceManager().detachCopyAll(users);
        return new ArrayList(users);
    }

    public User getUser(Long id) {
        User user = (User) getJdoTemplate().getObjectById(User.class, id);
        if (user == null) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
        return (User) getPersistenceManager().detachCopy(user);
    }

    public void saveUser(User user) {
        if (user.getId() == null) {
            getJdoTemplate().makePersistent(user);
        } else {
            getPersistenceManager().attachCopy(user, true);
        }
    }
}

```

```

    public void removeUser(Long id) {
        getJdoTemplate().deletePersistent(
            getJdoTemplate().getObjectById(User.class, id));
    }
}

```

9. Add a "userDAO" bean definition to *applicationContext-jdo.xml*.

```

<bean id="userDAO" class="org.appfuse.dao.jdo.UserDAOJdo">
    <property name="persistenceManagerFactory">
        <ref bean="persistenceManagerFactory"/>
    </property>
</bean>

```

Test It!

Testing this class isn't as simple as previous examples. JDO requires a transaction for any write operations, unless you specifically indicate that it doesn't need one. If you run `ant test -Dtestcase=UserDAO`, you will see a stack trace in your console complaining that a transaction isn't active.

```

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

compile:
enhance:
[jpoxenhancer] JPOX Enhancer : Processing JDO Meta-Data C:\Source\myusers\build\
classes\org\appfuse\model\package.jdo
[jpoxenhancer] JPOX Enhancer : Processing class org.appfuse.model.User
[jpoxenhancer] ERROR - GeneratorBase.enhance(1409) : Class org.appfuse.model.Use
r is already enhanced.
[jpoxenhancer] WARN - GeneratorBase.update(1516) : class not updated : org.appfu
se.model.User
[jpoxenhancer] ENHANCED: org.appfuse.model.User

test:
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 3, Time elapsed: 1.391 sec

[junit] Testcase: testGetUsers(org.appfuse.dao.UserDAOTest): Caused a
n ERROR
[junit] Transaction is not active; nested exception is org.jpox.exceptions.T
ransactionNotActiveException: Transaction is not active
[junit] org.springframework.orm.jdo.JdoUsageException: Transaction is not ac
tive; nested exception is org.jpox.exceptions.TransactionNotActiveException: Tra
nsaction is not active
[junit] org.jpox.exceptions.TransactionNotActiveException: Transaction is no
t active

```

Figure 7.5: Stack trace, result of the `ant test -Dtestcase=UserDAO` test

The JDO spec says you can turn this off using by setting `javax.jdo.option.NontransactionalWrite` to `true`, but JPOX doesn't support this (yet). In reality, you *do* want the save operation to participate in a transaction. If you don't have an active transaction, you must create a new one. That's why the `userManager` bean has a `PROPOGATION_REQUIRED` transaction attribute on its `save*` methods.

In the normal course of the application, the save methods in the `UserManager` will participate in a transaction, so the only issue is in the `UserDAOTest`. To prove the issue is with this class and not the `UserDAOJdo`, run **ant test -Dtestcase=UserManager**.

The easiest way to fix this is to override the definition of `userDAO` and wrap it with declarative transactions.

1. Create a file named *applicationContext-test.xml* in *test/org/appfuse/dao*. Put the following XML into this file:

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

<beans>
    <!-- userDAO for testing UserDAOJdo -->
    <bean id="userDAO"

class="org.springframework.transaction.interceptor.TransactionProxyFactor
yBean">
        <property name="transactionManager">
            <ref bean="transactionManager"/>
        </property>
        <property name="target">
            <bean class="org.appfuse.dao.jdo.UserDAOJdo"
                autowire="byName"/>
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
</beans>
```

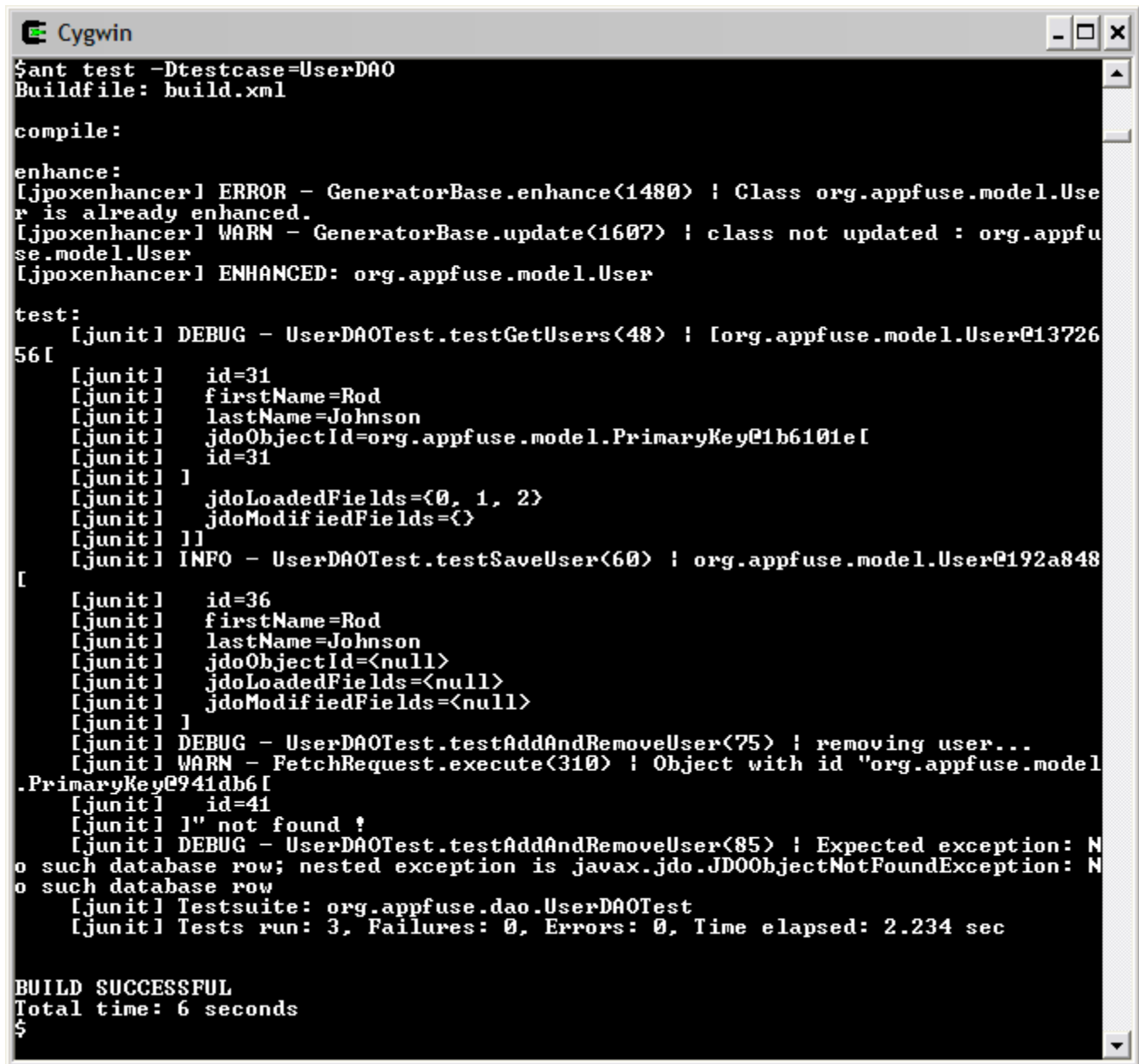

2. Change the `setUp()` method in `UserDAOTest` to load this file before grabbing the `userDAO`.

```
protected void setUp() throws Exception {
    String[] paths =
        {"/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

3. Improve the "compile" target in *build.xml* to copy this file into the test classpath.

```
<!-- Copy JDO mapping files -->
<fileset dir="${src.dir}" includes="**/*.jdo"/>
</copy>
<!-- Copy overriding test files -->
<copy todir="${test.dir}/classes">
<fileset dir="${test.src}" includes="**/*.xml"/>
</copy>
</target>
```

4. Run `ant test -Dtestcase=UserDAO`.



```

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

compile:

enhance:
[jpoxenhancer] ERROR - GeneratorBase.enhance(1480) : Class org.appfuse.model.User
r is already enhanced.
[jpoxenhancer] WARN - GeneratorBase.update(1607) : class not updated : org.appfu
se.model.User
[jpoxenhancer] ENHANCED: org.appfuse.model.User

test:
[junit] DEBUG - UserDAOTest.testGetUsers(48) : [org.appfuse.model.User@13726
56[
[junit]     id=31
[junit]     firstName=Rod
[junit]     lastName=Johnson
[junit]     jdoObjectId=org.appfuse.model.PrimaryKey@1b6101e[
[junit]     id=31
[junit] ]
[junit]     jdoLoadedFields=<0, 1, 2>
[junit]     jdoModifiedFields=<>
[junit] ]]
[junit] INFO - UserDAOTest.testSaveUser(60) : org.appfuse.model.User@192a848
[
[junit]     id=36
[junit]     firstName=Rod
[junit]     lastName=Johnson
[junit]     jdoObjectId=<null>
[junit]     jdoLoadedFields=<null>
[junit]     jdoModifiedFields=<null>
[junit] ]
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(75) : removing user...
[junit] WARN - FetchRequest.execute(310) : Object with id "org.appfuse.model
.PrimaryKey@941db6[
[junit]     id=41
[junit] ]" not found !
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(85) : Expected exception: N
o such database row; nested exception is javax.jdo.JDOObjectNotFoundException: N
o such database row
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 2.234 sec

BUILD SUCCESSFUL
Total time: 6 seconds
$

```

Figure 7.6: Successful build, result of the `ant test -Dtestcase=UserDAO` test

Another approach to using transactions in your unit tests is available [on the Spring Forums](#).

Caching

See [JPOX's website](#) for information on its caching support.

Community and Support

JDO is a standard, so unlike the other persistence options, it's likely that many vendors will create JDO-compliant products. Many good books on JDO are available. As you may have noticed from this short tutorial, it's not as refined as the other options; it requires an *enhancement* step that the other frameworks don't. The JPOX project has [user forums](#), [good documentation](#) and seems to be one of the few that offers JDO 2.0 support.

[Kodo JDO](#) is a commercial implementation of JDO by SolarMetric. Along with their product, they offer commercial support and training. They also have a [sample app for download](#) that uses Spring and Kodo JDO.

For the past year, there's been competition in the J2EE Community between the JDO 2 Expert Group and the EJB 3 Expert Group. Both groups are implementing persistence based on ORM, but neither wanted to cooperate with the other. The EJB 3 group is basing much of its work on Hibernate, while JDO's Expert Group is simply trying to improve its first standard to be friendlier for relational database.

In a surprising turn of events, it was [recently announced](#) that the EJB 3 and JDO 2 groups would begin cooperating to define a new, unified POJO persistence model for Java in the J2EE 5.0 time frame. This will most likely result in a new standard for O/R mapping. Then both groups will use this standard to define their mappings.

OBJ

[ObjectRelationalBride \(OBJ\)](#) is an Apache project that is most similar to Hibernate. In fact, its description is quite similar: an O/R mapping tool that allows transparent persistence for Java Objects against relational databases. Because of this similarity, this chapter doesn't describe how it works. You can find a complete [feature list](#) on OBJ's website.

This section explains how to configure and use OBJ with Spring.



NOTE

Spring's OBJ support classes are located in the [org.springframework.orm.obj](#) and [org.springframework.orm.obj.support](#) packages.

Dependencies

This example uses OBJ version 1.0.0. Below are the JARs included in the MyUsers download as part of OBJ.

- **commons-dbcp.jar** and **commons-pool.jar**: Connection pool implementation
- **db-obj-1.0.0.jar**: Core OBJ classes

Configuration

The first step to integrating OJB is to create a [repository file](#) that maps objects to tables.

1. Create a new file named *repository.xml* and put it the *src/org/appfuse/model* directory.

```
<descriptor-repository version="1.0">
  <jdbc-connection-descriptor jcd-alias="dataSource"
    default-connection="true" useAutoCommit="1" platform="MySQL">
    <sequence-manager
      className="org.apache.ojb.broker.util.sequence.SequenceManagerNativeImpl"
    />
  </jdbc-connection-descriptor>

  <class-descriptor class="org.appfuse.model.User" table="app_user">
    <field-descriptor name="id" column="id" primaryKey="true"
      autoincrement="true" access="readonly"/>
    <field-descriptor name="firstName" column="first_name"/>
    <field-descriptor name="lastName" column="last_name"/>
  </class-descriptor>
</descriptor-repository>
```

The first part of this file describes the JDBC connection information (<jdbc-connection-descriptor>). The platform helps OJB figure out how it should retrieve primary keys, so it's important that it match your database. The *jcd-alias* attribute points to a "dataSource" bean that you need to define.

2. Create an *applicationContext-ojb.xml* file in the *web/WEB-INF* directory. While you're in that directory, rename *applicationContext-jdo.xml* to *applicationContext-jdo.xml.txt*. Also change the dependencies in *build.xml* so the "test" and "war" targets don't depend on "enhance". Add the "dataSource" bean definition as seen below:

```
<?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>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
      <value>jdbc:mysql://localhost/myusers</value>
    </property>
    <property name="username"><value>root</value></property>
```

```

        <property name="password"><value></value></property>
    </bean>
</beans>

```

3. Add a "transactionManager" bean definition that uses a [PersistenceBrokerTransactionManager](#). This transaction manager is similar to the others; it binds an OJB [PersistenceBroker](#) from the specified key to the thread. [PersistenceBrokerTemplate](#) is aware of thread-bound persistence brokers and participates in such transactions automatically.

```

<bean id="transactionManager"
      class="org.springframework.orm.ojb.PersistenceBrokerTransaction
        Manager"/>

```

4. Add an "objConfigurer" bean. [LocalObjConfigurer](#) exposes Spring's [BeanFactory](#) to OJB so it can use the Spring-managed DataSource.

```

<bean id="objConfigurer"
      class="org.springframework.orm.ojb.support.LocalObjConfigurer"/>

```

5. Configure OJB to recognize Spring's "dataSource" bean. Do this by overriding a few settings in the default OJB.properties.
 - a. [Download the default OJB.properties file](#) and put it in *web/WEB-INF/classes*. Be sure it's named *OJB.properties*.
 - b. Change repositoryFile to the following:

```
repositoryFile=org/appfuse/model/repository.xml
```

- c. Change ConnectionFactoryClass to the following to allow a Spring-defined DataSource:

```
ConnectionFactoryClass=org.springframework.orm.ojb.support.LocalDataSo
urceConnectionFactory
```

- d. Change ObjectCacheClass to use a first-level cache per broker:

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl
```

6. Create a `UserDAOObjb.java` class in `src/org/appfuse/dao/ojb` (you must create this directory). This class extends `PersistenceBrokerDaoSupport` and implements `UserDAO`.

```
package org.appfuse.dao.ojb;

// organize imports with your IDE

public class UserDAOObjb extends PersistenceBrokerDaoSupport implements
UserDAO {

    public List getUsers() {
        return new ArrayList(getPersistenceBrokerTemplate().
            getCollectionByQuery(new QueryByCriteria(User.class)));
    }

    public User getUser(Long id) {
        Criteria criteria = new Criteria();
        criteria.addEqualTo("id", id);
        User user = (User) getPersistenceBrokerTemplate()
            .getObjectByQuery(new QueryByCriteria(User.class,
criteria));
        if (user == null) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
        return user;
    }

    public void saveUser(User user) {
        getPersistenceBrokerTemplate().store(user);
    }

    public void removeUser(Long id) {
        getPersistenceBrokerTemplate().delete(getUser(id));
    }
}
```

7. Add this class as the "userDAO" in `web/WEB-INF/applicationContext-ojb.xml`:

```
<bean id="userDAO" class="org.appfuse.dao.ojb.UserDAOObjb"/>
```

8. The `app_user` table created by JDO does not allow nulls in the `id` column. With OJB, it's easiest to insert nulls for primary keys and retrieve the generated id back from the database. To do this, drop and recreate the `app_user` table.
 - a. Login to MySQL by typing `mysql -u root -p myusers` from the command line.
 - b. Execute `drop table app_user;.`
 - c. Execute the following SQL statement:

```
create table app_user (id bigint not null auto_increment,  
first_name varchar(50), last_name varchar(50),  
primary key (id));
```

Test It!

Before you run the `UserDAOTest`, change its `setUp()` method so it doesn't override the `userDAO` you just created. Below is a `setUp()` method that eliminates loading the transaction-wrapped DAO for JDO.

```
protected void setUp() throws Exception {  
    //String[] paths = {"org/appfuse/dao/applicationContext-test.xml"};  
    //ctx = new ClassPathXmlApplicationContext(paths, ctx);  
    dao = (UserDAO) ctx.getBean("userDAO");  
}
```

You can also simply change the `class` attribute of the `target` bean in `applicationContext-test.xml`:

```
<property name="target">  
    <bean class="org.appfuse.dao.ojb.UserDAOOjb"/>  
</property>
```

Run `ant test -Dtestcase=UserDAO`. If you've used the first option (commenting out lines in `setUp()`), you'll see this warning: "No running tx found." The second option will eliminate this warning.

Caching

See [OJB's website](#) for information on its caching support.

Community and Support

OJB 1.0 was [released](#) on June 30, 2004. It's an Apache project with a [fair amount](#) of real-world implementations. For support, your best option is to use its [mailing lists](#). The OJB Website also has [excellent documentation](#).

Summary

This chapter explored the different persistence options that Spring supports: Hibernate, iBATIS SQL Maps, its own JDBC abstraction, JDO and OJB. You saw how Spring configures each one in the MyUsers application. It is my opinion that Hibernate and iBATIS are the best tools to use for persistence.

Hibernate makes it easy to persist objects using simple mapping files, and you can even use XDoclet to generate the mapping files for you. iBATIS is ideal if you have an existing, complicated schema or legacy database - or if you simply prefer writing your own SQL.

This chapter has shown how powerful Spring is as a configuration tool and loose-coupling promoter. Of the five persistence frameworks, you only had to change the `UserDAOTest` for one. Not only does Spring promote good design, it makes good design a lot easier to use.

Testing Spring Applications

How Spring Makes Testing Easier

This chapter explains how to use test-driven development to create high-quality, well-tested, Spring-based applications. You will learn how to test your components using tools like EasyMock, jMock and DBUnit. For the Controllers, you will learn how to use Cactus for in-container testing, and Spring Mocks for out-of-container testing. Lastly, you will learn how to use jWebUnit and Canoo's WebTest for testing the web interface.

Overview

Using *test-driven* development is the best way to produce high-quality code. Not only do you think about how your application is supposed to work as you're writing the test (from your user's APIs perspective), but you actually *design* the contracts that your application is supposed to fulfill. If your tests don't pass as you expect, then you have a problem with either your test or your implementation. If your tests do as expected in that contract, you must modify your implementation to produce that expectation.

