Jakarta-Struts LIVE

Rick Hightower

SourceBeat

Jakarta Struts Live

by Richard Hightower

Published by SourceBeat, LLC, Highlands Ranch, Colorado.

Managing Editor:	James Goodwill
Technical Editor:	Kelly Clauson
Copy Editor:	Brent Barkley
Layout Designer:	Amy Kesic
Cover Designer:	Max Hays

ISBN: 0974884308

Table of Contents

Dedication

To my wife: Kiley.

Acknowledgments

Firstly, thanks to James Goodwill for inviting me to the SourceBeat party.

Next and very importantly, I'd like to thank the Jakarta Struts team: Ted Husted, Craig R. McClanahan, and all the Struts committers (and contributors) for the long hours and dedication they spent bringing the J2EE blueprint vision for web application to fruition. Struts pushed the limits of the J2EE web application development stack, and influenced JSTL, and JavaServerFaces as well as many competing frameworks.

I'd like to thank Andrew Barton, and Brent Buford the co-founders of eBlox for hiring me as director of development, and letting me work with their talented developers. We were early adopters of Struts at eBlox. I'd like to thank my fellow eBlox developers: Nick Liescki, Paul Visan, John Thomas, Andrew Barton, Ron Tapia, Warner Onstine, Doug Albright, Brett, Ali Khan, Bill Shampton, and many more. It was through hours of pair-programming with them that we sharpened our Struts and web development skills. I am a better developer for the experience. Most of all from the eBlox crew, I'd like to thank Erik Hatcher, who quickly became the onsite Struts and Ant expert and has always provided me with a lot of feedback on my projects, long after we both left eBlox.

I'd like to thank Kimberly Morrello who hired me as CTO of her training and consulting company where I led the authoring of their Struts course and went around the U.S. consulting companies with their first Struts and J2EE applications. It was a great experience. I learned so much by meeting so many people, and I learned a lot of great Struts tricks along the way. From this group, special thanks to Andy Larue and Paul Hixson, who created a wonderful architecture guide to using Struts.

I would like to thank everyone at my company, ArcMind, INC., for their support while I was writing this book. Special thanks to Sydney Bailey who helps me manage my hectic schedule so that I can respond to our clients in a timely manner and focus on things like writing a book or developing a web application.

Special thanks to Matt Raible, the author of AppFuse and a regular in the Struts community, for reviewing several chapters and providing valuable input. I'd also like to thank Matt Raible for creating AppFuse, which I am using as a base for my current project and plan on using for my next project too (see the chapter on AppFuse for more detail).

Thanks to the technical editor, Kelly Clauson. I've never worked with a more thorough technical editor. Kelly has an eye for detail that has really improved the step-by-step instructions in the book.

Finally, I'd like to thank my wife. I could not run the technical aspects of ArcMind or write books without her support. Also, I'd like to thank Starbucks for creating a wonderful array of caffeinated beverages that keep me working till the wee hours of the morning.

About the Author

Rick Hightower is the co-founder, CTO and chief mentor at ArcMind, INC, in Tucson, AZ. ArcMind is a training and consulting company focusing on Struts and J2EE. With 14 years of experience, Rick leads the technical consulting and training group. Previously, Rick worked as the director of architecture at a J2EE container company, director of development at a web development company, and CTO of a global training and consulting company. Rick is also available to consult and mentor teams on Struts projects (http://www.arc-mind.com).

Rick is the co-author of the best-selling: *Java Tools for Extreme Programming.* Rick has also contributed to several other books on subjects ranging from CORBA to Python. Rick has written articles and papers for Java Developers Journal and IBM developerWorks on subjects ranging form EJB CMP/CMR to Struts Tiles.

Rick has spoken at TheServerSide Symposium, JavaOne, JDJEdge, No Fluff Just Stuff, XPUniverse and more on subjects ranging from *Extending Struts* to *J2EE development with XDoclet*.

Preface

Wow. It is a great honor to work on this book. I dig the concept of SourceBeat. I've worked on a lot of books in the past, and they all have one thing in common: they are out of date as soon as they are written and it takes a year or more before they are updated. Bringing books online in this format is long overdue. Book writing, meet Internet time!

Here is a quote from James Goodwill that sums up my feelings on this subject:

> "Well, I have to tell you that this is one of the more exciting books that I have written. Over my writing career I have written books describing several different open-source topics and while I really enjoyed writing about the topics themselves, I did not enjoy working on a title for 4 to 6 months, sending it to the publisher, waiting for them to bind and print the text, and then see it hit the bookshelves 3 months later completely out of date. I have had some books hit the bookshelves and, because of the lengthy printing process, the project changed and my examples were completely wrong. This progression was frustrating both to me and my readers.
>
> "As I am sure you are aware (because you did purchase this title) this text is being published using a completely different publishing model—it is an annual subscription. This gives me the ability to write a book on an open source topic and continually update that book based upon changes in the targeted project. I even have the ability to add additional content, when I see a new way of doing things or just have a really cool idea that I would like to add. To me this is going to be a lot of fun. My readers, you included, will have timely material describing an open-source topic and I will, for once in my writing career, feel good about the book that I am writing."

This book covers what you need to know about Struts and related technologies like the Validator framework , JavaServerFaces, JSTL, Tiles and more. Rather then regurgitating the online reference materials, this book endeavors to provide guides to using the technologies and techniques to create web applications with Struts. The guides include background information (theory), step-by-step instructions, and best practices.

My goals for this book are simple: be the best, most exhaustive, comprehensive guide to developing web applications with Struts. I need you to help me with this goal. If you know of best practices, topics, common fixes, related technologies, etc. that you think should be in this book, let me know. If I put it in the book, I will credit you as the person who suggested the new topic. Remember that this book is an online book that will be updated and added-to continually. Like software, books get better with refactoring! You can contact me at rick@sourcebeat.com.

Thanks,

Richard M. Hightower Sr.

Introduction

The book is a step-by-step guide to using Struts with best practices and enough theory to get you using and understanding the technology on your web applications. The focus of the book is to provide examples and step-by-step guides, not a rehash of the online documents.

Chapter 1: Struts Tutorial Step-by-step covers getting started with Struts—just the facts to getting started with Struts ASAP. In one chapter you will learn about the struts configuration, writing Actions, working with Struts Custom tags, setting up datasource, handling exceptions, and displaying objects in a JSP in enough detail for you to get started with each. In this chapter you will perform the following steps: download Struts, setup a J2EE web application project that uses Struts, write your first Action, write your first "forward", configure the Action and forward in the Struts configuration file, run and Test your first Struts application and more. This chapter also covers an important topic for newbies and Struts veterans alike, namely, debugging struts-config.xml, and logging support for debugging and understanding your Struts web application at runtime.

Chapter 2: Testing Struts covers test-driven-development with Struts. This chapter lays the groundwork with JUnit then covers StrutsTestCase and jWebUnit. You will use JUnit to test your model code. You will use StrutsTestCase to test your controller Actions. And, you will use jWebUnit to test your view and presentation logic. This chapter lays the groundwork for TDD. Chapter 9 picks up the torch on TDD.

Chapter 3: Working with ActionForms and DynaActionForms is divided into two sections. The first section covers mostly theory and concepts behind ActionForms. The second part covers common tasks that you will need to do with ActionForms like: ActionForms that implement a master detail, ActionForms with nested JavaBean properties, ActionForms with nested indexed JavaBean properties, ActionForms with mapped backed properties, loading form data in an ActionForm to display, working with wizards-style ActionForms, configuring DynaActionForms and more. All of the concepts introduced are backed up with examples, and step-by-step guides.

Chapter 4: Working with the Validator Framework covers understanding how the Validator Framework integrates with Struts and using the Validator Framework with static ActionForms and with DynaActionForms. The chapter includes step-by-step guides to working with common validation rules. Then it continues by demonstrating building and using your own configurable validation rules. There are some special considerations when using the Validator framework with Wizard; thus this chapter covers in detail working with wizards and Validator framework by employing the page attribute. It also covers the common technique of using the Validator Framework and your own custom validation at the same time. In addition the chapter covers employing JavaScript validation on the client side.

Chapter 5: Working with Actions covers all of the standard actions and the helper methods of the action class as well as advanced action mapping configurations.

Chapter 6: MVC for Smarties covers background and theory of MVC as it applies to Struts. Java promotes object-oriented programming, but does not enforce it, i.e., you can create non OO code in Java. Similarly, Struts promotes MVC / Model 2 architecture, but does not enforce it. To get the best out of Struts, you need to know MVC / Model 2. This chapter covers the basic rules and guidelines of MVC so you can get the most out of Struts.

Chapter 7: Working with Struts tags covers JSP custom tag fundamentals and using the standard Struts tags (html, logic and bean). This chapter is not a rehash of the documents. There are a lot of examples.

Chapter 8: JSTL and Struts EL covers combining JSP Standard Tag Library (JSTL) and Struts. Struts pushed JSP custom tags to the limit. JSTL extends many of the concepts found in Struts Logic Tags. In many ways, JSTL is the natural successor to the Struts tag libraries. You may wonder why we mention JSTL in a book on Struts. Simply, JSTL tags take the place of many Struts tags, and are easier to use. It is the recommendation of the Struts Development team that you should use JSTL tags in place of Struts tags when there is an overlap. In addition, many of the Struts standard tags have been updated to use JSTL EL. Thus, this chapter covers Struts-EL as well.

Chapter 9: Ignite your Struts web application development with AppFuse covers developing a real world web application. AppFuse is a starter web application that has a starter project that includes support for Object/Relation mapping, J2EE security, database testing, enforcing MVC with best practices, and common design patterns. It includes support for Hibernate, Canoo, DBUnit, and much more.

Chapter 10: Extending Struts covers writing plug-ins, custom config objects, and custom request dispatchers to extend the Struts framework. (More to come.)

Chapter 11: Using Tiles covers using Tiles to create reusable pages and visual components. This chapter covers building Web applications by assembling reusable tiles. You can use tiles as templates or as visual components. If you are using Struts but not Tiles, then you are not fully benefiting from Struts and likely repeat yourself unnecessarily. The Tiles framework makes creating reusable site layouts and visual components feasible. This chapter covers the following topics: the Tiles framework and architecture, building and using a tile layout as a site template, using tile definitions using XML config file and/or JSP, moving objects in and out of tile scope, working with attributes lists, working with nested tiles, building and using tile layouts as small visual components, extending definitions for maximum JSP reuse, creating a tile controller, and using a tile as an ActionForward.

Chapter 12: Writing Custom Tags covers writing custom tags that work with Struts (more to come).

Chapter 13: Working with Struts I18n support covers I18n (more to come).

Chapter 14: Integrating JSF and Struts covers using StrutsFaces, the Struts JSF support. By using Struts, Tiles, and JavaServer Faces (JSF) together, developers can ensure a robust, well-presented Web application that is easy to manage and reuse. The Struts framework is the current de facto web application development framework for Java. The Tiles framework, which ships as part of Struts, is the de facto document centric, templating-component framework that ships with Struts. Struts and JSF do overlap. However, Struts provides complementary features above and beyond the JSF base. Struts works with JSF via its Struts-Faces integration extensions. Using these extensions allows companies to keep the investment they have in Struts and still migrate to the new JSF programming model. This chapter covers the following: introduction to JavaServer Faces (JSF) and benefits; architecture of JSF; combing Struts with JSF; Configurations and Plugins; Architecture of JSF integration; and a step by step guide to getting all the pieces to work together.

Chapter 15: Integrating Portlets and Struts covers using Struts with Portlets (more to come).

Chapter 16: Using Velocity Struts Tools covers using Velocity for implementing the view (more to come).

Chapter 17: Integrating the Killer open source stack: Spring, Struts and Hibernate covers integrating Spring, Struts and Hibernate. (more to come).

Chapter 1

Struts Quick Start Tutorial

Getting started with Struts, skip the fluff, just the facts!

This chapter is a tutorial that covers getting started with Struts—just the basics, nothing more, nothing less. This tutorial assumes knowledge of Java, JDBC, Servlets, J2EE (with regards to Web applications) and JSP. Although you can follow along if you are not an expert in all of the above, some knowledge of each is assumed.