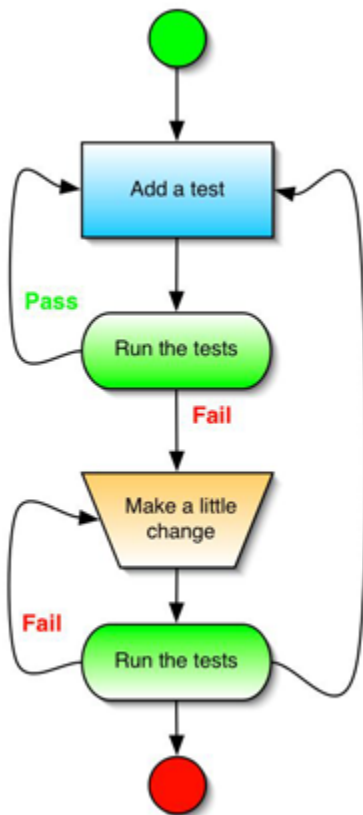


Figure 8.1: Model of test-first development

Test-first development is where developers write tests and ensure they fail before proceeding. They continue to take baby steps to get the test to pass/fail as they develop. The diagram^a on the left illustrates this technique.

While *test-driven* and *test-first* techniques are similar, the most important aspect is that the process of writing tests forces you to consider how your classes behave. *Test-first* advocates a more rigorous system of small steps.

In this chapter, you will learn how to use testing technologies to develop Spring applications faster and more efficiently. By employing test-first development, you can virtually eliminate the trial-and-error procedures that are common with developing browser-based applications. You can also gain confidence in your code and add regular automation to ensure that no one "broke the build."

Developing software that you can test easily is a very liberating experience because it allows you to find out quickly whether something worked.

a. This diagram is based on one in Scott W. Ambler's [Test Driven Development](#) essay.

Spring makes it easy to write testable software. Its IoC Container gives you the freedom to set dependencies on classes any way you like. You can load Spring's [ApplicationContext](#) in your test and use your beans like your application does, or you can set a bean's dependencies using mock objects. This chapter explores both techniques and shows you how to use tools like [EasyMock](#) and [jMock](#) to create mock versions of your classes on-the-fly. You will also see how to use [DbUnit](#) to load data in a database before testing.



NOTE

A mock object is a fake and simplistic version of a real object. For instance, you can use mocks to imitate the Servlet API so you can run tests outside of a servlet container.

For testing the Controllers in your application, you will learn how to use [StrutsTestCase](#), Spring Mocks and [Cactus](#). For testing the view layer, you will learn how to use [jWebUnit](#) and [Canoo WebTest](#).



NOTE

A future chapter will cover [Anthill](#), [CruiseControl](#) and [DamageControl](#). These automated testing systems are essential for maintaining a high-quality code base.

JUnit

[JUnit](#) has quickly become the *de facto* standard for TDD in Java. It's an easy framework to use and most modern IDEs support it. If you don't use an IDE, you probably use Ant, which is also easy to use with a [<junit> task](#). In this chapter, you will use JUnit to write most of your tests. If you're not using JUnit directly, you'll be using an extension. For example, DbUnit, Cactus and StrutsTestCase are all JUnit extensions.

A basic JUnit test simply contains a `testXXX()` method, where XXX is a descriptive name of what the test does. It's important to use a name that describes the test accurately. For example, if you have an object that allows results to be filtered, `testGetUsersWithNoFiltering()` is better than the more generic `testGetUsers()`. Below is an example that demonstrates a simple JUnit test:

```
package org.appfuse.dao;

import junit.framework.TestCase;

import org.appfuse.model.User;

public class SimpleTest extends TestCase {

    public void testGetFullName() {
        User user = new User();
        user.setFirstName("Jack");
        user.setLastName("Raible");
        assertEquals(user.getLastName(), "Jack Raible");
    }
}
```

Running this test results in a failure because you're getting the last name, rather than the full name. Running your tests with failures first is important to make sure they meet your expectations. Changing `assertEquals()` to the following will yield a passing test:

```
assertEquals(user.getFullName(), "Jack Raible");
```

Optional methods to implement in a JUnit Test include `setUp()`, `tearDown()` and `main(String[])`. Using `setUp()` and `tearDown()` are useful for creating a testing environment, then tearing down that environment *before running the next test*. These methods are called before and after each `testXXX()` method. If you have test-wide setup procedures, you should implement those in a static initialization block or use JUnit's `TestSetup` class.

It is necessary to implement the `main(String[])` to run a JUnit test from the command line using the `java` command. However, with modern IDE and Ant support, you may never need to implement this method. To prove it, remove the `main(String[])` from the `UserDAOTest` in `MyUsers`; your tests will continue to run.

In the Java Community, using TDD has increasingly become the norm, especially among open-source developers. The projects with a rich test suite typically gain more credibility from other developers. However, there is a philosophical difference of *how* one should do test-driven development. Many advocate the use of Mock Objects, or mocks, while others think integration testing is good enough.

This chapter often refers to *unit tests*. You may not agree that it's a *unit test*, but rather an *integration test*. J. B. Rainsberger said it best in [JUnit Recipes](#):

"The testing that programmers do is generally called unit testing, but we prefer not to use this term. It is overloaded and overused, and it causes more confusion than it provides clarity. As a community, we cannot agree on what a unit is - is it a method, a class, or a code path? If we cannot agree on what unit means, then there is little chance we will agree on what unit testing means."

This chapter shows you two types of testing: *mock testing* and *integration testing*. Let's take a brief look at the philosophies behind them.

Mock Testing

Mock testing is the process of isolating your test case to test a single unit of work, most often a class. Using mocks, you can test your class in isolation, without reliance on any of its dependencies. In the instance of a business façade (or Manager), you would *mock* its dependent DAO. This allows you to test the business façade for its specific functionality, not including the DAO's functionality.

A mock is typically a *stub* (a minimal test-only class you write yourself) or a *dynamic mock*. Dynamic mocks are a slick way of setting *expectations* on a class or interface. With dynamic mocks, you can create an instance of your class on-the-fly and write code to set expected behavior. The general rule for using mocks is "only mock types that you can change," or "only mock APIs that are yours."

Integration Testing

Integration testing is the process of testing your classes in their normal environment with all of their dependencies intact. The main disadvantages of integration testing are *speed* and *non-isolated tests*. For example, for a business façade test that depends on its real DAO, you must establish a database connection to retrieve actual data. By coding and running non-isolated tests, you're writing tests that depend on your environment, which really has nothing to do with your test.

There is a place for both types of tests. Dynamic mocks are great for team development, where different developers are responsible for different layers. For instance, if developer Bruce is responsible for the DAO layer and developer James is responsible for the business logic layer, they have no reason to be dependent on each other's implementations. Bruce can provide James with interfaces and James can create mocks of these in order to test his code in isolation. Then it doesn't matter if Bruce takes longer to deliver his DAO implementation.

If you're working on a small team or alone, integration tests will probably suffice. Of course, this means you will have to develop your application in a specific order, where dependencies are coded first so implementations that depend on them can run properly.

In my opinion, a properly designed application has integration tests in the database layer, mock tests in the business and web layers and integration tests in the UI. Of course, this all depends on how complicated each layer is. For the database layer, it's important to test that its interaction with your underlying datastore works. There's no reason to mock the database connection or Spring-supplied `Template`. These are framework-level components and shouldn't be a concern of your application. Once your DAOs return the expected results, it's pointless to verify this again in the business layer. Therefore, you can mock the DAOs in the business (or service) layer and concentrate on testing your business logic.

In the web tier, particularly with MVC Controllers (regardless of web framework), it's helpful to mock a J2EE container environment. Mocks can impersonate requests, responses and other servlet-related classes, allowing you to run your tests out-of-container, which is often much faster. Using real business objects vs. mocks in your Controller tests is a matter of taste at that point. Testing them in isolation is usually a good idea, since the important functionality in Controllers is controlling which views are returned for specific inputs.

The UI layer is the best place to test the full stack of integrated functionality. Testing the UI layer involves shadowing the actions a user would take when using your application: clicking links, entering data, clicking buttons, etc. These tests don't talk to your classes directly, but perform browser actions. They typically pull HTML from a URL, parse it and submit request parameters based on your test code. The frameworks for UI testing make this very simple so you can write one-liners to submit forms, enter data and verify page titles or text.

Now that you understand the different strategies of testing, let's look at some specific examples in the context of the MyUsers application. In this chapter, like previous ones, you can follow along and do the examples as you go. The easiest way to do this is to download the **MyUsers Chapter 8** bundle from <http://sourcebeat.com/downloads>. This project tree is the result of *Chapter 7* exercises. It also contains all the JARs you will need in this chapter in its *web/WEB-INF/lib* directory. You can also use the application you've been developing in previous chapters. If you go this route, download **Chapter 8 JARs** from <http://sourcebeat.com/downloads>. Each section notes new JARs you might need, so you can use this chapter as a reference for integrating these principles into your own applications.

In *Chapter 7*, you integrated a number of persistence strategies. If you completed all the exercises in the chapter, your MyUsers application should be using OJB, since that was the last strategy explored. If you'd like to change your application to use a different framework, follow the specific section for that framework in *Chapter 7*. Below are a few things to remember if you decide to switch from OJB to something else:

- JDO requires modifications to *build.xml* so "enhance" is called by the "test" and "war" targets. You may also have to change `UserDAOTest` so it loads *applicationContext-test.xml*.
- You may need to drop the `app_user` table and recreate it (see the instructions for each DAO type).

If you have issues with switching DAOs, feel free to e-mail me at mattr@sourcebeat.com or enter an issue in the [Spring Live Issue Tracker](#) provided by [Atlassian](#).

Testing the Database Layer

As mentioned earlier, it's best to test your DAOs directly against your database. This ensures that your persistence framework works as you want. Testing a database layer involves retrieving, adding, updating and removing data using your application-specific classes. To review the `UserDAOTest` you created in *Chapter 2*:

```
package org.appfuse.dao;

// use your IDE to organize imports

public class UserDAOTest extends BaseDAOTestCase {

    private User user = null;
    private UserDAO dao = null;

    protected void setUp() throws Exception {
        dao = (UserDAO) ctx.getBean("userDAO");
    }

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

    public void testGetUsers() {
        // add a record to the database so we have something to work with
        user = new User();
        user.setFirstName("Rod");
        user.setLastName("Johnson");
        dao.saveUser(user);

        List users = dao.getUsers();
        assertTrue(users.size() >= 1);
        assertTrue(users.contains(user));
    }

    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());

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

```

This class is an *integration test* because it depends on Spring's [ApplicationContext](#) to retrieve the UserDAO implementation class, with all its dependencies set. The ApplicationContext is initialized using [ClassPathXmlApplicationContext](#) in the BaseDAOTestCase (which this class extends).

```

protected ApplicationContext ctx = null;

public BaseDAOTestCase() {
    String[] paths = { "/WEB-INF/applicationContext*.xml" };
    ctx = new ClassPathXmlApplicationContext(paths);
}

```

In addition to the [ClassPathXmlApplicationContext](#), a [FileSystemXmlApplicationContext](#) exists. Both of these classes allow the Ant-style syntax (*.xml) to load any resources that match the specified pattern. If you'd rather load context files using that class, you could refactor the above code to the following:

```

public BaseDAOTestCase() {
    String[] paths = { "web/WEB-INF/applicationContext*.xml" };
    ctx = new FileSystemXmlApplicationContext(paths);
}

```

**TIP**

It is possible to store your bean definitions in .properties files, too. [Read more](#) about [PropertiesBeanDefinitionReader](#).

The advantage of using `ClassPathXmlApplicationContext` is your files don't have to be in a precise location; they just have to be in your classpath. The reason I'm using `/WEB-INF/applicationContext*.xml` instead of `/applicationContext*.xml` is because the first is in my classpath and the second isn't. The classpath is set in both the Eclipse .classpath file and the Ant build file.

The Eclipse setting in the .classpath file is listed below:

```
<classpathentry excluding="WEB-INF/classes/" kind="src" path="web"/>
```

The Ant setting in "test" target is listed below:

```
<classpath>
  <path refid="classpath"/>
  <path location="${build.dir}/classes"/>
  <path location="${test.dir}/classes"/>
  <path location="web/WEB-INF/classes"/>
  <path location="web"/>
</classpath>
```

Using `/WEB-INF/applicationContext*.xml` is also helpful because `action-servlet.xml` refers to validator resource files using this path, requiring the `web` folder to be in the classpath when testing classes use that file. When you refactor your Controller tests to be self-contained unit tests, you will no longer need to load the context, freeing you to change this path.

DbUnit

DbUnit is a testing framework useful for putting a database in a known state before running tests. DbUnit has a single JAR that you need in your classpath: **dbunit-2.1.jar**. This is included in the MyUsers download and *Chapter 8* JARs download. You can also [download](#) it from SourceForge.

In the `UserDAOTest`, a record is added in the `testGetUsers()` method. Refactor this class to use DbUnit to load a record before running the test. You must first create some sample data with which to populate the database. A number of [datasets](#) are available to use with DbUnit, but the easiest is the [XmlDataSet](#).



TIP

To export data from an existing database, use DbUnit's [Ant tasks](#).

1. Create a *sample-data.xml* file in *test/data* (you must create this directory). Populate it with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<dataset>
  <table name='app_user'>
    <column>id</column>
    <column>first_name</column>
    <column>last_name</column>
    <row>
      <value>1</value>
      <value>Rod</value>
      <value>Johnson</value>
    </row>
  </table>
</dataset>
```

2. With DbUnit, you can extend its [DatabaseTestCase](#) or use your own `TestCase`. It's easier to use your own `TestCase` than to change the parent class. Add the following two variables as member variables of the `BaseDAOTestCase` class:

```
private IDatabaseConnection conn = null;
private IDataset dataSet = null;
```

3. Create a `setUp()` method to clear out the database for any tables specified in the *sample-data.xml* file.

```
protected void setUp() throws Exception {
    DataSource ds = (DataSource) ctx.getBean("dataSource");
    conn = new DatabaseConnection(ds.getConnection());
    dataSet = new XmlDataSet(new FileInputStream(
        "test/data/sample-data.xml"));
    // clear table and insert only sample data
    DatabaseOperation.CLEAN_INSERT.execute(conn, dataSet);
}
```

4. Add logic in the `tearDown()` method to close the connection and delete any added data.

```
protected void tearDown() throws Exception {
    // clear out database
    DatabaseOperation.DELETE.execute(conn, dataSet);
    conn.close();
    conn = null;
}
```

5. Alter **UserDAOTest** so the `setUp()` and `tearDown()` methods call `super.setUp()` and `super.tearDown()`.

```
protected void setUp() throws Exception {
    super.setUp();
    dao = (UserDAO) ctx.getBean("userDAO");
}

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



NOTE

If you'd prefer not to explicitly grab beans from the context, [use Dependency Injection in your unit tests](#). Explicitly grabbing your beans allows you to use whatever naming convention you like for the variables. In Spring 1.1.1, an [AbstractDependencyInjectionSpringContextTests](#) class was added for dependency injection in your tests.

6. Change the `getUsers()` method to verify only one record is in the database, and it's the one DbUnit entered.

```
public void testGetUsers() {  
    List users = dao.getUsers();  
    assertTrue(users.size() == 1);  
    User user = (User) users.get(0);  
    assertEquals(user.getFullName(), "Rod Johnson");  
}
```

Running `ant -Dtestcase=UserDAO` should yield "BUILD SUCCESSFUL," and the `app_user` table should be empty in the database. In performance comparisons, this test takes a mere 0.21 seconds longer than the previous one (0.056 vs. 0.77).

Ant is another way to use DbUnit in your application. You can configure a target that loads the database before running your tests. AppFuse uses this strategy; see its [build.xml](#) file and the `db-load` target for an example. This is useful because you can run a block of tests with pre-defined data, rather than reloading the data for each test. The advantage of using this strategy is your suite of tests will run faster since the database isn't reset before each test is run. An advantage of using DbUnit directly in your classes is you can run your tests in an IDE without any Ant dependencies.

Database Switching

Using Spring to configure your database connection makes it easy to swap the database with which you test your code. For example, you could use an HSQL database for unit tests, and switch to a DB2 database for production. In most cases, it's a good idea to test against your production database, but if your developers use PowerBooks, DB2 doesn't have an install for OS X.