Instead of breaking the chapters into a myriad of single topics, which are in depth ad nauseum, we treat Struts in a holistic manner, minus the beads and crystals. The focus of this chapter is to skim briefly over many topics to give you a feel for the full breadth of Struts. Later chapters will delve deeply into many topics in detail.

In this chapter, you will cover:

- The struts configuration
- Writing Actions
- Working with Struts Custom tags
- Setting up datasource
- Handling exceptions
- Displaying objects in a JSP

In this chapter you will perform the following steps:

1. Download Struts
2. Setup a J2EE web application project that uses Struts
3. Write your first Action
4. Write your first “forward”
5. Configure the Action and forward in the Struts configuration file
6. Run and Test your first Struts application.
7. Debugging Struts-Config.xml with the *Struts Console*
8. Add Logging support with Log4J and commons logging.
9. Write your first ActionForm
10. Write your first input view (JSP page)
11. Update the Action to handle the form, and cancel button
12. Setup the database pooling with Struts
13. Declaratively Handle Exception in the Struts Config file
14. Display an Object with Struts Custom Tags

Download Struts

The first step in getting started with Struts is to download the Struts framework. The Struts home page is located at http://jakarta.apache.org/struts/index.html. You can find online documentation at the Struts home page. However, to download Struts you need to go to the Jakarta Download page at http://jakarta.apache.org/site/binindex.cgi. Since all of the Jakarta download links are on the same page, search for “Struts” on this page. Look for the link that looks like this:

Struts KEYS

- 1.1 zip PGP MD5
- 1.1 tar.gz PGP MD5

Download either compressed file.

One of the best forms of documentation on Struts is the source. Download the source from http://jakarta.apache.org/site/sourceindex.cgi. Once you have both the source and binaries downloaded, extract them. (WinZip works well for Windows users.) This tutorial will assume that you have extracted the files to c:\tools\ jakarta-struts-1.1-src and c:\tools\ jakarta-struts-1.1. If you are using another drive, directory, or *n?x (UNIX or Linux), adjust accordingly.

Set up a J2EE Web Application Project That Uses Struts

Struts ships with a started web application archive file (WAR file) called *struts-blank.war*. The struts-blank.war file has all of the configuration files, tag library descriptor files (tld files) and JAR files that you need to start using Struts. The struts-blank.war file includes support for Tiles and the Validator framework. You can find the struts-blank.war file under C:\tools\jakarta-struts-1.1\webapps.

1. A war file is the same format as a ZIP file. Extract the struts-blank.war file to a directory called c:\strutsTutorial (adjust if *n?x). When you are done, you should have a directory structure as follows:

```
C:.
|---META-INF
|---pages
|---WEB-INF
  |---classes
  | |--resources
  |---lib
  |---src
    |---java
       |---resources
```

   The blank war file ships with an Ant build script under WEB-INF/src. The structure of the extracted directory mimics the structure of a deployed web application.

2. In order for the *build.xml* file to work, you need to modify it to point to the jar file that contains the Servlet API and the jar file that points to the JDBC API from your application server.

   For example, if you had Tomcat 5 installed under c:\tomcat5, then you would need to modify the *servlet.jar* property as follows:

```
<property name="servlet.jar"
          value="C:/tomcat5/common/lib/servlet-api.jar"/>
```

Tip: Tomcat is a Servlet container that supports JSP. If you are new to web development in Java, there are several very good books on Java web development. You will need to know about JSP, Servlets and web applications to get the most out of this chapter and this book. If you are new to Java web development, try this tutorial: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/WebApp.html#wp76431.

3. If you are using an IDE (Eclipse, NetBeans, JBuilder, WSAD, etc.), set up a new IDE project pointing to C:\strutsTutorial\WEB-INF\src\java as the source directory, add your application server's servlet.jar file (servlet-api.jar for tomcat) and all the jar files from C:\strutsTutorial\WEB-INF\lib.

Write Your First Action

Actions respond to requests. When you write an Action, you subclass `org.apache.struts.action.Action` and override the execute method.

The execute method returns an ActionForward. You can think of an ActionForward as an output view.

The execute method takes four arguments: an ActionMapping, ActionForm, HttpServletRequest and an HttpServletResponse (respectively).

The ActionMapping is the object manifestation of the XML element used to configure an Action in the Struts configuration file. The ActionMapping contains a group of ActionForwards associated with the current action.

For now, ignore the ActionForm; we will cover it later. (It is assumed that you are familiar with HttpServletRequest and HttpServletResponse already.)

Go to strutsTutorial\WEB-INF\src\java and add the package directory strutsTutorial. In the strutsTutorial directory, add the class UserRegistration as follows:

```
package strutsTutorial;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class UserRegistrationAction extends Action {

public ActionForward execute(
ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response)
throws Exception {

return mapping.findForward("success");
}

}
```

Notice that this Action forwards to the output view called `success`. That output view, ActionForward, will be a plain old JSP. Let's add that JSP.

Write Your First "Forward"

Your first forward will be a JSP page that notifies the user that their registration was successful. Add a JSP page to c:\strutsTutorial called regSuccess.jsp with the following content:

```
<html>
<head>
<title>
User Registration Was Successful!
</title>
</head>

<body>
<h1>User Registration Was Successful!</h1>
</body>
</html>
```

The forward is the output view. The Action will forward to this JSP by looking up a forward called `success`. Thus, we need to associate this output view JSP to a forward called success.

Configure the Action and Forward in the Struts Configuration File

Now that we have written our first Action and our first Forward, we need to wire them together. To wire them together we need to modify the Struts configuration file. The Struts configuration file location is specified by the config init parameter for the Struts ActionServlet located in the web.xml file. This was done for us already by the authors of the blank.war starter application. Here is what that entry looks like:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
  org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  ...
  <load-on-startup>2</load-on-startup>
</servlet>
```

Thus, you can see that the blank.war's web.xml uses WEB-INF/struts-config.xml file as the Struts configuration file.

Follow these steps to add the success forward:

1. Open the c:\strutsTutorial\WEB-INF\struts-config.xml file.
2. Look for an element called action-mappings.
3. Add an action element under action-mappings as shown below.

```
<action path="/userRegistration"
       type="strutsTutorial.UserRegistrationAction">

   <forward name="success" path="/regSuccess.jsp"/>
</action>
```

The above associates the incoming path /userRegistration with the Action handler you wrote earlier, `strutsTutorial.UserRegistrationAction`.

Whenever this web application gets a request with /userRegistration.do, the execute method of the strutsTutorial.UserRegistrationAction class will be invoked. The web.xml file maps request that end in *.do to the Struts ActionServlet. Since the web.xml file was provided for us, you will not need to edit it. Here is the mapping for in the web.xml file for reference:

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

The Struts ActionServlet will invoke our action based on the path attribute of the above action mapping. The ActionServlet will handle all requests that end in *.do. You may recall that our Action looks up and returns a forward called success. The forward element maps the success forward to the regSuccess.jsp file that you just created in the last section.

The ActionMapping that gets passed to the execute method of the Action handler is the object representation of the action mapping you just added in the Struts config file.

Run and Test Your First Struts Application

The blank.war file ships with a started Ant script that will build and deploy the web application.

If you are new to Ant, then today is a good day to get up to speed with it. Ant is a build system from Jakarta. Struts uses a lot of Jakarta projects. Most Jakarta projects use Ant. Ant is also a Jakarta project. (Technically, it used to be a Jakarta project, and it was promoted to a top level project at the 1.5 release.) You can learn more about Ant and read documentation at http://ant.apache.org/. You can download Ant at http://www.apache.org/dist/ant/. Also, you can find an install guide for Ant at http://ant.apache.org/manual/installlist.html.

Technically, you do not have to use Ant to continue on with this tutorial, but it will make things easier for you. It's up to you. If you are not using Ant, now is a good time to start. Read http://ant.apache.org/manual/usinglist.html to start using Ant after you install it.

If you are using the Ant build.xml file that ships with the blank.war (look under WEB-INF/src), you will need to add the ${servlet.jar} file to the compile.classpath as follows:

```
<path id="compile.classpath">
    <pathelement path ="lib/commons-beanutils.jar"/>
    <pathelement path ="lib/commons-digester.jar"/>
    <pathelement path ="lib/struts.jar"/>
    <pathelement path ="classes"/>
    <pathelement path ="${classpath}"/>
    <pathelement path ="${servlet.jar}"/>
</path>
```

Notice the addition of the <pathelement path ="${servlet.jar}"/>. The compile classpath is used by the compile target.

After you add the pathelement, change the project.distname property to strutsTutorial as follows:

```
<property name="project.distname" value="strutsTutorial"/>
```

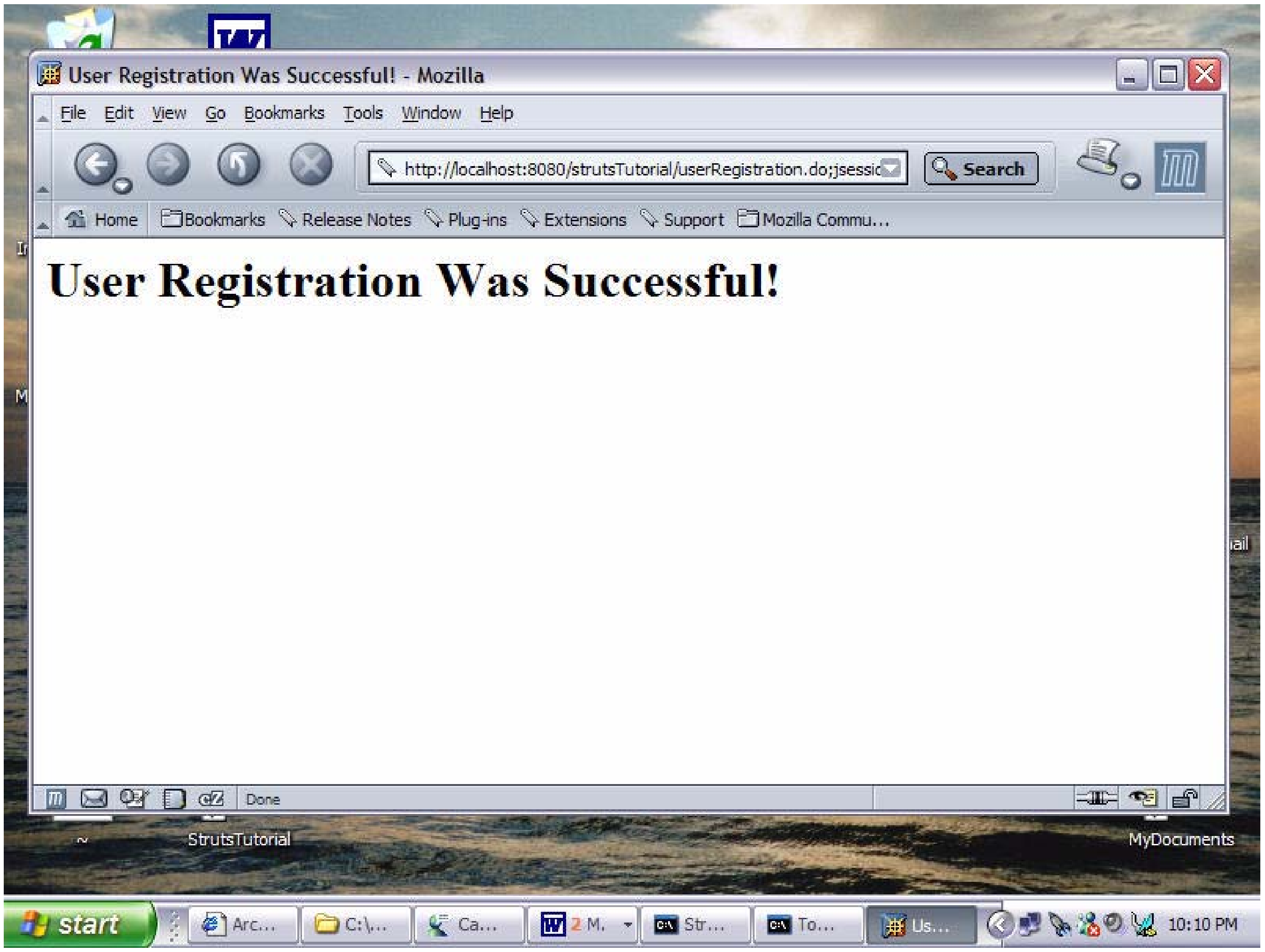
Go ahead and run the Ant build script as follows:

```
C:\strutsTutorial\WEB-INF\src> ant
```

If things go well, you should see the message BUILD SUCCESSFUL. Once you run the Ant build script with the default target, you should get a war file in your c:\projects\lib directory called strutsTutorial.war. Deploy this war file to your application server under the web context strutsTutorial. You will need to refer to your application server manual for more details on how to do this. For Tomcat and Resin, this is a simple matter of copying the war file to Tomcat's or Resin's home-dir/webapps directory. The webapps directory is under the server directory.

If you are not Ant savvy, you can simulate what this ant script does by setting your IDE to output the binary files to C:\strutsTutorial\WEB-INF\classes, and then zipping up the c:\strutsTutorial directory into a file called strutsTutorial.war.

Now that you have built and deployed your web application, test your new Action/forward combination by going to http://localhost:8080/strutsTutorial/userRegistration.do. You should see the following:


User Registration Was Successful! - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/strutsTutorial/userRegistration.do;jsessio
Search
Home Bookmarks Release Notes Plug-ins Extensions Support Mozilla Commu...
Done
StrutsTutorial
MyDocuments
start
10:10 PM

User Registration Was Successful!

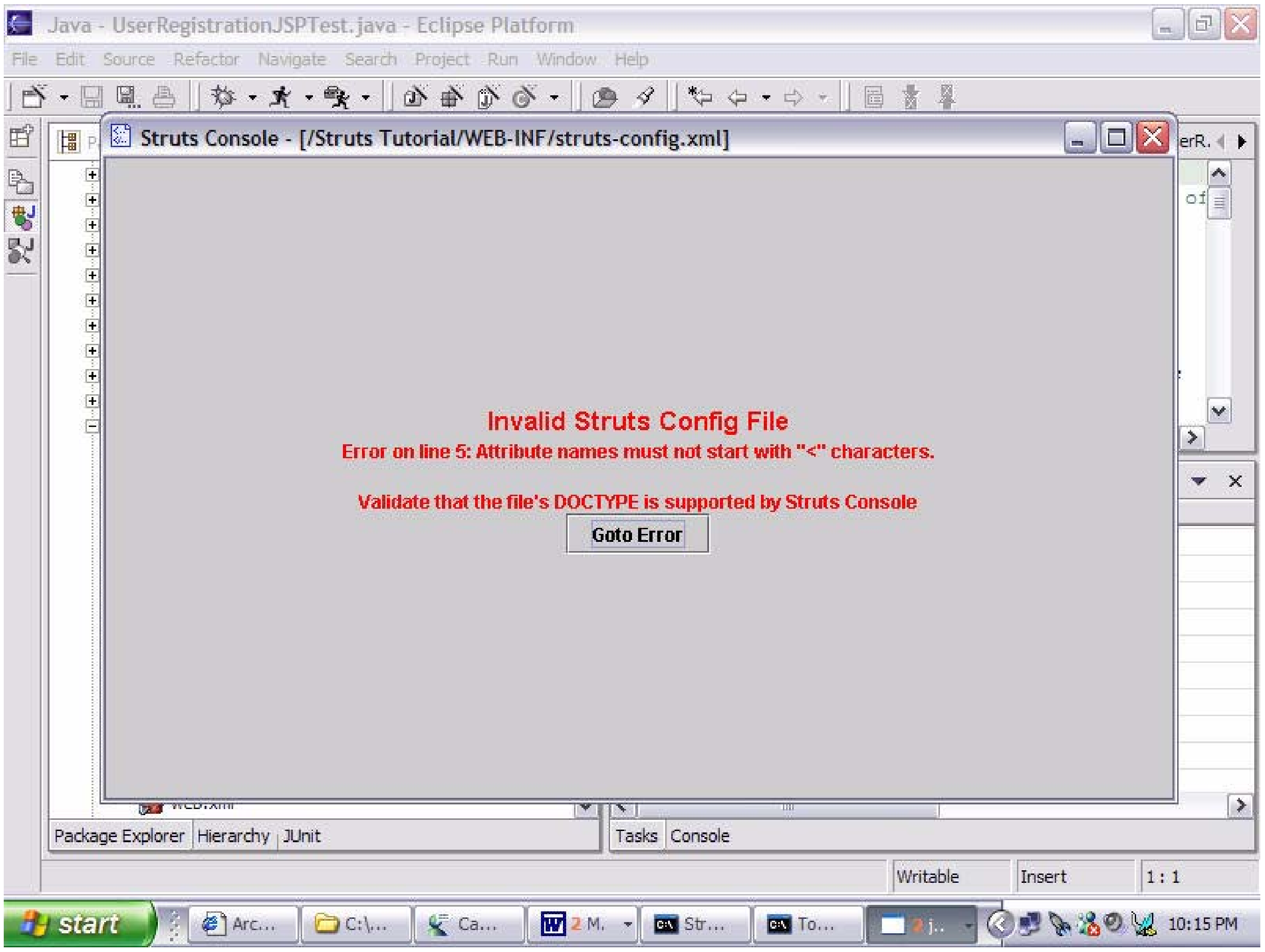
Figure 1.1 Running The Action for the first time

Congratulations! You have just written your first Struts Action. Now, admittedly this Action does not do much.

At this point, you may be having troubles. The most obvious problem is probably a misconfigured struts-config.xml file. There are two ways to solve this.

Debug Struts-Config.xml with the *Struts Console*

If you are new to XML, it may be a little hard to edit the struts-config.xml file. If you are having troubles with the struts-config.xml file, you should download the *Struts Console*. The *Struts Console* is a Swing based struts-config.xml editor; it provides a GUI for editing struts-config files. The *Struts Console* works as a plugin for JBuilder, NetBeans, Eclipse, IntelliJ and more. The *Struts Console* can be found at http://www.jamesh-olmes.com/struts/console/; follow the install instructions at the site. If you have a problem, you can edit the struts-config file with *Struts Console*. If there is a problem with the struts-config.xml file, the *Struts Console* will take you to the line / column of text that is having the problem. For example, here is an example of editing a struts-config file with a malformed XML attribute:


Java - UserRegistrationJSPTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Struts Console - [/Struts Tutorial/WEB-INF/struts-config.xml]
Goto Error
Package Explorer Hierarchy JUnit
Tasks Console
Writable Insert 1 : 1
start
10:15 PM

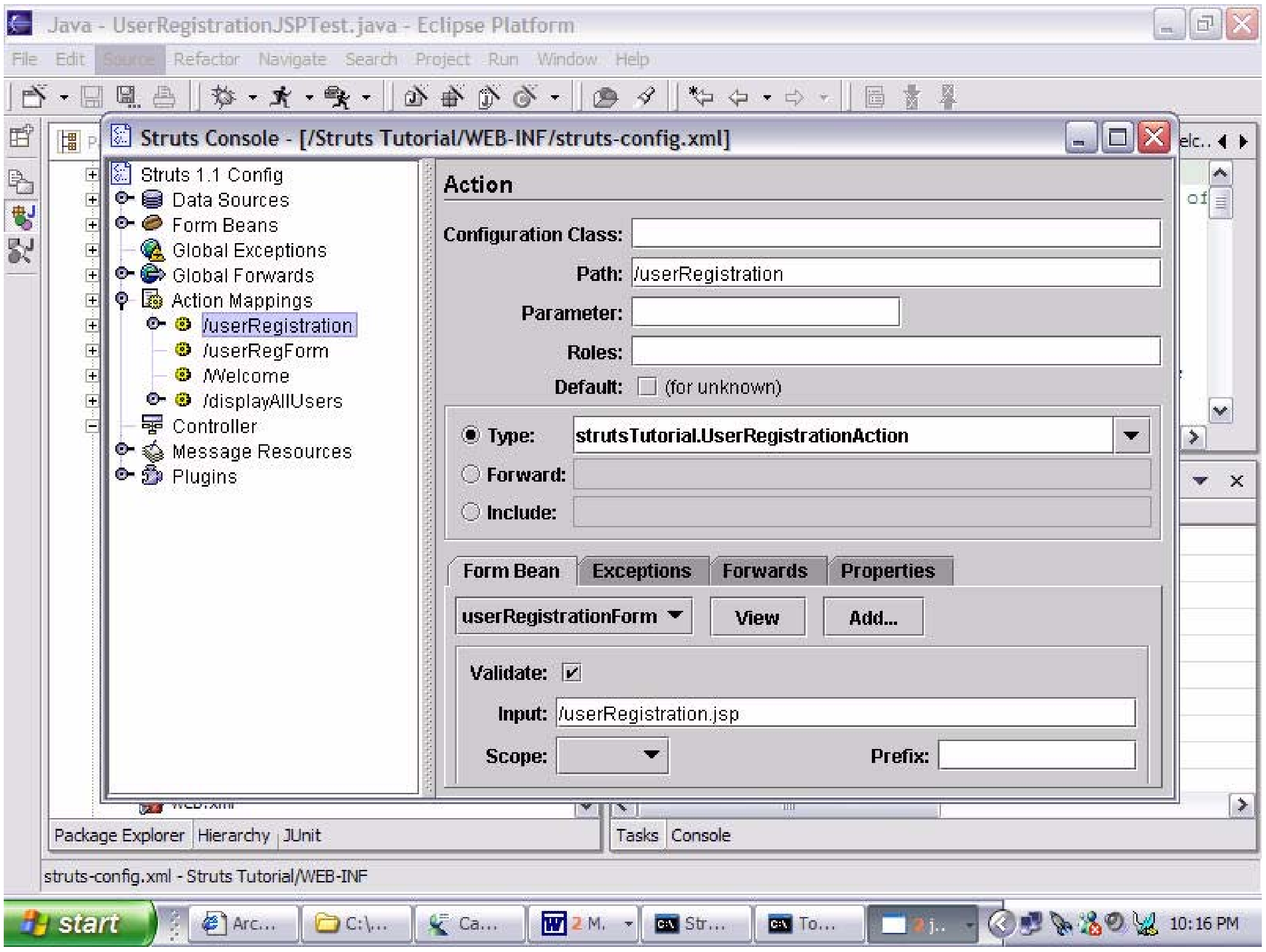
Invalid Struts Config File
Error on line 5: Attribute names must not start with "<" characters.

Validate that the file's DOCTYPE is supported by Struts Console

Figure 1.2 Running *Struts Console* against a malformed struts-config.xml file

Notice the Goto Error button. Clicking the button takes you to the exact line in the struts-config.xml file that is having the problem.

Once everything is fixed, you can view and edit the struts-config.xml file with the *Struts Console* as follows:



Configuration Class:
Path: /userRegistration
Parameter:
Roles:
Default: ☐ (for unknown)
● Type: strutsTutorial.UserRegistrationAction
○ Forward:
○ Include:
userRegistrationForm
Validate: ☑
Input: /userRegistration.jsp
Scope:
Prefix:

struts-config.xml - Struts Tutorial/WEB-INF

Figure 1.3 Running *Struts Console* in a happy world

The figure above shows editing struts-config.xml and inspecting the userRegistration action that you just configured.

Personally, I only use the *Struts Console* if there is a problem or I want to validate my struts-config.xml file without launching the application server. I prefer editing the struts-config.xml with a text editor, but find that the *Struts Console* comes in handy when there is a problem. In addition to mentoring new Struts developers and doing development myself, I teach a Struts course. I have found that *Struts Console* is extremely valuable to new Struts developers. Students (and new Struts developers) can easily make a small mistake that will cause the config file to fail. I can stare for a long time at a struts-config.xml file, and not find a “one off” error. Most of these errors, you will not make once you are a seasoned Struts developer, but they can be very hard to diagnose without the *Struts Console* when you are first getting started.

Another debugging technique is to use common logging to debug the application at runtime.

Add Logging Support with Log4J and Commons Logging

You may wonder why we dedicate a whole section to logging. Well to put it simply, when you are new to Struts, you will need to do more debugging, and logging can facilitate your debugging sessions dramatically.

The Struts framework uses Commons Logging throughout. Logging is a good way to learn what Struts does at runtime, and it helps you to debug problems. The Commons Logging framework works with many logging systems; mainly Java Logging that ships with JDK 1.4 and Log4J.

Using Struts without logging can be like driving in the fog with your bright lights, especially when something goes wrong. You will get a much better understanding how Struts works by examining the logs. Logging can be expensive. Log4J allows you to easily turn off logging at runtime.

Log4J is a full-featured logging system. It is easy to set up and use with Struts. You need to do several things:

1. Download Log4J
2. Unzip the Log4J distribution
3. Copy the log4j.jar file to c:\strutsTutorial\WEB-INF\lib
4. Create a log4j.properties file
5. Start using logging in our own classes

Like Struts, Log4J is an Apache Jakarta project. The Log4J home page is at http://jakarta.apache.org/log4j/docs/index.html. You can download Log4J from http://jakarta.apache.org/site/binindex.cgi. Search for Log4J on this page. Look for the link that looks like:

Log4j KEYS

- 1.2.8 zip PGP MD5
- 1.2.8 tar.gz PGP MD5

Click on the 1.2.8 ZIP file link. Download this file to c:\tools, and adjust accordingly for different drives, directories and operating systems (like *n?x). Copy the log4j-1.2.8.jar file located in C:\tools\jakarta-log4j-1.2.8\dist\lib to c:\strutsTutorial\WEB-INF\lib.

Now that you have the jar file in the right location you need to add log4j.properties files so that the web application classloader can find it. The Ant script copies all properties (*.properties) files from \WEB-INF\src\java to \WEB-INF\classes. The Ant script also deletes the \WEB-INF\classes directory. Thus, create a log4j.properties file in the \WEB-INF\src\java directory with the following contents:

```
log4j.rootLogger=WARN, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n
```

The code above sends output to the stdout of the application with the priority, date time stamp, file name, method name, line number and the log message. Logging is turned on for classes under the Struts package and the strutsTutorial code.

To learn more about Log4J, read the online documentation at http://jakarta.apache.org/log4j/docs/manual.html, and then read the JavaDoc at http://jakarta.apache.org/log4j/docs/api/index.html. Look up the class org.apache.log4j.PatternLayout in the JavaDocs at the top of the file is a list of conversion characters for the output log pattern. You can use the conversion characters to customize what gets output to the log.

Putting log4j on the classpath (copying the jar file to WEB-INF\lib), causes the Commons Logging to use it. The log4J framework finds the log4j.properties file and uses it to create the output logger.

If you start having problems with Struts, then set up the logging level of Struts to debug by adding the following line to log4j.properties file:

```
log4j.logger.org.apache.struts=DEBUG
```

Underneath the covers, we are using Log4J. However, if we want to use logging in our own code, we should use Commons Logging, which allows us to switch to other logging systems if necessary. Thus, we will use the Commons Logging API in our own code. To learn more about Commons Logging, read the “short online manual” at http://jakarta.apache.org/commons/logging/userguide.html.

Edit the UserRegistrationAction by importing the two Commons Logging classes and putting a trace call in the execute method as follows:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

…
private static Log log = LogFactory.getLog(UserRegistrationAction.class);

public ActionForward execute(...) throws Exception {
  log.trace("In execute method of UserRegistrationAction");
  return mapping.findForward("success");
}
```

You will need to add the Commons Logging Jar file to the compile.classpath in \WEB-INF\src\java\build.xml file as follows:

```
<path id="compile.classpath">
    <pathelement path ="lib/commons-beanutils.jar"/>
    <pathelement path ="lib/commons-digester.jar"/>
    <pathelement path ="lib/struts.jar"/>
    <pathelement path ="classes"/>
    <pathelement path ="${classpath}"/>
    <pathelement path ="${servlet.jar}"/>
    <pathelement path ="lib/commons-logging.jar"/>
</path>
```

Then to get the above trace statement to print out to the log you need to add this line to the log4j.properties file:

```
log4j.logger.strutsTutorial=DEBUG
```

Rebuild and deploy the war file, re-enter the url to exercise the action class and look for the log line in the app server’s console output.

There are six levels of logging as follows listed in order of importance: `fatal`, `error`, `warn`, `info`, `debug` and `trace`. The log object has the following methods that you can use to log messages.

```
    log.fatal(Object message);
    log.fatal(Object message, Throwable t);
    log.error(Object message);
    log.error(Object message, Throwable t);
    log.warn(Object message);
    log.warn(Object message, Throwable t);
    log.info(Object message);
    log.info(Object message, Throwable t);
    log.debug(Object message);
    log.debug(Object message, Throwable t);
    log.trace(Object message);
    log.trace(Object message, Throwable t);
```

Logging is nearly essential for debugging Struts applications. You must use logging; otherwise, your debugging sessions may be like flying blind.