One way to configure database switching is to use two properties files. Most of Spring's sample applications (in *CVS/samples*) use this strategy. By defining a "propertyConfigurer" bean (with class [PropertyPlaceholderConfigurer](#)), you can configure your "dataSource" bean's properties with ant-style placeholders: `${...}`. This allows you to refer to test settings and production settings. It also allows you to share your database configuration between your Spring config files and your *build.xml* file. To configure a properties file with your database settings in *MyUsers*, complete the following steps:

1. Modify your current DAO strategy's XML file. This should be *applicationContext-obj.xml*. Replace the "dataSource" bean's property values with Ant-style properties:

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

2. Create a *jdbc.properties* file in the *web/WEB-INF/classes* directory:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/myusers
jdbc.username=root
jdbc.password=
```

3. Add a *PropertyPlaceholderConfigurer* bean to *web/WEB-INF/applicationContext.xml*. The commented out section at the bottom of this definition shows you how to use a single "location" property as an alternative to the "locations" property.

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholder
        Configurer">
  <property name="location">
    <value>classpath:jdbc.properties</value>
  </property>
</bean>
```

To specify multiple properties files for loading, use the "locations" property instead of "location".

```
<property name="locations">
  <list>
    <value>classpath:jdbc.properties</value>
  </list>
</property>
```

4. To expose *jdbc.properties* to *build.xml*, add it as a properties file at the top of *build.xml*:

```
<property file="build.properties"/>
<property file="web/WEB-INF/classes/jdbc.properties"/>
```

5. Change the "populate" target's <sql> task to use the following:

```
<sql driver="${jdbc.driverClassName}" url="${jdbc.url}"
userid="${jdbc.username}" password="${jdbc.password}">
```

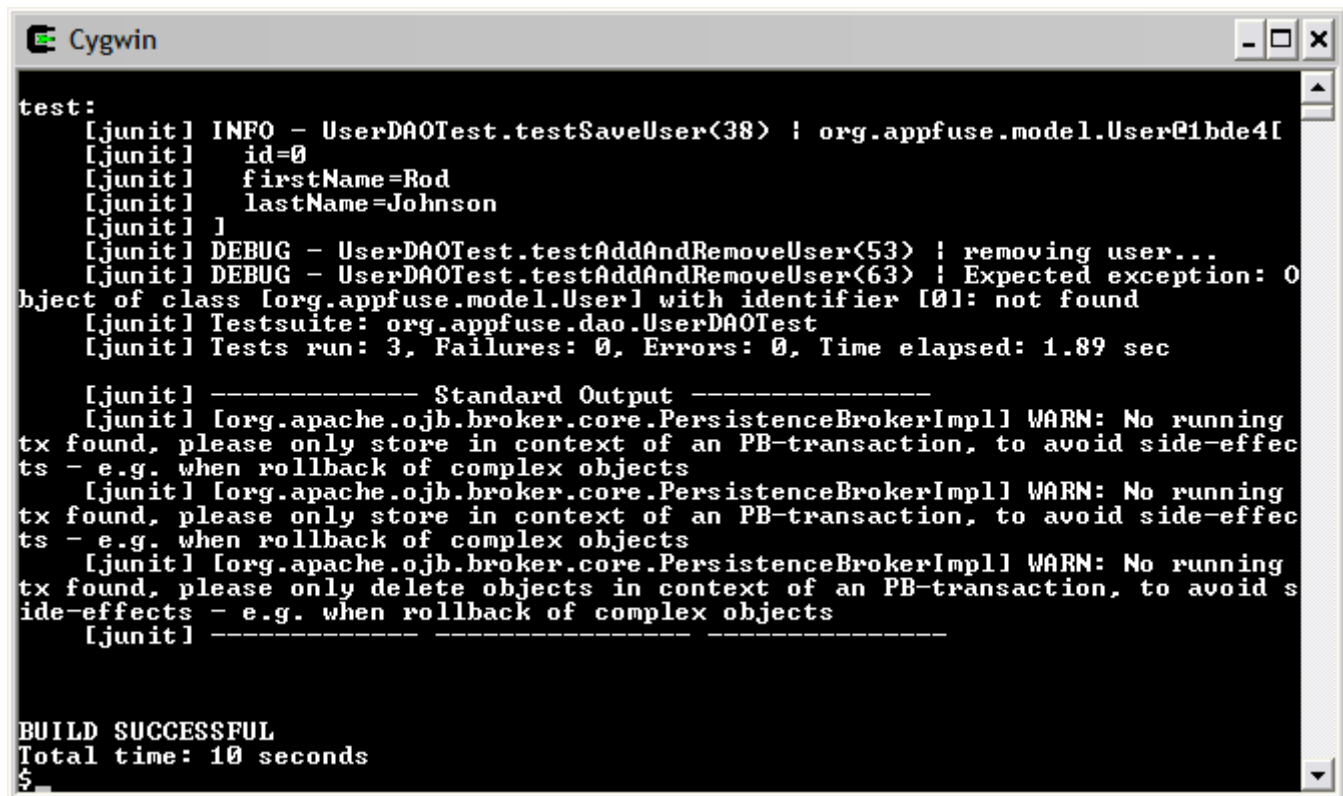


NOTE

Another strategy (used in AppFuse) is to use a manually configured DataSource when testing and a JNDI DataSource when in production.

Overriding Beans for Tests

If you have DAOs with methods that require transactions to be wrapped around them, you can define beans specifically for unit tests. This was done in *Chapter 7* for the JDO DAO because JDO requires that any `makePersistent()` calls are inside a transaction. Using Spring's declarative transactions in your test is much easier than using Transactions. If you're still using the OJB DAO, you may have received several warning messages about no running transaction.



```
test:
[junit] INFO - UserDAOTest.testSaveUser(38) : org.appfuse.model.User@1bde41
[junit]      id=0
[junit]      firstName=Rod
[junit]      lastName=Johnson
[junit] 1
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(53) : removing user...
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(63) : Expected exception: 0
object of class [org.appfuse.model.User] with identifier [0]: not found
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 1.89 sec

[junit] ----- Standard Output -----
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl1] WARN: No running
tx found, please only store in context of an PB-transaction, to avoid side-effec
ts - e.g. when rollback of complex objects
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl1] WARN: No running
tx found, please only store in context of an PB-transaction, to avoid side-effec
ts - e.g. when rollback of complex objects
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl1] WARN: No running
tx found, please only delete objects in context of an PB-transaction, to avoid s
ide-effects - e.g. when rollback of complex objects
[junit] -----

BUILD SUCCESSFUL
Total time: 10 seconds
$
```

Figure 9.1: Warning messages when running the OJB DAO

To fix this, create a bean definition just for testing, and override the regular `userDAO` by loading this definition in `UserDAOTest`.

1. In `test/org/appfuse/dao`, create an `applicationContext-test.xml` file with a transaction-wrapped "userDAO" bean:

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

<beans>
    <!-- userDAO for testing UserDAOJdo -->
    <bean id="userDAO"
        class="org.springframework.transaction.interceptor.TransactionProxy
            FactoryBean">
        <property name="transactionManager"><ref
            bean="transactionManager"/></property>
        <property name="target">
            <bean class="org.appfuse.dao.objb.UserDAOObjb" autowire="byName"/
        >
            </property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
</beans>
```

2. Refactor the `setUp()` method of `UserDAOTest` to load this file before each test.

```
protected void setUp() throws Exception {
    super.setUp();
    String[] paths =
        {"/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

3. Now the `UserDAOTest` should run (`ant test -Dtestcase=UserDAO`) without the transactions warning.



NOTE

You won't use this test configuration in the `UserManagerTest` since the "userManager" bean is already wrapped in transactions.

Use JNDI DataSource in Tests

As mentioned previously, you can define two different "dataSource" beans for testing and production. The production bean will likely be a DataSource that's looked up via JNDI. An example definition is below:

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

To use this bean in your tests, use Spring's JNDI mocks to bind this to a simple JNDI implementation.

```
try {
    SimpleNamingContextBuilder builder =
        SimpleNamingContextBuilder.emptyActivatedContextBuilder();

    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://localhost/myusers");
    ds.setUsername("root");
    ds.setPassword("");
    builder.bind("java:comp/env/jdbc/myusers", ds);
} catch (NamingException ne) {
    // do nothing, test will fail on its own
}
```

To try this, put the JNDI DataSource in *applicationContext-test.xml* and add the context build code (above) to the beginning of the static initialization block in *BaseDAOTestCase.java*. For the *DriverManagerDataSource*'s properties above, you could also use a *ResourceBundle* to pull the information from *jdbc.properties*.

To use a JNDI DataSource in Tomcat, replace the "dataSource" bean in your DAO context file with the JNDI one configured in your container appropriately. Below are instructions for Tomcat 5.x:

1. Create a *myusers.xml* file in *\$CATALINA_HOME/conf/Catalina/localhost* and populate it with the XML below:

```
<Context path="/myusers" docBase="myusers" debug="0" reloadable="true">
  <Resource name="jdbc/myusers" auth="Container"
    type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/myusers">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter><name>maxActive</name><value>30</value></parameter>
    <parameter><name>maxIdle</name><value>5</value></parameter>
    <parameter><name>maxWait</name><value>5000</value></parameter>
    <parameter><name>username</name><value>root</value></parameter>
    <parameter><name>password</name><value></value></parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>defaultAutoCommit</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost/myusers</value>
    </parameter>
  </ResourceParams>
</Context>
```

2. Copy the MySQL JDBC Driver (*mysql-connector-java-3.0.14-production-bin.jar*) from *web/WEB-INF/lib* to *\$CATALINA_HOME/common/lib*.
3. Add the following XML fragment to the bottom of *web/WEB-INF/web.xml*.

```
<resource-ref>
  <description>Connection Pool</description>
  <res-ref-name>jdbc/myusers</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Now if you run **ant deploy** and stop/start Tomcat, everything should work as before.

Load Context Once

To speed up your unit tests, especially the ones that load an `ApplicationContext`, alter your tests so the context is only loaded once for the entire test. In the previous examples, the context is loaded by `setUp()` before running each `testXXX()` method. You can do this using one of two ways:

- Use JUnit's [TestSetup](#) class to configure once-per-test methods.
- Use a static initialization block in your test to initialize resources once.

To use either of these methods, you must first change some variables to be static.

1. Change `BaseDAOTest` so its `ctx` variable is static:

```
protected static ApplicationContext ctx = null;
```

2. Change `UserDAOTest` so the `dao` variable is static:

```
private static UserDAO dao = null;
```

3. The second *static initialization block* is easier to implement; simply add the following block to the beginning of `BaseDAOTestCase`.

```
static {
    String[] paths = {"/WEB-INF/applicationContext*.xml",
                     "/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths);
}
```

4. Remove the block to load the test context in the `UserDAOTest.setUp()` method.

```
public void setUp() throws Exception {
    super.setUp();
    String[] paths = {
        "/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

Now running `ant test -Dtestcase=UserDAO` should be a bit faster. Right now, the difference is negligible, but it will greatly improve as your context files grow in size.

While using `TestSetup` is easy, I've chosen not to include it in this chapter. Using a static initialization block works great and it's [significantly faster](#) than `TestSetup`. Using `TestSetup` vs. static initialization blocks is [a matter of personal preference](#). Other alternatives are available, such as Cedric Beust's [TestNG](#) framework. For more information on `TestSetup`, and how JUnit initializes objects for each test, see [Martin Fowler's website](#).

Testing the Service Layer

In the last section, you learned how to test your DAO implementations against a database. This is important for your database layer to verify that what you put into the database is the same thing that comes out of it. However, once you have your DAOs tested and verified, you don't have much reason to use them in your service layer tests.

A nice feature of Spring is that it not only promotes interface-based design, but it doesn't put any restrictions on how you write your unit tests. As the last section demonstrates, you can easily load your context files in your tests and interact with your beans just as you would in a production environment. This is an important concept that is often overlooked. Many folks get caught up in mocking everything, which is great for test isolation, but it ignores testing how the application's layers interact with each other.

You should always try to use integration tests first (load the context in your test and operate on your beans accordingly). This is because it's easy to set up and it tests your *real* code, not a fake object. However, this approach breaks down in two areas:

- You're working on a team and each developer is responsible for different layers. You don't want to wait for an implementation before writing tests for a parent layer.
- Your application context files grow so large that it takes several seconds to load them.

To work around these issues, use *mocks* to simulate dependencies of your class.

Using Mocks for DAO Objects

This section refactors the `UserManagerTest` to use mocks for the `UserDAO`. There are two types of mocks: **stubs**, which you create yourself, and **dynamic mocks**, which allow you to implement classes on-the-fly. In the process of creating dynamic mocks, the developer is responsible for setting expected results when methods are called. Dynamic mocks are typically easier to use and don't litter your source tree with a second set of classes to maintain.

EasyMock

[EasyMock](#) is a project that provides mocks for interfaces in JUnit tests by generating them on-the-fly using [Java's proxy mechanism](#). A class extension is also available to mock implementation classes. To use EasyMock in a test, perform the following steps:

1. Create a `MockControl` for the interface you would like to simulate.
2. Get the mock from the `MockControl`.
3. Specify the behavior of the mock (record state).
4. Activate the mock with the control (replay state).
5. Execute the methods you want to test.
6. Verify that the methods were called as expected.

The code below illustrates these setups:

```
private UserManager mgr = new UserManagerImpl();
private MockControl control;
private UserDAO mockDAO;
protected void setUp() throws Exception {
    // Create a MockControl
    control = MockControl.createControl(UserDAO.class);
    // Get the mock
    mockDAO = (UserDAO) control.getMock();

    // Set required dependencies
    mgr.setValidator(new UserValidator());
    mgr.setUserDAO(mockDAO);
}

protected void testGetUser() {
    // Set expected behavior
    mockDAO.removeUser(new Long(1));
}
```



```
control.setVoidCallable();

// Active the mock
control.replay();

// Execute method to test
mgr.removeUser("1");

// Verify methods called
control.verify();
}
```

This code never uses an implementation of the `UserDAO`. Rather, a mock is created from its interface and a dynamic implementation is created using `EasyMock`.

In the `MyUsers` application, the `userManagerTest` *integration test* already exists for the middle tier. This class loads `applicationContext.xml` files and uses the beans as you would normally in your application. Refactoring this class to use `EasyMock` is easy and will run much faster in the long run. You might as well keep your original `userManagerTest` around, since it will continue to serve as a nice integration test.

Create a new class named `userManagerEMTest` in `test/org/appfuse/service`. This class does not contain any Spring dependencies in this example and has the same basic functionality as `userManagerTest`. The main difference is this class is isolated (thanks to mocks) and will run very quickly. It has the following code:

```
package org.appfuse.service;

// use your IDE to organize imports

public class userManagerEMTest extends TestCase {
    private static Log log =
        LogFactory.getLog(userManagerEMTest.class);
    private UserManager mgr = new UserManagerImpl();
    private MockControl control;
    private UserDAO mockDAO;

    protected void setUp() throws Exception {
        control = MockControl.createControl(UserDAO.class);
        mockDAO = (UserDAO) control.getMock();
        mgr.setValidator(new UserValidator());
        mgr.setUserDAO(mockDAO);
    }

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

```
// set expected behavior on dao
mockDAO.saveUser(user);
control.setVoidCallable();

// switch from record to playback
control.replay();

user = mgr.saveUser(user);
assertEquals(user.getFullName(), "Easter Bunny");
control.verify();

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

// set userId since Hibernate doesn't do it
String userId = "1";
user.setId(new Long(userId));

// reset to record state
control.reset();
mockDAO.removeUser(new Long(1));
control.setVoidCallable();
control.replay();

mgr.removeUser(userId);

control.verify();

try {
    // reset to record state
    control.reset();
    control.expectAndThrow(mockDAO.getUser(user.getId()),
        new ObjectRetrievalFailureException(
            User.class, user.getId()));
    // switch to playback
    control.replay();

    user = mgr.getUser(userId);

    control.verify();
    fail("User '" + userId + "' found in database");
} catch (DataAccessException dae) {
    log.debug("Expected exception: " + dae.getMessage());
    assertNotNull(dae);
}
}
```

Run this test using `ant test -Dtestcase=UserManagerEM`. The speed difference is substantial between this test and `userManagerTest`. In my meager tests, the `UserManagerEM` is 0.172 seconds, while the `UserManagerTest` is 1.59 seconds.

jMock

[jMock](#) is another open-source project that produces dynamic mocks. jMock has a slightly different approach to mocking than EasyMock. It requires you to subclass `MockObjectTestCase`. This class performs method invocation verification and provides the syntactic sugar that makes jMock tests easy to read. To use jMock in a test, perform the following steps:

1. Create a [Mock](#) for the interface you would like to simulate.
2. Set expected behavior on the mock.
3. Execute methods you want to test.
4. Verify expectations.

The code below is a simple example of the above steps:

```
private UserManager mgr = new UserManagerImpl();
private Mock mockDAO;

protected void setUp() throws Exception {
    // Create a Mock
    mockDAO = new Mock(UserDAO.class);

    // Set dependencies
    mgr.setValidator(new UserValidator());
    mgr.setUserDAO((UserDAO) mockDAO.proxy());
}

protected void testGetUser() {
    // Set expected behavior
    mockDAO.expects(once()).method("getUser")
        .with( eq(new Long(1)) );

    // Execute method to test
    mgr.removeUser("1");

    // Verify expectations
    mockDAO.verify();
}
```

jMock's syntax is a bit cleaner, but its expectations syntax is more complicated. A nice feature of jMock is the ability to write [custom stubs](#) to simulate side-effects of calling methods. For instance, in the `UserDAOHibernate.saveUser()`, the `User` object is assigned an `id` by Hibernate (if one doesn't exist).

To mimic this same functionality when mocking `UserDAO`, create an `AssignIdStub` class in *test/org/appfuse/service*. The code for this class contains logic that merely sets a random `Long` as the `id` on the `User` object.

```
package org.appfuse.service;

// use your IDE to organize imports

public class AssignIdStub implements Stub {

    public StringBuffer describeTo(StringBuffer buffer) {
        return buffer.append("assigns random id to an object");
    }

    public Object invoke(Invocation invocation) throws Throwable {
        Long id = new Long(new Random().nextInt(10));
        ((User) invocation.parameterValues.get(0)).setId(id);

        return null;
    }
}
```

Create a `userManagerJMTest` class in this same directory.

```
package org.appfuse.service;

// use your IDE to organize imports

public class UserManagerJMTest extends MockObjectTestCase {
    private static Log log =
        LoggerFactory.getLog(UserManagerJMTest.class);
    private UserManager mgr = new UserManagerImpl();
    private Mock mockDAO;

    protected void setUp() throws Exception {
        mockDAO = new Mock(UserDAO.class);
        mgr.setValidator(new UserValidator());
        mgr.setUserDAO((UserDAO) mockDAO.proxy());
    }

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

        // set expected behavior on dao
        mockDAO.expects(once()).method("saveUser")
            .with( same(user) ).will(assignId());

        user = mgr.saveUser(user);

        // verify expectations
        mockDAO.verify();

        assertEquals(user.getFullName(), "Easter Bunny");

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

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

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

        mockDAO.expects(once()).method("removeUser")
            .with( eq(new Long(userId)) );

        mgr.removeUser(userId);

        // verify expectations
        mockDAO.verify();
    }
}
```

```

    try {
        // set expectations
        Throwable ex =
            new ObjectRetrievalFailureException(
                User.class, user.getId());

        mockDAO.expects(once()).method("getUser")
            .with( eq(new Long(userId)))
            .will(throwException(ex));

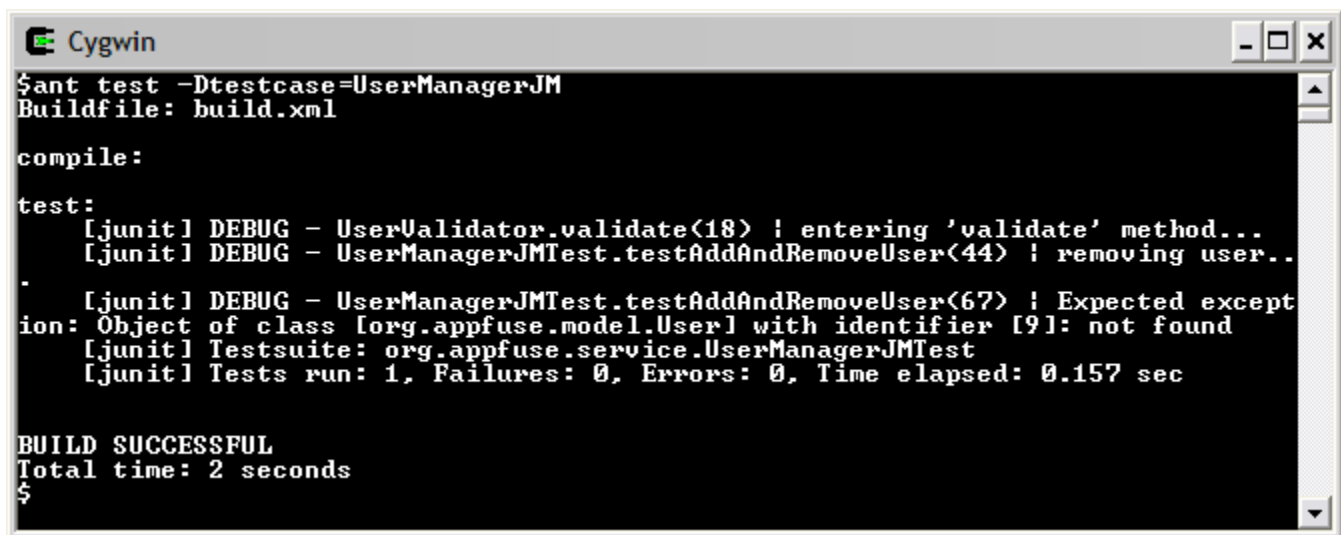
        user = mgr.getUser(userId);

        // verify expectations
        mockDAO.verify();
        fail("User '" + userId + "' found in database");
    } catch (DataAccessException dae) {
        log.debug("Expected exception: " + dae.getMessage());
        assertNotNull(dae);
    }
}

private Stub assignId() {
    return new AssignIdStub();
}
}

```

Run this test with `ant test -Dtestcase=UserManagerJM`. The test should run as fast as the EasyMock test, and its output should resemble the following screenshot.



```

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

compile:
test:
[junit] DEBUG - UserValidator.validate(18) ! entering 'validate' method...
[junit] DEBUG - UserManagerJMTest.testAddAndRemoveUser(44) ! removing user..
-
[junit] DEBUG - UserManagerJMTest.testAddAndRemoveUser(67) ! Expected except
ion: Object of class [org.appfuse.model.User] with identifier [9]: not found
[junit] Testsuite: org.appfuse.service.UserManagerJMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.157 sec

BUILD SUCCESSFUL
Total time: 2 seconds
$

```

Figure 8.3: Results of the `ant test -Dtestcase=UserManagerJM` test

This section has shown you how to do true unit tests on your business façades and how using Mocks in your testing strategy can greatly reduce the amount of time for tests to run.

In these tests, the `UserManagerImpl` class was instantiated with its default constructor, and a mock object simulated its interactions with the `UserDAO`. Spring was not in any of the mock tests, proving that it's non-evasive. Spring's JavaBeans philosophy for setter-based dependency injection made it easy to set whatever dependencies you wanted.

If you'd like to learn more about the differences between jMock and EasyMock, the jMock website has a [detailed comparison](#). In my experience, jMock is more flexible, but its API can be a bit difficult to understand. Since it requires you extend its `MockObjectTestCase`, you cannot always use it. jMock has a much more active mailing list, but EasyMock enjoys the benefit of being released first and used more widely.

Testing the Web Layer

Testing the web layer in an application is the most important part of an application test suite. By properly testing your Controllers, you verify the control-flow of your application and determine if inputs will return the expected outputs. In Spring MVC, inputs are requests and outputs are [ModelAndView](#) objects returned by Controller methods. Spring Mocks allow you to easily mimic requests and verify results. You can test in-container with Cactus.

While testing Controllers is important, testing the view by interacting with the User Interface (UI) is usually the best way to be sure your application works properly. You may have a Quality Assurance (QA) department that does this for you. However, it's easy to write tests for the UI using [jWebUnit](#) and [Canoo WebTest](#), both of which are simplifications of [HttpUnit](#). HttpUnit emulates the relevant portions of a browser's behavior, including form submission, JavaScript, basic http authentication, cookies and automatic page redirection. It allows Java test code to examine returned pages as either text, XML DOM, or containers of forms, tables, and links. Tests that interact with the UI are great *integration tests* to verify all layers of your application. Of course, they won't verify that colors, fonts and position are correct; that will always need a human eye to test.

You can automate all of the tests mentioned in this chapter. That is, you can test them without human interaction. This ability enables a healthy continuous build mechanism that you will explore in the *Automated Testing* section.

Testing Controllers

In *Chapter 4*, you created [JUnit](#) tests for the two main Controllers in MyUsers: `UserController` and `UserFormController`. In this section, you will refactor those tests to be independent of Spring's [ApplicationContext](#). Instead, you will use EasyMock and jMock to mock your Controller's dependencies.

You will create Cactus tests to see how to test Controllers in-container. You will learn how to use mocks in many of these tests to isolate your tests and make them true *unit tests* (they only test a single class). You will also learn how to write a test for the `FileUploadController` and test the e-mail it sends using [Dumbster](#).



NOTE

Dumbster is a fake SMTP server for unit and system testing applications that send e-mail messages. It responds to all standard SMTP commands but does not deliver messages to the user. The messages are stored within the Dumbster for later extraction and verification.

Spring Mocks

Spring Mocks is a general name for the mock classes that Spring distributes for testing [JNDI](#) and [Controllers](#). These classes were originally used internally by Spring to test the framework itself. As developers tested Spring's Controllers with mocks (specifically, Mock Object's Servlet API), it became apparent that Mock Objects did not provide rich enough functionality. In June 2002, Spring developers realized that their mocks were actually quite feature-rich, and they began releasing them as part of the distribution (since 1.0.2).

One of the best things about Spring mocks is that the classes are for mocking the Servlet API and aren't tied to Spring at all. This means that you could easily use them to test other Controllers, such as WebWork. In my experience, Struts, Spring and WebWork have the best support for testing. Struts has it via StrutsTestCase, Spring has its mocks and WebWork can be tested using regular JUnit tests. A later chapter will implement WebWork, Tapestry and JSF in MyUsers. Unlike the request/response web frameworks, Tapestry and JSF are component- and event-driven web frameworks. They often depend on JavaScript and currently don't have a lot of support for unit testing their Controller components.

Testing Controllers

Below is the `UserControllerTest` that you created in *Chapter 4*:

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserControllerTest extends TestCase {
    private static Log log = LogFactory.getLog(UserControllerTest.class);

    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");
    }
}
```

This class makes use of two Spring mocks: [MockHttpServletRequest](#) and [MockServletContext](#). It loads context files and grabs the beans from the initialized [XmlWebApplicationContext](#). This works well, but is more of an *integration test*, rather than a *unit test*. This is because Spring has already wired the `UserController` and all its dependent objects. In order to create a *unit test*, you must remove any trace of an `ApplicationContext` and set the dependencies manually in the unit test.

To refactor this test to use EasyMock, create a `UserControllerEMTest.java` class in `test/org/appfuse/web`. This class extends JUnit's `TestCase` class.

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserControllerEMTest extends TestCase {
    private MockControl control = null;
    private UserManager mockManager = null;
    private UserController c = new UserController();

    protected void setUp() throws Exception {
        control = MockControl.createControl(UserManager.class);
        mockManager = (UserManager) control.getMock();
        c.setUserManager(mockManager);
    }

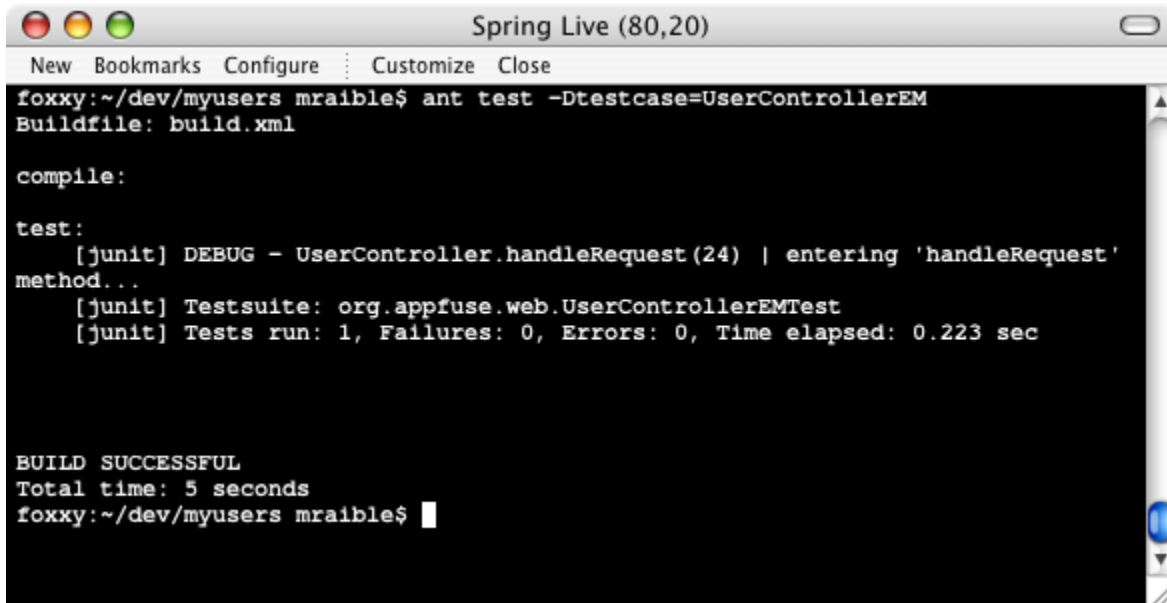
    public void testGetUsers() throws Exception {
        // set expected behavior on manager
        mockManager.getUsers();
        control.setReturnValue(new ArrayList());

        // switch from record to playback
        control.replay();

        ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
                           (HttpServletResponse) null);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");

        // verify getUsers() was called on manager
        control.verify();
    }
}
```

To run this test, execute `ant test -Dtestcase=UserControllerEM`. Your console output should resemble the screenshot below.



```
Spring Live (80,20)
New Bookmarks Configure : Customize Close
foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerEM
Buildfile: build.xml

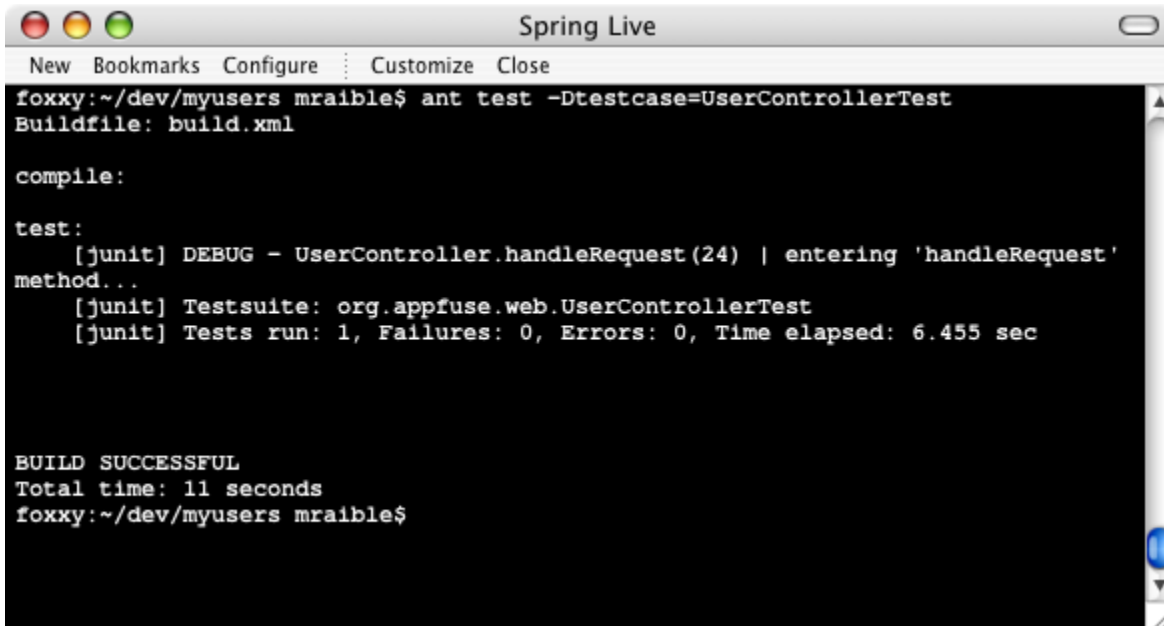
compile:

test:
[junit] DEBUG - UserController.handleRequest(24) | entering 'handleRequest'
method...
[junit] Testsuite: org.appfuse.web.UserControllerEMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.223 sec

BUILD SUCCESSFUL
Total time: 5 seconds
foxy:~/dev/myusers mraible$
```

Figure 8.4: Results of the `ant test -Dtestcase=UserControllerEM` test

This test runs *much* faster than the previous one. To compare, run `ant test -Dtestcase=UserControllerTest` and compare the "Time elapsed" value. On my 1.33Mhz/512RAM PowerBook, there's quite a difference: 0.223 seconds for the mocked test and 6.455 seconds for the non-mock version.



```

Spring Live
New Bookmarks Configure Customize Close
foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserController.handleRequest(24) | entering 'handleRequest'
method...
[junit] Testsuite: org.appfuse.web.UserControllerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 6.455 sec

BUILD SUCCESSFUL
Total time: 11 seconds
foxy:~/dev/myusers mraible$

```

Figure 8.5: Showing the time elapsed for the non-mocked version

You've created an EasyMock version of `UserControllerTest`; now create a jMock version. Create a new `UserControllerJMockTest.java` class in `test/org/appfuse/web`. This class extends jMock's `MockObjectTestCase`.

```

package org.appfuse.web;

// use your IDE to organize imports

public class UserControllerJMockTest extends MockObjectTestCase {
    private UserController c = new UserController();
    private Mock mockManager = null;

    protected void setUp() throws Exception {
        mockManager = new Mock(UserManager.class);
        c.setUserManager((UserManager) mockManager.proxy());
    }

    public void testGetUsers() throws Exception {
        // set expected behavior on manager
        mockManager.expects(once()).method("getUsers")

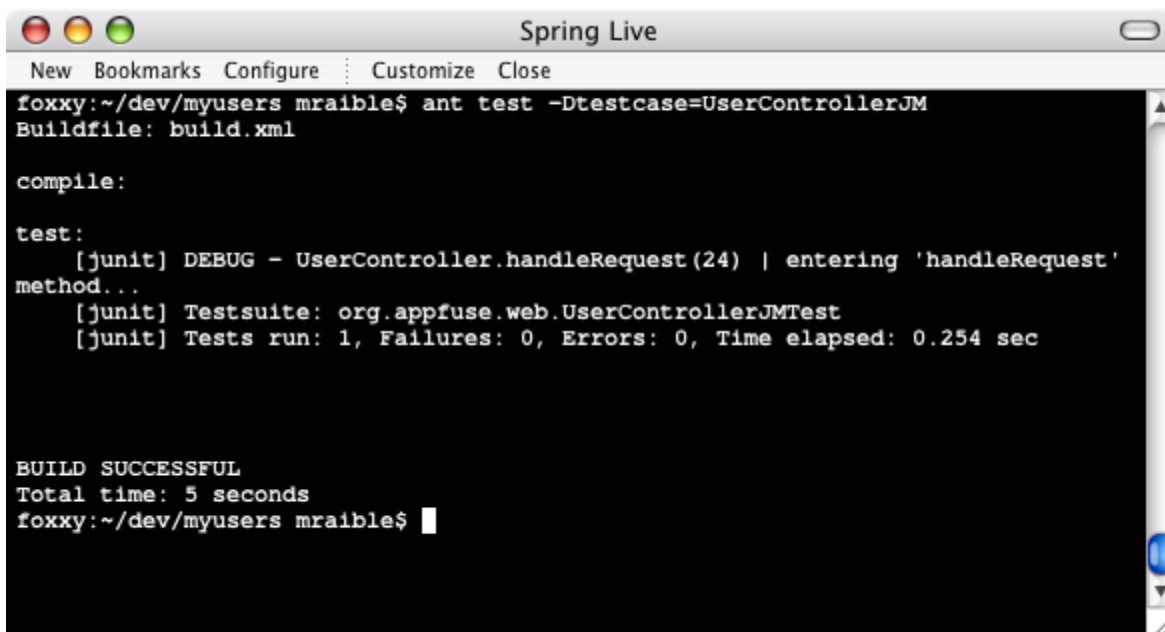
```

```
        .will(returnValue(new ArrayList()));

    ModelAndView mav =
        c.handleRequest((HttpServletRequest) null,
            (HttpServletResponse) null);
    Map m = mav.getModel();
    assertNotNull(m.get("users"));
    assertEquals(mav.getViewName(), "userList");

    // verify expectations
    mockManager.verify();
}
}
```

Run this test using `ant test -Dtestcase=UserControllerJM`.

A screenshot of a terminal window titled "Spring Live". The window has a menu bar with "New", "Bookmarks", "Configure", "Customize", and "Close". The terminal shows the command `foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerJM` and its output. The output includes "Buildfile: build.xml", "compile:", "test:", and detailed JUnit test results for `org.appfuse.web.UserControllerJMTest`, showing 1 test run, 0 failures, 0 errors, and a time elapsed of 0.254 seconds. The terminal concludes with "BUILD SUCCESSFUL", "Total time: 5 seconds", and the prompt `foxy:~/dev/myusers mraible$`.

```
foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerJM
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserController.handleRequest(24) | entering 'handleRequest'
method...
[junit] Testsuite: org.appfuse.web.UserControllerJMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.254 sec

BUILD SUCCESSFUL
Total time: 5 seconds
foxy:~/dev/myusers mraible$
```

Figure 8.6: Results of the `ant test -Dtestcase=UserControllerJM` test

Testing FormControllers

Testing form classes, which extend [SimpleFormController](#), is much like testing classes that implement [Controller](#). The major difference is that FormControllers do more; for example, they validate data, set messages and use request parameters. The test for `UserFormController` (from *Chapter 4*) is listed below. It verifies that no validation errors occur in its `testSave()` method, and that a success message is set.

```
package org.appfuse.web;

// use your IDE to organize imports

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

    protected void setUp() throws Exception {
        super.setUp();
        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 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"));
    }
}

```

While this test works, it takes several seconds to run (10.2 on my PowerBook!). Refactoring it to remove Spring dependencies and use jMock will speed things up dramatically.

To refactor this class, create a new `UserFormControllerJMTest.java` class in the `test/org/appfuse/web` directory. This class extends `MockObjectTestCase` and contains the following code:

```

package org.appfuse.web;

// use your IDE to organize imports

public class UserFormControllerJMTest extends MockObjectTestCase {
    private static Log log =
        LoggerFactory.getLog(UserFormControllerJMTest.class);
    private UserFormController c = new UserFormController();
    private MockHttpServletRequest request = null;
    private ModelAndView mv = null;
    private User user = new User();
    private Mock mockManager = null;

    protected void setUp() throws Exception {
        super.setUp();
        mockManager = new Mock(UserManager.class);

        // manually set properties (dependencies) on userFormController
        c.setUserManager((UserManager) mockManager.proxy());
        c.setFormView("userForm");

        // set context with messages avoid NPE when controller calls
        // getMessageSourceAccessor().getMessage()
        StaticApplicationContext ctx = new StaticApplicationContext();
    }
}

```



```
Map properties = new HashMap();
properties.put("basename", "messages");
ctx.registerSingleton("messageSource",
    ResourceBundleMessageSource.class,
    new MutablePropertyValues(properties));

ctx.refresh();
c.setApplicationContext(ctx);

// setup user values
user.setId(new Long(1));
user.setFirstName("Matt");
user.setLastName("Raible");
}

public void testEdit() throws Exception {
    log.debug("testing edit...");

    // set expected behavior on manager
    mockManager.expects(once()).method("getUser")
        .will(returnValue(new User()));

    request = new MockHttpServletRequest("GET", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertEquals("userForm", mv.getViewName());
    // The getCommandName() method is available in Spring 1.1.1+
    User editUser = (User) mv.getModel().get(c.getCommandName());
    assertEquals(editUser.getFullName(), "Matt Raible");

    // verify expectations
    mockManager.verify();
}

public void testSave() throws Exception {
    // set expected behavior on manager
    // called by formBackingObject()
    mockManager.expects(once()).method("getUser")
        .will(returnValue(user));

    User savedUser = user;
    savedUser.setLastName("Updated Last Name");
    // called by onSubmit()
    mockManager.expects(once()).method("saveUser")
        .with(eq(savedUser));

    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"));

        // verify expectations
        mockManager.verify();
    }

    public void testRemove() throws Exception {
        // set expected behavior on manager
        // called by formBackingObject()
        mockManager.expects(once()).method("getUser")
            .will(returnValue(user));
        // called by onSubmit()
        mockManager.expects(once()).method("removeUser").with(eq("1"));

        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"));

        // verify expectations
        mockManager.verify();
    }
}

```

In the above class's `setUp()` method, Spring's [StaticApplicationContext](#) provides a convenient way to add and use beans in unit tests. Its `registerSingleton()` method registers the `messages.properties` file for accessing messages. Without doing this, it would throw a `NullPointerException` when the `FormController` tries to access the "messageSource" bean. You could also use the [StaticMessageSource](#) class for adding messages programmatically in your tests. More information on using these classes in your tests is available in Spring's [StaticApplicationContextTestSuite](#).



NOTE

Spring's [internal tests](#) are an excellent source of information on how to write your own unit tests.

Cactus

This section uses Cactus to test Controllers in their deployed environment. [Cactus](#) is an extension of JUnit for unit testing server-side Java code.

To use Cactus for testing Controllers, modify MyUser's *build.xml* by adding a "test-cactus" target.

```
<target name="test-cactus" depends="war"
    description="Runs Cactus tests in-container">

    <!-- Define Cactus Tasks -->
    <taskdef resource="cactus.tasks" classpathref="classpath"/>

    <cactifywar srcfile="${dist.dir}/${webapp.name}.war"
        destfile="${dist.dir}/${webapp.name}-cactus.war">
        <classes dir="${test.dir}/classes"/>
        <classes dir="test" includes="cactus.properties"/>
        <!-- If you use EasyMock or Spring Mocks in a Cactus test
            it needs to be included in the WAR -->
        <lib dir="web/WEB-INF/lib" includes="*mock.jar"/>
        <servletredirector/>
    </cactifywar>

    <mkdir dir="${test.dir}/data/tomcat"/>

    <cactus warfile="${dist.dir}/${webapp.name}-cactus.war"
        printsummary="yes" failureproperty="tests.failed">
        <classpath>
            <path refid="classpath"/>
            <path location="${build.dir}/classes"/>
            <path location="${test.dir}/classes"/>
        </classpath>
        <containerset>
            <tomcat5x dir="${tomcat.home}" if="tomcat.home"
                port="8080" todir="${test.dir}/data/tomcat"/>
        </containerset>
        <formatter type="xml"/>
        <formatter type="brief" usefile="false"/>
        <batchtest todir="${test.dir}/data" if="testcase">
            <fileset dir="${test.dir}/classes">
                <include name="**/*${testcase}*" />
                <exclude name="**/*TestCase.class" />
            </fileset>
        </batchtest>
        <batchtest todir="${test.dir}/data" unless="testcase">
            <fileset dir="${test.dir}/classes">
                <include name="**/*Cactus*Test.class*" />
            </fileset>
        </batchtest>
    </cactus>
</target>
```

```

        </batchtest>
    </cactus>
    <fail if="tests.failed">Cactus test(s) failed.</fail>
</target>

```

The `<cactifywar>` task at the beginning of this listing is responsible for modifying the *web.xml* and WAR files so Cactus can test it. A `<tomcat5x>` task is in the middle of this target. This task specifies the container in which to run the tests. See Cactus's project site for [more information on its Ant Integration](#) and the [list of supported containers](#) for the `<cactus>` task.

Now that you've set up Cactus, create a `UserCactusTest.java` class in *test/org/appfuse/web*. This class extends `ServletTestCase`. This first version manually creates the `UserController` and `UserFormController`, then sets their necessary dependencies (`userManager` and context for "messageSource" bean). The `ApplicationContext` for this test is retrieved from the `ServletContext`, not initialized from the `ClassPathXmlApplicationContext`. This is because the test will be running in the context of the application, and Spring's application context is initialized by the `ContextLoaderListener` in *web.xml*.

```

package org.appfuse.web;

// use your IDE to organize imports

public class UserCactusTest extends ServletTestCase {
    private static Log log = LogFactory.getLog(UserCactusTest.class);
    private UserController list = new UserController();
    private UserFormController form = new UserFormController();

    protected void setUp() throws Exception {
        super.setUp();
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        UserManager userManager =
            (UserManager) ctx.getBean("userManager");
        list.setUserManager(userManager);
        form.setUserManager(userManager);
        // needed to prevent NPE with getMessageSourceAccessor()
        form.setApplicationContext(ctx);
    }

    public void beginAddUser(WebRequest wRequest) {
        wRequest.addParameter("firstName", "Dion", "post");
        wRequest.addParameter("lastName", "Almaer", "post");
    }
}

```

```

    public void testAddUser() throws Exception {
        form.handleRequest(request, response);
        assertTrue(request.getSession().getAttribute("message") != null);
    }

    public void testUserList() throws Exception {
        ModelAndView mav = list.handleRequest(request, response);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
    }
}

```

Create a *cactus.properties* file in the *test* directory. Cactus requires the following information to run its servlet tests:

```

# Web app Context under which our application to test runs
cactus.contextURL=http://localhost:8080/myusers-cactus

# Default Servlet Redirector Name. Used by ServletTestCase test cases.
cactus.servletRedirectorName=ServletRedirector

```

Run this test by executing **ant test-cactus -Dtestcase=UserCactusTest**.

Another way to write this test is to grab the Controllers directly from the *ApplicationContext*, where their dependencies are already set.

```

...
public class UserCactusTest extends ServletTestCase {
    private static Log log = LogFactory.getLog(UserCactusTest.class);
    private UserController list = null;
    private UserFormController form = null;

    protected void setUp() throws Exception {
        super.setUp();
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        list = (UserController) ctx.getBean("userController");
        form = (UserFormController) ctx.getBean("userFormController");
    }

    public void beginAddUser(WebRequest wRequest) {
        ...
    }
}

```

Finally, if you choose, you can mock the `userManager` with `EasyMock`. The `UserCactusEMTest` class below illustrates this technique. You *cannot* use `jMock` with Cactus since both of them require you to extend their `TestCase` classes.

```
package org.appfuse.web;

// user your IDE to organize imports

public class UserCactusEMTest extends ServletTestCase {
    private UserController list = new UserController();
    private UserFormController form = new UserFormController();
    private MockControl control = null;
    private UserManager mockManager = null;

    protected void setUp() throws Exception {
        control = MockControl.createControl(UserManager.class);
        mockManager = (UserManager) control.getMock();
        list.setUserManager(mockManager);
        form.setUserManager(mockManager);

        // needed to prevent NPE with getMessageSourceAccessor()
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        form.setApplicationContext(ctx);
    }

    public void beginAddUser(WebRequest wRequest) {
        wRequest.addParameter("firstName", "Dion", "post");
        wRequest.addParameter("lastName", "Almaer", "post");
    }

    public void testAddUser() throws Exception {
        // set expected behavior on manager
        User user = new User();
        user.setFirstName("Dion");
        user.setLastName("Almaer");
        mockManager.saveUser(user);
        control.setReturnValue(user);

        // switch from record to playback
        control.replay();

        form.handleRequest(request, response);

        // verify saveUser() was called
        control.verify();
        assertTrue(request.getSession())
```

```
        .getAttribute("message") != null);
    }

    public void testUserList() throws Exception {
        // set expected behavior on manager
        control.expectAndReturn(mockManager.getUsers(),
                                new ArrayList());
        // switch from record to playback
        control.replay();

        ModelAndView mav = list.handleRequest(request, response);

        // verify getUsers() was called
        control.verify();

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

To run this test, execute **ant test-cactus -Dtestcase=UserCactusEM**.

The second approach (grab controllers directly) is usually the best approach. Running Cactus tests requires quite a bit of time since it has to parse/modify the WAR and start/stop the container; therefore, using mocks is unlikely to increase your test performances. However, they are still helpful for isolating your test in the container.

Testing File Upload and E-Mail

As promised in *Chapter 5*, you will now learn how to test the `FileUploadController` class. This class is responsible for uploading files and sending a notification e-mail.

1. In `test/org/appfuse/web`, create a `FileUploadControllerTest` that extends JUnit's `TestCase`. The code below represents the contents of this class with a `setUp()` method to initialize the context files:

```
package org.appfuse.web;

// organize imports using your IDE

public class FileUploadControllerTest extends TestCase {
    private static Log log =
        LogFactory.getLog(FileUploadControllerTest.class);
    private XmlWebApplicationContext ctx = null;
    private FileUploadController fileUpload = null;

    public void setUp() {
        String[] paths = {"/WEB-INF/applicationContext*.xml",
                        "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ServletContext servletContext =
            new MockServletContext("file:web");

        ctx.setServletContext(servletContext);
        ctx.refresh();
        fileUpload = (FileUploadController)
            ctx.getBean("fileUploadController");

        // Create a mailSender that'll use Dumbster's ports
        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
        mailSender.setHost("localhost");
        mailSender.setPort(2525);
        fileUpload.setMailSender(mailSender);
    }

    // continued below
}
```

This `setUp()` method contains unique configurations. First, the `MockServletContext` is initialized using `"file:web"`. A [ResourceLoader](#) class uses this value to determine the root of the web application. Without this, the following variable assignment in `FileUploadController` will fail:

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


Second, `setUp()` contains a custom [MailSender](#) bean that sends e-mail on a non-standard port (2525; 25 is standard). This is so you can use [Dumbster](#). You will use Dumbster to start/stop an SMTP server on port 2525 before and after running the `FileUploadControllerTest`.



NOTE

This example uses port 2525 to avoid conflicts with a server that might be running on port 25.

2. Finish the test by adding the `testUpload()` method below:

```
public void testUpload() throws Exception {
    log.debug("testing upload...");
    MockHttpServletRequest request =
        new MockHttpServletRequest("POST", "/fileUpload.html");

    MockCommonsMultipartResolver resolver =
        new MockCommonsMultipartResolver();
    ctx.getServletContext();

    request.setContentType("multipart/form-data");
    request.addHeader("Content-type", "multipart/form-data");
    assertTrue(resolver.isMultipart(request));
    MultipartHttpServletRequest multipartRequest =
        resolver.resolveMultipart(request);

    // setup a simple mail server using Dumbster
    SimpleSmtpServer server = SimpleSmtpServer.start(2525);

    ModelAndView mav =
        fileUpload.handleRequest(multipartRequest,
                                new MockHttpServletResponse());

    server.stop();
    // the getReceieved() method is spelled wrong in the API. ;-)
    assertEquals(1, server.getReceievedEmailSize());

    log.debug("model: " + mav.getModel());

    assertNotNull(request.getSession().getAttribute("message"));

    // ensure the file got uploaded
    Resource uploadedFile =
        ctx.getResource("file:web/upload/test.xml");
    assertTrue(uploadedFile.exists());

    // delete the upload directory
    Resource uploadDir = ctx.getResource("file:web/upload");
    uploadedFile.getFile().delete();
}
```

```

        uploadDir.getFile().delete();
        assertFalse(uploadDir.exists());
    }

    public static class MockCommonsMultipartResolver extends
        CommonsMultipartResolver {

        private boolean empty;

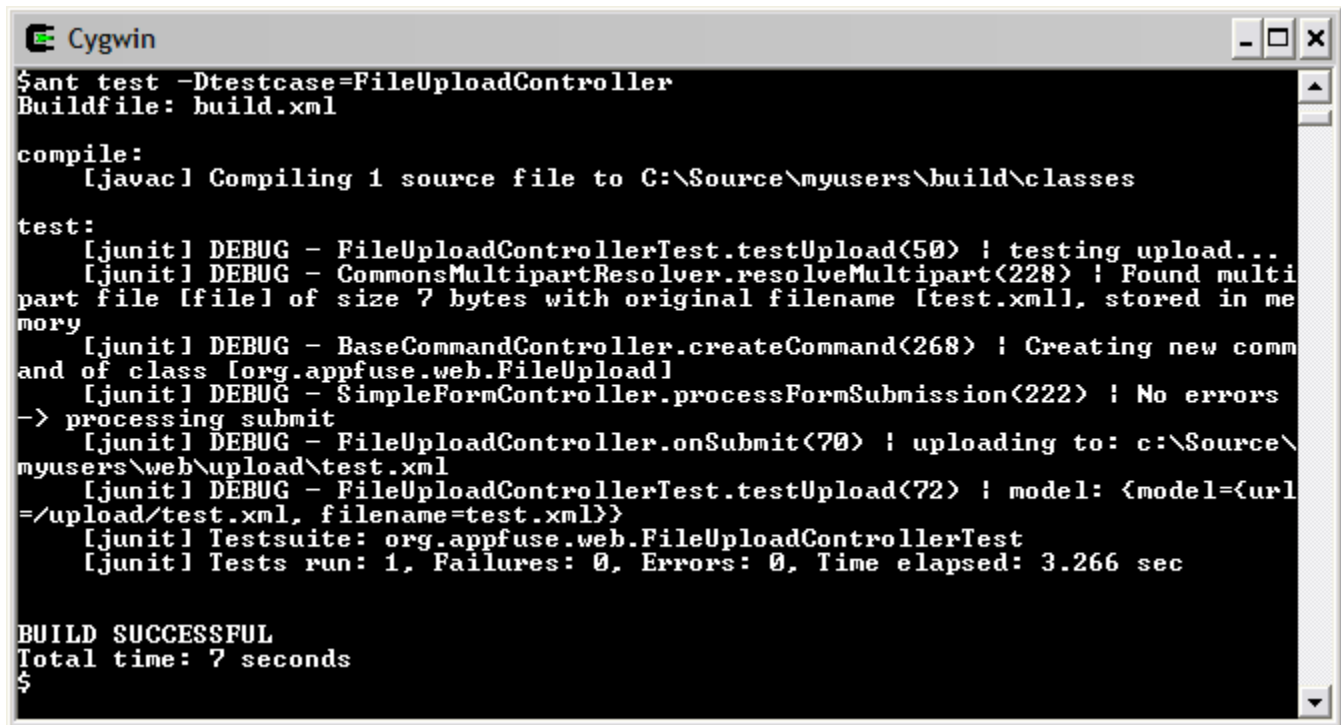
        protected void setEmpty(boolean empty) {
            this.empty = empty;
        }

        protected DiskFileUpload newFileUpload() {
            return new DiskFileUpload() {
                public List parseRequest(HttpServletRequest request) {
                    if (request instanceof MultipartHttpServletRequest) {
                        throw new IllegalStateException(
                            "Already a multipart request");
                    }
                    List fileItems = new ArrayList();
                    MockFileItem fileItem = new MockFileItem(
                        "file", "text/html", empty ? "" :
                        "test.xml", empty ? "" : "<root/>");
                    fileItems.add(fileItem);
                    return fileItems;
                }
            };
        }
    }
}

```

The `MockCommonsMultipartResolver` class in this test refers to a `MockFileItem` class. This class mimics an uploaded file. The `MockFileItem.java` file is included in the download for this chapter; it's also available in Spring's [CommonsMultipartResolverTests](#) class. You can implement it as an inner class like the Spring test does; this example extracts it out in the interest of saving space.

3. Run this test using `ant test -Dtestcase=FileUploadController`. The test's output should resemble the screenshot below.



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

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

test:
[junit] DEBUG - FileUploadControllerTest.testUpload<50> ! testing upload...
[junit] DEBUG - CommonsMultipartResolver.resolveMultipart<228> ! Found multi
part file [file] of size 7 bytes with original filename [test.xml], stored in me
mory
[junit] DEBUG - BaseCommandController.createCommand<268> ! Creating new comm
and of class [org.appfuse.web.FileUpload]
[junit] DEBUG - SimpleFormController.processFormSubmission<222> ! No errors
-> processing submit
[junit] DEBUG - FileUploadController.onSubmit<70> ! uploading to: c:\Source\
myusers\web\upload\test.xml
[junit] DEBUG - FileUploadControllerTest.testUpload<72> ! model: {model={url
=/upload/test.xml, filename=test.xml}}
[junit] Testsuite: org.appfuse.web.FileUploadControllerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 3.266 sec

BUILD SUCCESSFUL
Total time: 7 seconds
$
```

Figure 8.7: Results of the `ant test -Dtestcase=FileUploadController` test

From all of the examples in this section, you can see that many options for testing Controllers are available. The simplest way to write Controller tests is to initialize a context in your test class, retrieve your Controllers (using `ctx.getBean("controllerName")`) and execute methods on them. This technique allows Spring to wire the Controller's dependencies. Using mocks requires a little more work because you must know your Controller's dependencies. You also must be aware what methods will be called on your mocked object. The advantage is you'll know exactly what your Controller depends on and how it interacts with those dependencies.

Testing Views

In *Chapter 6*, you created a [jWebUnit](#) test (`UserWebTest`) that interacted with links and buttons to verify the functionality of the UI. `jWebUnit` is a nice framework for testing views because it allows you to write your tests in Java. This section modifies the `"test-web"` target in *build.xml* so that Tomcat starts and stops before any `jWebUnit` tests run. You will also change the current test to switch locales and verify text against *messages.properties*, rather than hard coding it in the class. Finally, you will use [Canoo WebTest](#) as an alternative method of testing the UI. `WebTest` is friendlier to non-programmers because tests are written in an Ant build file using XML.

The tests in this section are commonly referred to as *integration tests*. They verify that all the layers are integrated properly by navigating the UI as a user would in a browser. Both `jWebUnit` and `Canoo WebTest` support JavaScript testing. This means that if you have an "onclick" handler on a button, and you specify to click this button, the JavaScript call will execute. However, this JavaScript support is quite flakey and reports errors in JavaScript when there are none. For this reason, I advocate 1) developing your UIs so they don't require JavaScript to run and 2) writing your tests to work around any UIs that have JavaScript in them. This means if the URL calls `<button onclick="addPage()" />` by the `addPage()` function, it can simply be called in the test.

jWebUnit

Before you change the `UserWebTest` in `MyUsers`, review its source. The constructor should always contain a call to `setBaseUrl()`. Since `jWebUnit` refers to submit buttons by name, you must specify a `name` attribute on your buttons. If you want to test tables and their data, you must specify an `id` attribute on your tables. `MyUsers` should already contain all the necessary names and ids.

```
package org.appfuse.web;

// use your IDE to organize imports

public class UserWebTest extends WebTestCase {

    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
    }

    public void testWelcomePage() {
        beginAt("/");
        assertEquals("MyUsers ~ Welcome");
    }

    public void testAddUser() {
        beginAt("/editUser.html");
    }
}
```

```

        assertEquals("MyUsers ~ User Details");
        setFormElement("firstName", "Spring");
        setFormElement("lastName", "User");
        submit("save");
        assertTextPresent("saved successfully");
    }

    public void testListUsers() {
        beginAt("/users.html");

        // check that table is present
        assertTablePresent("userList");

        //check that a set of strings are present somewhere in table
        assertTextInTable("userList",
            new String[] {"Spring", "User"});
    }

    public void testEditUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertFormElementEquals("firstName", "Spring");
        submit("save");
        assertEquals("MyUsers ~ User List");
    }

    public void testDeleteUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertEquals("MyUsers ~ User Details");
        submit("delete");
        assertEquals("MyUsers ~ User List");
    }

    /**
     * Convenience method to get the id of the inserted user
     * Assumes last inserted user is "Spring User"
     */
    public String getInsertedUserId() {
        String[] paths = {"WEB-INF/applicationContext*.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(paths);
        List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
        // assumed that user inserted in testAddUser() is last user
        return "" + ((User) users.get(users.size() - 1)).getId();
    }
}

```

In order to internationalize this test, you first must internationalize the titles in MyUsers. Currently, they are hard-coded into their respective templates.

1. Create entries in *messages.properties*:

```
# Page titles
index.title=MyUsers ~ Welcome
userList.title=MyUsers ~ User List
userForm.title=MyUsers ~ User Details
```

2. The download for this chapter uses Velocity templates for the view, so you must edit the Velocity templates:

- web/WEB-INF/velocity/userList.vm:

```
<title>${rc.getMessage("userList.title")}</title>
```

- web/WEB-INF/velocity/userForm.vm:

```
<title>${rc.getMessage("userForm.title")}</title>
```

- web/index.vm:

```
<title>${rc.getMessage("index.title")}</title>
```

3. The welcome page (*index.vm*) requires a bit more work to resolve the `rc` variable. In the download for this chapter, *index.vm* is located in the *web* directory and configured as a `welcome-file` in *web.xml*. It's not easy to expose the *messages.properties* file to the [SiteMesh Servlet](#) (which parses *index.vm*); however, you can work around this issue by putting *index.vm* with the rest of the templates and using Paul Tuckey's [UrlRewriteFilter](#) to resolve it as the welcome page. Below are the steps to make this change:

- a. Move *index.vm* to *web/WEB-INF/velocity*.
- b. Add `/index.html` as a URL to the `urlMapper` bean in *web/WEB-INF/action-servlet.xml*.

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

- c. To make this the welcome file, use the `UrlRewriteFilter`. To configure it, declare it as a filter in *web/WEB-INF/web.xml*:

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

- d. After the sitemesh mapping, add a filter-mapping for this filter:

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

- e. Change the *urlrewrite.xml* file in *web/WEB-INF* to have the following XML:

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



NOTE

Use these same steps to alter the FreeMarker templates. For JSP, use JSTL's `<fmt:message>` tag in the `<title>` tags.

At this point, start Tomcat and run **ant deploy test-web** to verify that this change worked. After verifying that it worked, modify this class to take advantage of jWebUnit's internationalization capabilities. Namely, it has the ability to set a `ResourceBundle` and verify titles and text based on key names, rather than text values. More information on this feature and others is available in [jWebUnit's QuickStart Guide](#).

The main difference between the class below and the previous un-internationalized class is that the `testAddUser()` method verifies the resulting page's title, rather than the success message text. This is because the success message gets the new user's name substituted into its final output and, therefore, the key's value is different from the value displayed. [A patch for jWebUnit](#) to allow testing keys with dynamic values is available.

Changed code is underlined below, and previous assertions are commented out.

```
package org.appfuse.web;

import java.util.List;
import java.util.Locale;

import net.sourceforge.jwebunit.WebTestCase;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.appfuse.dao.UserDAO;
import org.appfuse.model.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserWebTest extends WebTestCase {

    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
        getTestContext().setResourceBundleName("messages");
        getTestContext().setLocale(Locale.ENGLISH);
    }

    public void testWelcomePage() {
        beginAt("/");
        //assertTitleEquals("MyUsers ~ Welcome");
        assertTitleEqualsKey("index.title");
    }

    public void testAddUser() {
        beginAt("/editUser.html");
        assertTitleEquals("MyUsers ~ User Details");
        setFormElement("firstName", "Spring");
        setFormElement("lastName", "User");
        submit("save");
        //assertTextPresent("saved successfully");
        assertTitleEqualsKey("userList.title");
    }

    public void testListUsers() {
        beginAt("/users.html");

        // check that table is present
        assertTablePresent("userList");

        //check that a set of strings are present somewhere in table
    }
}
```



```

        assertTextInTable("userList",
                           new String[] { "Spring", "User" });
    }

    public void testEditUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertFormElementEquals("firstName", "Spring");
        submit("save");
        //assertTitleEquals("MyUsers ~ User List");
        assertTitleEqualsKey("userList.title");
    }

    public void testDeleteUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        //assertTitleEquals("MyUsers ~ User Details");
        assertTitleEqualsKey("userForm.title");
        submit("delete");
        //assertTitleEquals("MyUsers ~ User List");
        assertTitleEqualsKey("userList.title");
    }

    /**
     * Convenience method to get the id of the inserted user
     * Assumes last inserted user is "Spring User"
     */
    public String getInsertedUserId() {
        String[] paths = { "/WEB-INF/applicationContext*.xml" };
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(paths);
        List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
        // assumed that user inserted in testAddUser() is last user
        return "" + ((User) users.get(users.size() - 1)).getId();
    }
}

```

To test this class using a different locale (and *messages.properties* file), follow the steps below:

1. Duplicate the *messages.properties* file and name it *message_de.properties*. Change some of the title values so you know it's using the new message bundle.
2. Change the `setLocale()` call in the constructor to set the locale to German.

```
getTestContext().setLocale(Locale.GERMAN);
```

This will load the German messages file, but it will *not* set the locale for the request. This has been [reported as a bug in jWebUnit](#). To work around this bug, set the "Accept-Language" header as part of the underlying HttpUnit WebClient.

```
getTestContext().getWebClient()
    .setHeaderField("Accept-Language", "de");
```

3. Run the test using **ant deploy reload test-web**.

Automate Starting Tomcat

The current setup for running jWebUnit tests requires you to start Tomcat before running the tests. To alleviate this pain, add a new target to *build.xml* that automates the startup and shutdown of Tomcat before and after running the test. [Cargo](#) is an open-source project started by Vincent Massol (who also created Cactus). It provides a Java API and Ant Tasks to start/stop and configure Java containers. Since the Cargo JAR is already included in this chapter's download, you must simply add the following target to *build.xml*:

```
<target name="test-tomcat" depends="war"
  description="Runs jWebUnit tests in a running server">
  <taskdef resource="cargo.tasks" classpathref="classpath"/>
  <cargo-tomcat5x homeDir="${tomcat.home}"
    output="${test.dir}/cargo.log"
    workingDir="${test.dir}/tomcat5x" action="start">
    <war warFile="${dist.dir}/${webapp.name}.war"/>
  </cargo-tomcat5x>
  <antcall target="test-web"/>
</target>
```

Now you should be able to stop Tomcat and run **ant test-tomcat** to run the `UserWebTest` in Tomcat without manually starting it.



NOTE

Cargo is a very new project and the JAR used in MyUsers is an alpha release from early September 2004. A few things still need improvement, like allowing the startup/shutdown log messages to be printed to the console. Currently, they can only be directed to the file specified in the "output" attribute.

Canoo WebTest

[Canoo WebTest](#) is a free, open-source tool for automating web application tests. It's similar to jWebUnit, except you write its tests using XML and execute them using Ant. One nice feature is the ability to test PDFs and their contents. In order to add Canoo WebTest tests to MyUsers, complete the following steps:

1. Add links to the *userList.vm* page in *web/WEB-INF/velocity* so that you can bring up a PDF by just clicking on the **PDF** link. Put the following HTML just after the **Add** button.

```
<p style="text-align: right; margin-bottom: -10px">
<strong>Export Options:</strong>
  <a href="?report=XML">XML</a> &middot;
  <a href="?report=Excel">Excel</a> &middot;
  <a href="?report=PDF">PDF</a>
</p>
```

2. Add a "test-canoo" target to *build.xml*.

```
<target name="test-canoo" depends="deploy"
  description="Runs Canoo WebTests in Tomcat to test JSPs">

  <taskdef file="webtestTasks.properties">
    <classpath>
      <path refid="classpath"/>
      <!-- for log4j.xml -->
      <path location="web/WEB-INF/classes"/>
    </classpath>
  </taskdef>

  <mkdir dir="${test.dir}/data"/>
  <!-- Delete old results file if it exists -->
  <delete file="${test.dir}/data/web-tests-result.xml"/>

  <property name="testcase" value="run-all-tests"/>
  <ant antfile="test/web-tests.xml" target="${testcase}"/>
</target>
```

3. Create a *config.xml* file in the *test* directory and add the following XML to it:

```
<config host="localhost" port="8080"
  protocol="http" basepath="${webapp.name}"
  resultpath="${test.dir}/data"
  resultfile="web-tests-result.xml"
  summary="true" saveresponse="true"/>
```

4. Create a *web-tests.xml* file in the *test* directory with the following structure:

```
<!DOCTYPE project [
    <!ENTITY config SYSTEM "file:./config.xml">
]>
<project basedir="." default="run-all-tests">
    <!-- Include messages.properties so we can test against
         keys, rather than values -->
    <property file="web/WEB-INF/classes/messages.properties"/>

    <!-- runs all targets -->
    <target name="run-all-tests" depends="UserTests"
        description="Call and executes all test cases (targets)">
        <echo>All UserTests passed!</echo>
    </target>

    <!-- Verify adding a user, viewing and deleting a user works -->
    <target name="UserTests"
        description="Adds a new user profile">
        <canoo name="userTests">
            &config;
            <steps>
                <!-- View add screen -->
                <invoke url="/editUser.html"/>
                <verifytitle text="${userForm.title}"/>
                <!-- Enter data and save -->
                <setinputfield name="firstName" value="Test"/>
                <setinputfield name="lastName" value="Name"/>
                <clickbutton label="Save"/>
                <!-- View user list -->
                <verifytitle text="${userList.title}"/>
                <!-- Verify PDF contains new user -->
                <clicklink label="PDF"/>
                <verifyPdfText text="Test Name"/>
                <!-- Delete first user in table -->
                <invoke url="/users.html"/>
                <clicklink href="editUser.html"/>
                <verifytitle text="${userForm.title}"/>
                <clickbutton label="Delete"/>
                <verifytitle text="${userList.title}"/>
            </steps>
        </canoo>
    </target>
</project>
```

This file's "UserTests" target contains the tasks to test the various operations on the MyUsers UI. In the middle of the target is a call to bring up the PDF and verify the newly added user.



WARNING

If the rendering of the PDF doesn't work for your application, make sure the "viewResolver2" bean in *action-servlet.xml* contains the following property:

```
<property name="order"><value>1</value></property>
```

If you get errors about log4j, make sure the following XML is in your *log4j.xml* file (in *web/WEB-INF/classes*):

```
<logger name="com.canoo">
  <level value="WARN"/>
</logger>
```

5. Run **ant deploy**, start Tomcat and then run **ant test-canoo**. All tests should pass and you will see a console output similar to Figure 8.8.

```
Cygwin
$ant test-canoo
Buildfile: build.xml

compile:
deploy:
test-canoo:
  [delete] Deleting: C:\Source\myusers\build\test\data\web-tests-result.xml
UserTests:
  [canoo] Rhino classes <js.jar> not found - Javascript disabled
run-all-tests:
  [echo] All UserTests passed!

BUILD SUCCESSFUL
Total time: 13 seconds
$_
```

Figure 8.8: Results of the `ant test-canoo` test

Automate Starting Tomcat

Just like with jWebUnit, it's easy to automate the startup/shutdown of Tomcat before and after running Canoo tests. Use the Cargo API again, and add the following "test-view" target to *build.xml*.

```
<target name="test-view" depends="war"
  description="Runs canoo tests in a running server">
  <taskdef resource="cargo.tasks" classpathref="classpath"/>
  <cargo-tomcat5x homeDir="${tomcat.home}"
    output="${test.dir}/cargo.log"
    workingDir="${test.dir}/tomcat5x" action="start">
    <war warFile="${dist.dir}/${webapp.name}.war"/>
  </cargo-tomcat5x>
  <antcall target="test-canoo"/>
</target>
```

If Tomcat isn't running, you should be able to test this target by running **ant test-view**.



TIP

The Canoo WebTest team keeps a [weblog of tips](#) related to this project.

From this brief overview of jWebUnit and Canoo's WebTest, I hope you learned enough about how each one works to use in your projects. Many developers don't know it's possible to automate UI tests, and both of these tools can save you a lot of time. Of course, you must still often look at your UI in a browser to tweak colors, fonts and position.

Summary

This chapter covered a lot of information on how to test Spring, as well as how to write unit tests in general. You learn how to use JUnit, DbUnit, EasyMock, jMock, Spring Mocks, Cactus, jWebUnit and Canoo's WebTest. For the database layer, you saw how it's best to talk directly to a running database and verify that your DAOs are operating as they should. It described techniques for overriding beans for tests (for example, to wrap transactions around a DAO implementation).

In the business façade layer, demonstrations were given of using EasyMock and jMock, both of which greatly simplify *mocking* objects. The best strategy for deciding what to mock is to never mock APIs that aren't yours. Since the `UserDAO` interface and its implementation(s) are classes you've created, it makes sense to mock it in the `UserManagerTest` unit test. Using mocks provides a nice way to isolate a test and, in most cases, makes your tests run much faster.

In the web layer, you learned how to use jWebUnit to write a Java-based test for verifying the UI's features. Canoo's WebTest demonstrated the same testing functionality, but with Ant/XML instead of Java. The new Cargo project showed how you can easily start/stop Tomcat as part of running your tests.

This chapter did not cover testing rich clients (for example, SWT or Swing applications). This is mainly because the Spring Rich has not been released at the time of this writing. Also, there seems to be a lack of tools for testing these types of applications.

Testing is an important part of developing any software. This chapter has provided you with the tools and techniques you need to use TDD in your projects to produce high-quality software.

AOP

What is AOP and How Can Spring Make It Easier?

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 how AOP might help you.

Overview

If you're active in the Java Community by reading weblogs, magazines or TheServerSide.com, you've probably heard of Aspect-Oriented Programming (AOP). While AOP has been around for several years, it has only recently gained attention. This is probably because many Java-based frameworks are now available to ease the use of AOP.

The most basic definition of AOP is "it's a way of removing code duplication." Java is an Object-Oriented (OO) language. It allows you to create applications and their objects in a hierarchical fashion. However, it doesn't offer an easy way to remove code duplication among classes that aren't in the same hierarchy. In this chapter, you will learn how to use AOP in your projects. You will modify the MyUsers application to use AOP for controlling logging, caching, transactions and e-mail.

An application typically has two types of concerns: *core concerns* and *crosscutting concerns*. Core concerns relate to application functionality, while crosscutting concerns don't apply to specific classes or modules, but are system-wide. Modules created to manage these crosscutting concerns are known as *aspects* - hence the name *Aspect-Oriented Programming*.

AOP is simply a nice way to modularize crosscutting concern logic from your classes and abstract them into application-wide concerns, such as logging, security, transactions, etc. By writing generic AOP classes, you can easily apply these concerns in other projects. Using Spring's IoC container, you can easily configure these classes, and simply package them as a JAR to include in your projects.

In Spring terms, an aspect is typically an *interceptor* configured to inspect method calls. During inspection, this interceptor can do many things: logging that the method is being called, modifying the returned object(s) or sending notifications. Servlet Filters are a form of AOP that offers the ability to perform pre- and post-processing of servlet invocations.

The Java Community's consensus on AOP seems to be "only factor concerns into AOP that your system can live without." In other words, if it's something vitally important to your application (such as business logic), it should remain in your application's code. This is because AOP configuration and code is not highly visible. With Spring, you must examine an XML file to see what interceptors are applied to which methods. New developers may not be aware of this and spend hours trying to figure out why certain behavior (performed by interceptors) is happening. If you need to see the functionality in your class, it shouldn't be in an aspect.

AOP helps you in terms of what you *should* see vs. what you *shouldn't* see. For example, it's better to have Spring handle transaction-handling code than to see or write it yourself. This works especially well in an environment with junior and senior developers; the experienced developers can hide aspects from the junior developers.

This chapter is for new users of AOP. Entire books are available on AOP, which are beyond the scope of this book. I recommend [AspectJ in Action](#) by Ramnivas Laddad. Its first chapter provides an excellent introduction to AOP. This chapter shows you *how* to use AOP, while keeping the theory and low-level details to a minimum. By showing you how Spring allows you to use AOP, you can concentrate on doing your job, rather than learning everything about AOP.

Getting Started

In this chapter, like previous ones, you can follow along and do the examples as we go. The easiest way to do this is to download the MyUsers *Chapter 9* bundle from <http://sourcebeat.com/downloads>. This project tree is the result of *Chapter 8* exercises. It also contains all the JARs you will need for this chapter in the *web/WEB-INF/lib* directory. You can also use the application you've been developing in previous chapters. If you go this route, download *Chapter 9* JARs from <http://sourcebeat.com/downloads>. Each section has a list of new JARs you may need for each new technology. You can use this chapter as a reference for integrating these principles into your own applications.

Note: If you have any issues downloading or running these examples, e-mail me at matr@sourcebeat.com or enter an issue in the [Spring Live Issue Tracker](#).

Logging Example

In most AOP books and articles, the first example shown is how to use AOP for logging. For debugging purposes, it's sometimes nice to add log messages to the beginning and end of methods. Spring makes this very easy, and it doesn't require any coding (only XML).

Note: In order to apply interceptors to beans, configure them as proxied classes. This is so JDK dynamic proxies can create implementations of interfaces at runtime. This section covers the different AOP implementation options after the logging examples below.

Spring ships with a [TraceInterceptor](#) that you can add as an interceptor to your proxied beans. In MyUsers, the `userManager` bean is already proxied (using [TransactionProxyFactoryBean](#)), so adding an interceptor is easy.

1. Open the *applicationContext.xml* file in *web/WEB-INF* and add a "loggingInterceptor" bean.

```
<bean id="loggingInterceptor"
      class="org.springframework.aop.interceptor.TraceInterceptor"/>
```

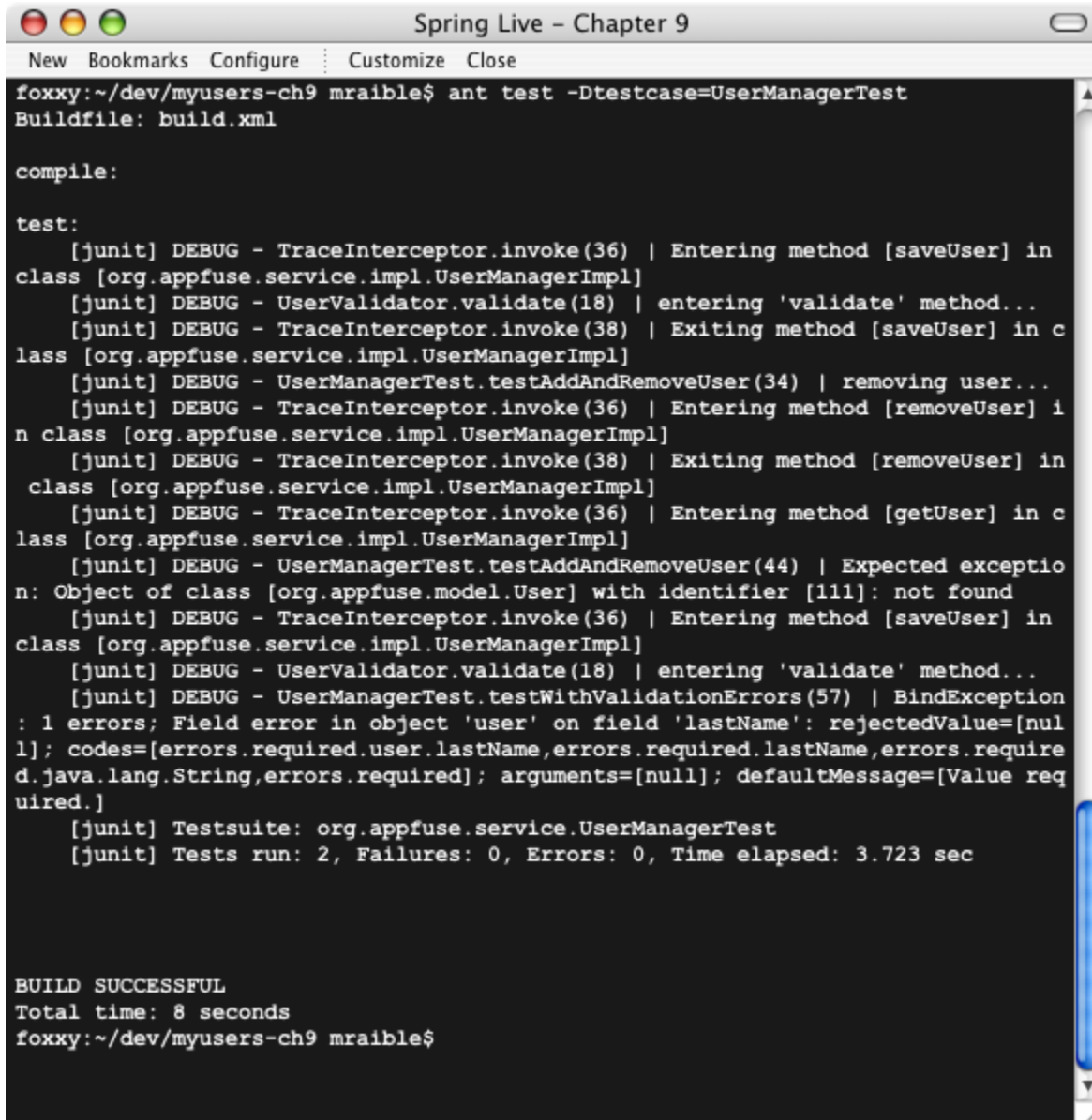
2. Add a "preInterceptors" property to the `userManager` bean with this interceptor.

```
<property name="preInterceptors">
  <list>
    <ref bean="loggingInterceptor"/>
  </list>
</property>
```

3. Add a logger to `web/WEB-INF/classes/log4j.xml` to show log output from `TraceInterceptor`.

```
<logger name="org.springframework.aop.interceptor">
    <level value="DEBUG"/>
</logger>
```

4. Run `ant test -Dtestcase=UserManagerTest`. Your results should resemble the screenshot below.



```
Spring Live - Chapter 9
New Bookmarks Configure Customize Close
foxy:~/dev/myusers-ch9 mraible$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [saveUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserValidator.validate(18) | entering 'validate' method...
[junit] DEBUG - TraceInterceptor.invoke(38) | Exiting method [saveUser] in c
lass [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(34) | removing user...
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [removeUser] i
n class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - TraceInterceptor.invoke(38) | Exiting method [removeUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [getUser] in c
lass [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(44) | Expected exceptio
n: Object of class [org.appfuse.model.User] with identifier [111]: not found
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [saveUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserValidator.validate(18) | entering 'validate' method...
[junit] DEBUG - UserManagerTest.testWithValidationErrors(57) | BindException
: 1 errors; Field error in object 'user' on field 'lastName': rejectedValue=[nul
l]; codes=[errors.required.user.lastName,errors.required.lastName,errors.require
d.java.lang.String,errors.required]; arguments=[null]; defaultMessage=[Value req
uired.]
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 3.723 sec

BUILD SUCCESSFUL
Total time: 8 seconds
foxy:~/dev/myusers-ch9 mraible$
```

Figure 9.1: Results of the `ant test -Dtestcase=UserManagerTest` test

If you look at the [source of TraceInterceptor](#), you can see that how simple AOP can be. The [org.springframework.aop.interceptor](#) package contains a number of other interceptors that you may find useful in your projects.

This simple example is a brief introduction to AOP. Before diving into more intricate examples, it's important to be familiar with AOP terminology.

Definitions and Concepts

AOP, much like other programming paradigms, has its own vocabulary. The table below defines a number of the words and phrases you may encounter when reading about or working with AOP. These definitions are not Spring-specific.

AOP Definitions

Concern	A particular issue, concept or area of interest for an application. Examples include transaction management, persistence, logging and security.
Crosscutting Concern	A concern in which the implementation cuts across many classes. These are often difficult to implement and maintain with OOP.
Aspect	The modularization of a crosscutting concern; implemented by gathering and isolating code.
Join Point	A point during the execution of a program or class. In Spring's AOP implementation, a join point is always a <i>method invocation</i> . Other examples include <i>accessing fields</i> , where read or write access occurs on an instance variable, and <i>exception handling</i> .
Advice	An action taken at a particular join point. Different types of advice in Spring include <i>around</i> , <i>before</i> , <i>throws</i> and <i>after returning</i> . Of these, <i>around</i> is the most powerful, as you get the opportunity to do something before and after a method is invoked. The previous <code>TraceInterceptor</code> used <i>around</i> advice by implementing the AOP Alliance's <code>MethodInterceptor</code> . You can use the other advice types by implementing the following Spring Interfaces: • MethodBeforeAdvice • ThrowsAdvice • AfterReturningAdvice
Pointcut	A set of join points specifying when an advice should fire. Pointcuts often use regular expressions or wildcard syntax.
Introduction	Adds fields or methods to an advised class. Spring allows you to introduce new interfaces to any advised object. For example, you could use an introduction to make any object implement an <code>IsModified</code> interface, to simplify caching.
Weaving	Assembles aspects to create an advised object. This can be done at compile time (this is how <code>AspectJ</code> does it) or at runtime. The <i>Weaving Strategies</i> section later in this chapter discusses in detail the different strategies for <i>weaving</i> (that is, implementing AOP).
Interceptor	An AOP implementation strategy, where a chain of interceptors may exist for a particular join point.
AOP Proxy	An object created by the AOP framework, including advice. In Spring, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
Target Object	An object containing the join point. In frameworks using interception, it's the object instance at the end of an interceptor chain. Also called an <i>advised</i> or <i>proxied</i> object.

The next section is on Pointcuts, which are the rules for applying advice. Because Spring's AOP is interceptor-based, I will often refer to advice as *interceptors*.

Pointcuts

Pointcuts are an important part of AOP. They give you the ability to specify when and where to invoke interceptors. In a sense, they're often similar to declarative validation, but rather than specifying a field to validate, you specify a method to inspect. In the table, pointcuts are defined as "a set of join points specifying when an advice (interceptor) should fire." Since Spring only supports *method invocation* join points, a pointcut is just a declaration of methods to which an interceptor should be applied.

The easiest way to define pointcuts in Spring AOP is using regular expressions in a context file. Below is an example that defines a pointcut for data manipulation operations:

```
<bean id="dataManipulationPointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*save.*</value>
      <value>.*remove.*</value>
    </list>
  </property>
</bean>
```

This pointcut says, "Intercept methods that start with *save* or *remove*."

Note: The [JdkRegexpMethodPointcut](#) class above requires J2SE 1.4, which has built-in regular expression support. You can also use the [Perl5RegexpMethodPointcut](#), which requires Jakarta ORO (included in MyUsers).

In most cases, you won't need to define individual pointcuts like the one listed above. Rather, Spring provides *advisor* classes that encapsulate interceptors and pointcuts in the same bean definition.

For regular expression pointcuts, you can use the [RegexpMethodPointcutAdvisor](#). Below is an example of a `RegularExpressionPointcutAdvice` that triggers a `NotificationInterceptor` when a user's information is saved:

```
<bean id="notificationAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref bean="notificationInterceptor"/>
  </property>
  <property name="pattern">
    <value>.*saveUser</value>
  </property>
</bean>
```

Currently, the `RegexMethodPointcutAdvisor` only supports Perl5 pointcuts, meaning that you must have *jakarta-oro.jar* in your classpath if you want to use it. A complete list of pointcuts and their useful advisors are available in the [org.springframework.aop.support](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.aop.support/) package.

Weaving Strategies

Weaving is the process of applying aspects to target objects. The list below shows the primary strategies for implementing AOP, ordered from simplest to most sophisticated.

Note: Much of the information in this section is based on information from [J2EE without EJB](#).

- JDK Dynamic Proxies
- Dynamic Byte-Code Generation
- Custom Class Loading
- Language Extensions

These strategies are implemented in various AOP Frameworks, all open-source.

Note: The best part about all of these frameworks is they have an interest in using the same standard in their APIs. To back this up, they created the [AOP Alliance](#) project, which specifies a number of interfaces for implementation. To see who's involved in this project, read its [member list](#).

The sections below describe each of these weaving strategies in detail.

JDK Dynamic Proxies

Dynamic Proxies is a feature built into J2SE 1.3+. It allows you to create an implementation of one or more interfaces on-the-fly. Dynamic proxies are built into the JDK, eliminating the risk of strange behavior in various environments. Their one limitation is they can't proxy classes, only interfaces. If you've designed your application correctly with interfaces, this shouldn't be an issue.

There is some reflection overhead when using dynamic proxies, but the performance impact is negligible in J2SE 1.4+ JVMs. Spring uses dynamic proxies by default when proxying against interfaces. The [dynaop](#) project uses this strategy when proxying interfaces.

For more information on Dynamic Proxies, see the [Java 2 SDK Documentation](#).

Dynamic Byte-Code Generation

Spring uses dynamic byte-code generation when proxying against classes. A popular tool for this is [CGLIB](#) (Code Generation Library). It intercepts methods by generating subclasses dynamically. These subclasses override parent methods and have hooks to invoke interceptor implementations. Hibernate uses CGLIB extensively, and it has been proven to be a very stable solution in J2EE environments.

One limitation is that dynamic subclasses cannot override and proxy `final` methods.

Custom Class Loading

Using a custom class loader allows you to advise all instances of created classes. This can be quite powerful, since it gives you the opportunity to change the behavior of the `new` operator. [JBoss AOP](#) and [AspectWerkz](#) use this approach, loading classes and weaving in their advices as defined in XML files.

The main danger in this approach is J2EE servers must typically control the class-loading hierarchy. What works on one server may not work in another.

Language Extensions

[AspectJ](#) is the pioneering framework of AOP in Java. Rather than using a simple strategy for weaving in aspects, it contains language extensions and its own compiler for using them.

While AspectJ is a very powerful and mature AOP implementation, its syntax is somewhat complex and not very intuitive. However, AOP itself is not very intuitive, so trying to implement with a new language can be difficult. Another limitation of this approach is the learning curve associated with a new language. However, if you want the full power of AOP, including field-level interception, AspectJ will likely become your new best friend. Integrating AspectJ with Spring is covered near the end of this chapter.

Spring takes a pragmatic "80/20 rule" approach to its AOP implementation. Rather than trying to satisfy *everyone's* needs, it solves the most common ones and leaves the rest to more specialized AOP frameworks.

Convenient Proxy Beans

As mentioned earlier, in order to apply advice to beans defined in a context file, they must be proxied. Spring contains a number of support classes (or Proxy Beans) to make proxying easier. The first is the [ProxyFactoryBean](#), which allows you to specify beans to be proxied and interceptors to apply. Below is an example of using this bean to create a proxy of a business object:

```
<bean id="businessObject"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="org.appfuse.service.BusinessObject"/>
  </property>
  <property name="interceptorNames">
    <list><ref bean="loggingInterceptor"/></list>
  </property>
</bean>
```

The example above uses an *inner-bean* in the "target" property's value. This is a convenient way to hide a business object behind a proxy, so it will always be advised when grabbed from the `ApplicationContext`.

The [TransactionProxyFactoryBean](#) is the most useful and most used Proxy Bean. It allows you to use AOP to define transactions declaratively on target objects. This useful feature was previously only available with EJB Container Managed Transactions (CMT). The usage of this bean is described in the *Practical AOP Examples* section.

AutoProxy Beans

The aforementioned Proxy Beans provide easy manipulation of single beans, but what if you want to proxy multiple beans, or all beans in a context? Spring provides two classes (in the [org.springframework.aop.framework.autoproxy](#) package) that simplify this procedure.

The first is the [BeanNameAutoProxyCreator](#), which allows you to specify a list of bean names as a property. This property supports literals (actual bean names) and wildcards like `*Manager`. You can set interceptors using the `"interceptorNames"` property.

```
<bean id="managerProxyCreator"
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames"><value>*Manager</value></property>
    <property name="interceptorNames">
      <list>
        <value>loggingInterceptor</value>
      </list>
    </property>
  </bean>
```

The second, more generic, multiple bean proxy creator is the [DefaultAdvisorAutoProxyCreator](#). To use this proxy bean, simply define it in your context file.

```
<bean id="autoProxyCreator"
      class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

Unlike the [BeanNameAutoProxyCreator](#), you cannot specify interceptors to apply. Rather, it will inspect any advisors you have in this file and figure if their pointcuts make them applicable to other beans. For more information on advisors, see Spring's [reference documentation](#).

Practical AOP Examples

This section contains several examples of using AOP to manage crosscutting concerns in your application. The concerns include transactions, caching and event notification.

Transactions

Specifying that operations should occur within a transaction can often result in a lot of duplication in your DAOs. Using transactions the traditional way requires a lot of calls to `tx.begin()` and `tx.commit()` in data manipulation methods. The good news is Spring and AOP can consolidate transaction concerns into a single location and configuration.

You might not have been aware, but you've been using AOP in the MyUsers application since the QuickStart Chapter. The ability to declaratively specify transaction attributes on the `userManager` bean is supported by Spring's AOP and the [TransactionProxyFactoryBean](#). The listing below shows a refactored version of the `userManager` bean – this time using a *transaction template* bean and an *inner-bean*.

```
<bean id="txProxyTemplate" lazy-init="true"
      class="org.springframework.transaction.interceptor.TransactionProxy
        FactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </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>
  <property name="preInterceptors">
    <list>
      <ref bean="cacheInterceptor"/>
    </list>
  </property>
</bean>

<bean id="userManager" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO"><ref bean="userDAO"/></property>
      <property name="validator">
        <ref bean="userValidator"/>
      </property>
    </bean>
  </property>
</bean>
```

```

        </bean>
    </property>
</bean>

```

Further details on declarative transactions and handling their exceptions and rollbacks will be covered in *Chapter 10*.

Caching in the Middle Tier

Another good example of a crosscutting concern is caching data. The main reason for adding a cache is to improve performance, especially if fetching data is a costly operation. Most of the frameworks discussed in the last chapter have their own caching solutions; however, caching is a practical crosscutting concern.

Since your motivation behind introducing caching is to improve performance, add a test to `UserManagerTest` to test the performance of `UserMangerImpl`. Put the code below in `test/org/appfuse/service/UserManagerTest.java`.

Note: The `StopWatch` class in this test is a utility class for timing tasks like this.

```

public void testGetUserPerformance() {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");

    user = mgr.saveUser(user);

    String name = "getUser";
    StopWatch sw = new StopWatch(name);
    sw.start(name);
    log.debug("Begin timing of method '" + name + "'");

    for (int i=0; i < 200; i++) {
        mgr.getUser(user.getId().toString());
    }

    sw.stop();
    log.info(sw.shortSummary());
    log.debug("End timing of method '" + name + "'");
}

```

Running `ant test -Dtestcase=UserManagerTest` (with no interceptors configured in `applicationContext.xml`) should yield results similar to Figure 9.2. In this initial test, fetching the same user took about 10 seconds.

Warning! If you use `-Dtestcase=UserManager` (without the "Test" suffix), it will run the mock tests we created in the last chapter. Since these tests isolate the `userManager` from Spring, they won't demonstrate applying interceptors to be defined in the `applicationContext.xml` file.

```

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

compile:

test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) ! Expected exception:
Object of class [org.appfuse.model.User] with identifier [36]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) ! Begin timing of
method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) ! Stopwatch 'getUser':
running time <seconds> = 10.125

[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) ! End timing of method
'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 11.719 sec

BUILD SUCCESSFUL
Total time: 13 seconds
$

```

Figure 9.2: Results of the `ant test -Dtestcase=UserManagerTest` test

In order to improve performance, add a simple cache to the `userManagerImpl`. The basic requirements for this cache are as follows:

1. Whenever you fetch objects from the database, put them into the cache (which is a simple `HashMap` in this example).
2. When the `save()` or `delete()` methods are called, remove objects from the cache.

In the `MyUsers` application, you can implement the basic (non-AOP) solution by adding a `BaseManager` (in the `org.appfuse.service.impl` package) from which all `ManagerImpls` extend:

```

package org.appfuse.service.impl;

// use your IDE to organize imports

public class BaseManager {
    // Adding this log variable will allow children to re-use it
    protected final Log log = LogFactory.getLog(getClass());
    protected Map cache;

```

```

protected void putIntoCache(String key, Object value) {
    if (cache == null) {
        cache = new HashMap();
    }
    cache.put(key, value);
}

protected void removeFromCache(String key) {
    if (cache != null) {
        cache.remove(key);
    }
}
}

```

To modify the `UserManagerImpl` to use this cache, **make it extend `BaseManager`**, then add calls to the cache in each method. Below is a simple version of `UserManagerImpl` before adding these calls:

```

public User getUser(String userId) {
    return dao.getUser(Long.valueOf(userId));
}

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

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

```

After adding these methods, they become much more verbose:

```

public User getUser(String userId) {
    // check cache for user
    User user = (User) cache.get(userId);
    if (user == null) {
        // user not in cache, fetch from database
        user = dao.getUser(Long.valueOf(userId));
        super.putIntoCache(userId, user);
    }
    return user;
}

public User saveUser(User user) {
    dao.saveUser(user);
    // update cache with saved user
    super.putIntoCache(String.valueOf(user.getId()), user);
}

```

```

        return user;
    }

    public void removeUser(String userId) {
        dao.removeUser(Long.valueOf(userId));
        // remove user from cache
        super.removeFromCache(userId);
    }

```

Running `ant test -Dtestcase=UserManagerTest` increases the performance. In Figure 9.3, the running time is 9 seconds (1 second faster).

```

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

compile:

test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) ! Expected exception
n: Object of class [org.appfuse.model.User] with identifier [48]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) ! Begin timing of
method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) ! Stopwatch 'getUs
er': running time (seconds) = 8.859
[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) ! End timing of m
ethod 'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 10.546 sec

BUILD SUCCESSFUL
Total time: 13 seconds
$_

```

Figure 9.3: Results of the `ant test -Dtestcase=UserManagerTest` test

While adding these calls to add/remove from the cache is easy with a simple application like MyUsers, it will become increasingly difficult with a larger application. You will have to remember to add these methods to each Manager. Since caching is not a core concern of the application, you shouldn't really be concerned with it.

By using AOP, you can remove these method calls and *intercept* the appropriate methods to use the cache. Furthermore, by abstracting the caching away from your code, it allows you to concentrate on the business logic and doing the project.

To add a caching interceptor, perform the following steps:

1. Create a `CacheInterceptor` class in the `org.appfuse.aop` package (you must create this). Populate it with the following code:

```
package org.appfuse.aop;

// use your IDE to organize imports

public class CacheInterceptor implements MethodInterceptor {
    private final Log log = LogFactory.getLog(CacheInterceptor.class);

    public Object invoke(MethodInvocation invocation) throws Throwable {
        String name = invocation.getMethod().getName();
        Object returnValue;

        // check cache before executing method
        if (name.indexOf("get") > -1 && !name.endsWith("s")) {
            String id = (String) invocation.getArguments()[0];
            returnValue = cache.get(id);
            if (returnValue == null) {
                // user not in cache, proceed
                returnValue = invocation.proceed();
                putIntoCache(id, returnValue);
                return returnValue;
            }
            else {
                //log.debug("retrieved object id '" + id + "' from cache");
            }
        }
        else {
            returnValue = invocation.proceed();

            // update cache after executing method
            if (name.indexOf("save") > -1) {
                Method getId = returnValue.getClass().getMethod("getId",
new Class[]{});
                Long id = (Long) getId.invoke(returnValue, new Object[]{});
                putIntoCache(String.valueOf(id), returnValue);
            }
            else if (name.indexOf("remove") > -1) {
                String id = (String) invocation.getArguments()[0];
                removeFromCache(String.valueOf(id));
            }
        }
        return returnValue;
    }

    protected Map cache;

    protected void putIntoCache(String key, Object value) {
        if (cache == null) {
```

```
        cache = new HashMap();
    }
    cache.put(key, value);
}

protected void removeFromCache(String key) {
    if (cache != null) {
        cache.remove(key);
    }
}
}
```

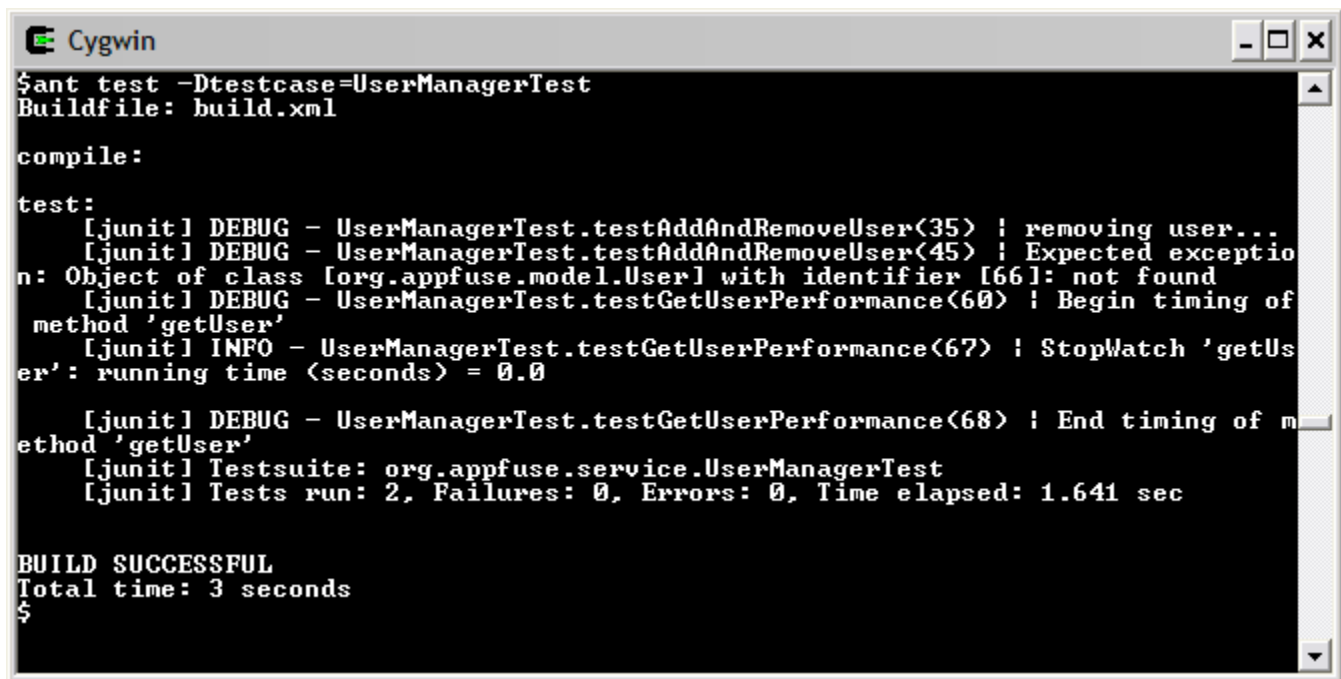
2. Add a "cacheInterceptor" bean to *applicationContext.xml* (in *web/WEB-INF*).

```
<bean id="cacheInterceptor"
      class="org.appfuse.aop.CacheInterceptor"/>
```

3. Replace the "loggingInterceptor" with the "cacheInterceptor" on the userManager bean.

```
<property name="preInterceptors">
    <list>
        <ref bean="cacheInterceptor"/>
    </list>
</property>
```

Running `ant test -Dtestcase=UserManagerTest` results in a dramatic performance increase.



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

compile:

test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) : removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) : Expected exception
n: Object of class [org.appfuse.model.User] with identifier [66]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) : Begin timing of
method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) : Stopwatch 'getUs
er': running time <seconds> = 0.0
[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) : End timing of m
ethod 'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 1.641 sec

BUILD SUCCESSFUL
Total time: 3 seconds
$
```

Figure 9.4: Results of the `ant test -Dtestcase=UserManagerTest` test

Using a *caching aspect*, you've reduced the performance to **0 seconds**! The reason for the dramatic increase in performance is the `CacheInterceptor` returns the data before almost all of the method invocations on `UserManagerImpl`.

The caching example provided here is very simplistic. For a more robust caching implementation, refer to Pieter Coucke's [Spring AOP Cache](#) or [using EHCACHE with Spring](#).

Event Notification

If you're developing a web application that's open to the public, you might want to be aware when new users sign up. This can be particularly useful if you want to examine the user's information before authorizing an account. In the following example, you'll create a `NotificationInterceptor` and configure it to send e-mail when a new user registers.

1. Modify the `UserManagerTest` to use `Dumbster` to ensure that a message is sent when a new user registers. Only relevant sections are included below. The new code in this class is underlined. After adding this, run `ant test -Dtestcase=UserManagerTest` to ensure it fails.

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

    // Modify the mailSender bean to use Dumbster's ports
    JavaMailSenderImpl mailSender =
    (JavaMailSenderImpl) ctx.getBean("mailSender");
    mailSender.setPort(2525);
}

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

    // setup a simple mail server using Dumbster
    SimpleSmtpServer server = SimpleSmtpServer.start(2525);

    user = mgr.saveUser(user);

    server.stop();
    assertEquals(1, server.getReceievedEmailSize()); // spelling is
correct
    SmtplibMessage sentMessage =
    (SmtplibMessage) server.getReceivedEmail().next();
    assertTrue(sentMessage.getBody().indexOf("Easter Bunny") != -1);
    log.debug(sentMessage);

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

2. Create a `NotificationInterceptor` class in `src/org/appfuse/aop`. Populate it with the code listed below:

```
package org.appfuse.aop;

// organize imports using your IDE

public class NotificationInterceptor implements MethodInterceptor {
    private final Log log =
        LoggerFactory.getLog(NotificationInterceptor.class);
    private MailSender mailSender;
    private SimpleMailMessage message;

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

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

    public Object invoke(MethodInvocation invocation) throws Throwable {

        User user = (User) invocation.getArguments()[0];
        if (user.getId() == null) {
            if (log.isDebugEnabled()) {
                log.debug("detected new user...");
            }
            Object returnValue = invocation.proceed();
            StringBuffer sb = new StringBuffer(100);
            sb.append("A new account has been created for " + user.getFullName());
            sb.append(".\n\nView this users information at:\n\t ");
            sb.append("http://localhost:8080/myusers/editUser.html?");
            sb.append("id=" + user.getId());
            message.setText(sb.toString());

            mailSender.send(message);
        }

        return user;
    }
}
```

3. Configure this interceptor and its dependent beans (MailSender and SimpleMailMessage) in *web/WEB-INF/applicationContext.xml*.

Warning! Be sure to change the e-mail address in the "to" property of the "accountMessage" bean below. If you don't, the test will fail.

```
<bean id="notificationInterceptor"
      class="org.appfuse.aop.NotificationInterceptor">
  <property name="mailSender"><ref bean="mailSender"/></property>
  <property name="message"><ref bean="accountMessage"/></property>
</bean>

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

<bean id="accountMessage" singleton="false"
      class="org.springframework.mail.SimpleMailMessage">
  <property name="to">
    <value><![CDATA[MyUser Admin <youremailhere>]]></value>
  </property>
  <property name="from">
    <value><![CDATA[MyUsers <mattr@sourcebeat.com>]]></value>
  </property>
  <property name="subject">
    <value>MyUsers Account Information</value>
  </property>
</bean>
```

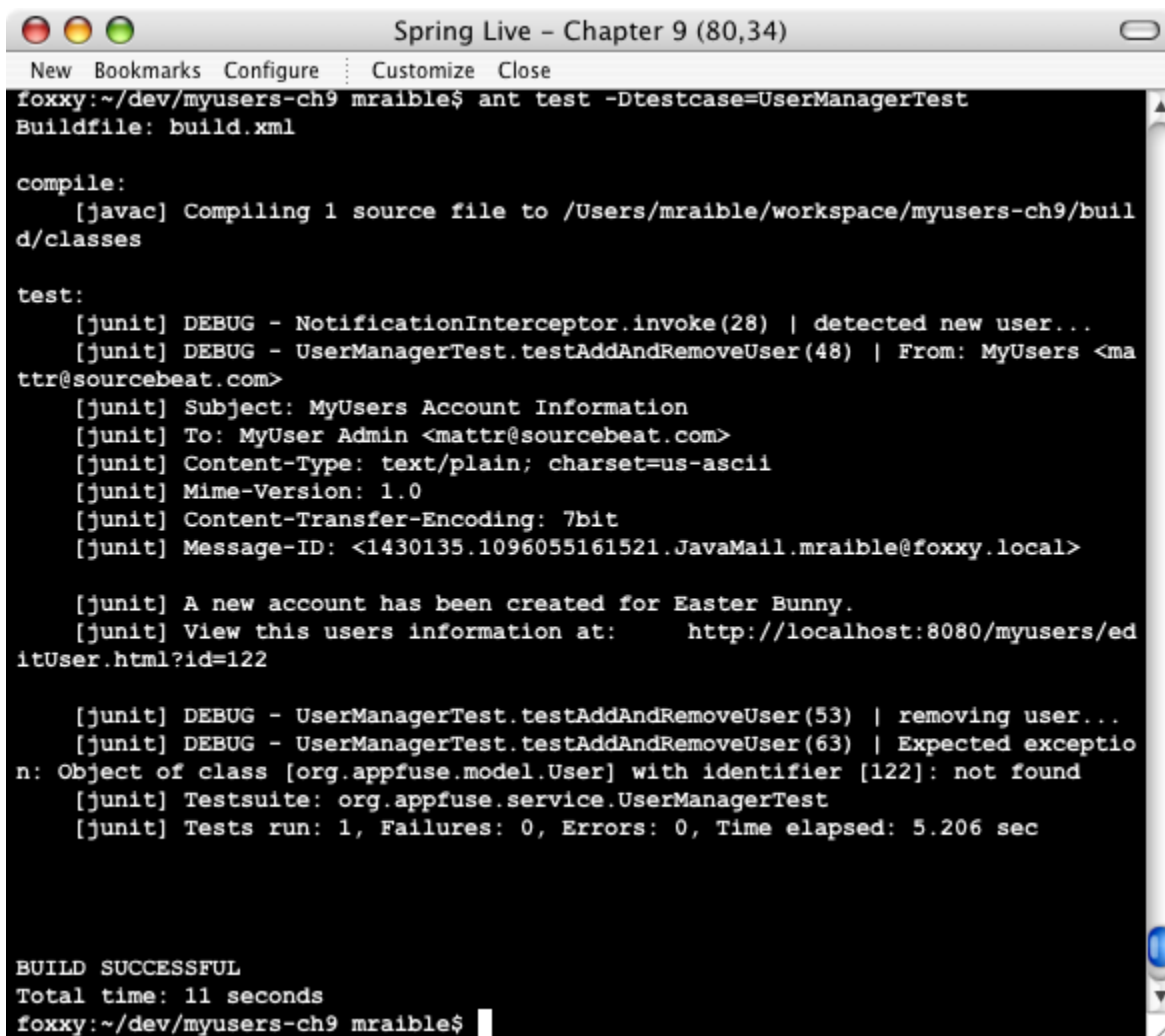
4. Configure this interceptor in an *advisor* to be fired using the *RegexMethodPointcutAdvisor* class.

```
<bean id="notificationAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref bean="notificationInterceptor"/>
  </property>
  <property name="pattern">
    <value>.*saveUser</value>
  </property>
</bean>
```

5. Add the "notificationAdvisor" bean to the list of "preInterceptors" on the userManager bean. I recommend commenting out the other examples.

```
<property name="preInterceptors">
  <list>
    <!--ref bean="loggingInterceptor"/-->
    <!--ref bean="cacheInterceptor"/-->
    <ref bean="notificationAdvisor"/>
  </list>
</property>
```

6. Save all your files and run `ant test -Dtestcase=UserManagerTest`. Your console output should be similar to Figure 9.5.



```
Spring Live - Chapter 9 (80,34)
New Bookmarks Configure Customize Close
fooxy:~/dev/myusers-ch9 mraible$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:
[javac] Compiling 1 source file to /Users/mraible/workspace/myusers-ch9/build/classes

test:
[junit] DEBUG - NotificationInterceptor.invoke(28) | detected new user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(48) | From: MyUsers <matr@sourcebeat.com>
[junit] Subject: MyUsers Account Information
[junit] To: MyUser Admin <matr@sourcebeat.com>
[junit] Content-Type: text/plain; charset=us-ascii
[junit] Mime-Version: 1.0
[junit] Content-Transfer-Encoding: 7bit
[junit] Message-ID: <1430135.1096055161521.JavaMail.mraible@fooxy.local>

[junit] A new account has been created for Easter Bunny.
[junit] View this users information at: http://localhost:8080/myusers/editUser.html?id=122

[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(53) | removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(63) | Expected exception: Object of class [org.appfuse.model.User] with identifier [122]: not found
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.206 sec

BUILD SUCCESSFUL
Total time: 11 seconds
fooxy:~/dev/myusers-ch9 mraible$
```

Figure 9.5: Results of the `ant test -Dtestcase=UserManagerTest` test

Using a setup like this, you could easily add notifications based on values or conditions.

These examples are a good introduction to AOP and how to use it in your application. Some other examples of where you might use AOP in your applications include:

- Resource pooling
- Thread pooling
- XSLT Stylesheet caching
- Authentication and authorization

Examples and code for implementing these are available in *AspectJ in Action*.

AOP is very powerful and new users should be careful when introducing it into their codebase. Two good ways to get into AOP and learn more about it are listed below:

1. Via recipe: Copy and paste code from books or articles.
2. Non-production: The nice thing about AOP is you don't have to use all its features at once. It has a nice incremental curve. You can start using it in development (for example, the Logging/Tracing example), then you can move into having items that check on policy or help with testing. Once you get comfortable, you can start using it in production.

While AOP might not be for everyone, it has a place for creating modular, maintainable Java applications.

Summary

AOP has been around for several years, but only recently has it gained so much popularity in the Java arena. This is likely due to the many open-source frameworks for implementing AOP. At the time of this writing, these include [AspectJ](#), [AspectWerkz](#), [dynaop](#), [JAC](#), [JBoss AOP](#) and Spring AOP.

Spring provides a simple-to-use API for AOP, and it integrates well with other AOP frameworks. As of Spring version 1.1, it provides [powerful integration with AspectJ](#). It also [integrates nicely with AspectWerkz](#). In both of these integrations, Spring's IoC container configures and manages the lifecycle of the created aspects.

In this chapter, you learned about the different concepts in AOP, from *aspects* to *pointcuts* to *weaving*. After defining the verbiage in AOP, the different implementation strategies were explored. JDK Dynamic Proxies and byte-code manipulation are the most popular weaving strategies used in the aforementioned AOP frameworks. AspectJ is the most mature and sophisticated AOP implementation, providing its own language and compiler for integrating aspects and classes. Finally, you learned how to write aspects and advisors in the MyUsers application to implement logging, caching, transactions and event notification.

So what's the future of Spring's AOP framework? According to its [roadmap](#), many of its goals have been met in the 1.1 release, with improved performance and AspectJ integration. [Spring AOP Cache](#) and the [Acegi Security System for Spring](#) are good examples of generic implementations for caching and security. This pool of resources will grow extensively as more users become familiar with AOP and start using it in their applications.