Write Your First ActionForm

ActionForms represent request data coming from the browser. ActionForms are used to populate HTML forms to display to end users and to collect data from HTML forms. In order to create an ActionForm, you need to follow these steps:

1. Create a new class in the `strutsTutorial` package called `UserRegistrationForm` that subclasses `org.apache.struts.action.ActionForm` as follows:

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;

import javax.servlet.http.HttpServletRequest;

public class UserRegistrationForm extends ActionForm {
```

2. Now you need to create JavaBean properties for all the fields that you want to collect from the HTML form. Let's create firstName, lastName, userName, password, passwordCheck (make sure they entered the right password), e-mail, phone, fax and registered (whether or not they are already registered) properties. Add the following fields:

```
private String firstName;
private String lastName;
private String userName;
private String password;
private String passwordCheck;
private String email;
private String phone;
private String fax;
private boolean registered;
```

Add getter and setter methods for each field as follows:

```
public String getEmail() {
return email;
}
public void setEmail(String string) {
email = string;
}
```

3. Now you have defined all of the properties for the form. (Reminder: each getter and setter pair defines a property.) Next, you need to override the reset method. The reset method gets called each time a request is made. The reset method allows you to reset the fields to their default value. Here is an example of overwriting the reset method of the ActionForm:

```
public void reset(ActionMapping mapping,
                       HttpServletRequest request) {
firstName=null;
lastName=null;
userName=null;
password=null;
passwordCheck=null;
email=null;
phone=null;
fax=null;
registered=false;
}
```

   If you like, please print out a trace method to this method using the logging API, e.g., `log.trace("reset")`.

4. Next you need validate the user entered valid values. In order to do this, you need to override the `validate` method as follows:

```
public ActionErrors validate(
                    ActionMapping mapping,
                    HttpServletRequest request) {
ActionErrors errors = new ActionErrors();

if (firstName==null
|| firstName.trim().equals("")){
errors.add("firstName",
               new ActionError(
                  "userRegistration.firstName.problem"));
}
...
return errors;
}
```

   The validate method returns a list of errors (ActionErrors). The ActionErrors display on the input HTML form. You use your Java programming skills to validate if their typing skills are up to task. The above code checks to see that `firstName` is present; if it is not present (i.e., it is null or blank), then you add an ActionError to the ActionErrors collection. Notice that when you construct an ActionError object, you must pass it a key into the resource bundle (“userRegistration.firstName”). Thus, we need to add a value to this key in the Resource bundle.

Please open the file C:\strutsTutorial\WEB-INF\src\java\resources\application.properties. Add a key value pair as follows:

```
userRegistration.firstName.problem=The first name was blank
```

If the `firstName` is blank, the control gets redirected back to the input form, and the above message displays. Using a similar technique, validate all the fields.

Write Your First Input View (JSP Page)

Next, we want to create an HTML form in JSP that will act as the input to our Action. The input is like the input view, while the forwards are like output views. In order to create the input view, you will do the following:

1. Create a JSP page called userRegistration.jsp in the c:\strutsTutorial directory.
2. Import both the Struts HTML and Struts Bean tag libraries. The tag libraries have already been imported into the web.xml file as follows:

   ```
   <taglib>
     <taglib-uri>/tags/struts-bean</taglib-uri>
     <taglib-location>/WEB-INF/struts-bean.tld
     </taglib-location>
   </taglib>

   <taglib>
     <taglib-uri>/tags/struts-html</taglib-uri>
     <taglib-location>/WEB-INF/struts-html.tld
     </taglib-location>
   </taglib>
   ```

   One of the advantages of using the blank.war file is that all the things you need are already configured. You just add the parts that are needed for your application. To import the two tag libraries, you would use the taglib directive as follows:

   ```
   <%@ taglib uri="/tags/struts-html" prefix="html"%>
   <%@ taglib uri="/tags/struts-bean" prefix="bean"%>
   ```
3. Use the `html:form` tag to associate the form with the Action. The `html:form` tag associates the form to an action mapping. You use the action attribute to specify the path of the action mapping as follows:

   ```
   <html:form action="userRegistration">
   ```

4. Output the errors associated with this form with the `html:errors` tag. ActionForms have a validate method that can return ActionErrors. Add this to the JSP:

   ```
   <html:errors/>
   ```

   Note there are better ways to do this. Struts 1.1 added `html:messages`, which is nicer as it allows you to get the markup language out of the resource bundle. This is covered in more detail later.

5. Update the Action to associate the Action with the ActionForm and input JSP. In order to do this, you need to edit the struts-config.xml file. If you do not feel comfortable editing an XML file, then use the *Struts Console*. Add the following form-bean element under the form-beans element as follows:

   ```
   <form-bean name="userRegistrationForm"
              type="strutsTutorial.UserRegistrationForm" />
   ```

   The code above binds the name userRegistration to the form you created earlier: `strutsTutorial.UserRegistrationForm`.

   Now that you have added the form-bean element, you need to associate the `userRegistration` action mapping with this form as follows:

   ```
   <action path="/userRegistration"
           type="strutsTutorial.UserRegistrationAction"
           name="userRegistrationForm"
           input="/userRegistration.jsp">

       <forward name="success" path="/regSuccess.jsp" />
   </action>
   ```

   Notice the addition of the `name` and `input` attributes. The name attribute associates this action mapping with the userRegistrationForm ActionForm that you defined earlier. The input attribute associates this action mapping with the input JSP. If there are any validation errors, the execute method of the action will not get called; instead the control will go back to the userRegistration.jsp file until the form has no ActionErrors associated with it.

6. Create the labels for the Form fields in the resource bundle. Each field needs to have a label associated with it. Add the following to the resource bundle (c:/strutsTutorial/WEB-INF/src/java/resources/application.properties):

   ```
   userRegistration.firstName=First Name
   userRegistration.lastName=Last Name
   userRegistration.userName=User Name
   userRegistration.password=Password
   userRegistration.email=Email
   userRegistration.phone=Phone
   userRegistration.fax=Fax
   ```

   You could instead hard code the values in the JSP page. Putting the value in the resource bundle allows you to internationalize your application.

7. Use the `bean:message` to output the labels. When you want to output labels in the JSP from the resource bundle, you can use the `bean:message` tag. The `bean:message` tag looks up the value in the resource bundle and outputs it from the JSP. The following outputs the label for the firstName from the resource bundle:

   ```
   <bean:message key="userRegistration.firstName" />
   ```

8. Use the `html:text` tag to associate the ActionForm's properties to the HTML form's fields. The `html:text` associates an HTML text field with an ActionForm property as follows:

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

   The above associates the HTML text field with the firstName property from your ActionForm. The `html:form` tag is associated with the ActionForm via the action mapping. The individual text fields are associated with the ActionForm's properties using the `html:text` tag.

9. Create an `html:submit` tag and an `html:cancel` tag to render a submit button and a cancel button in html as follows:

   ```
   <html:submit />
   ...
   <html:cancel />
   ```

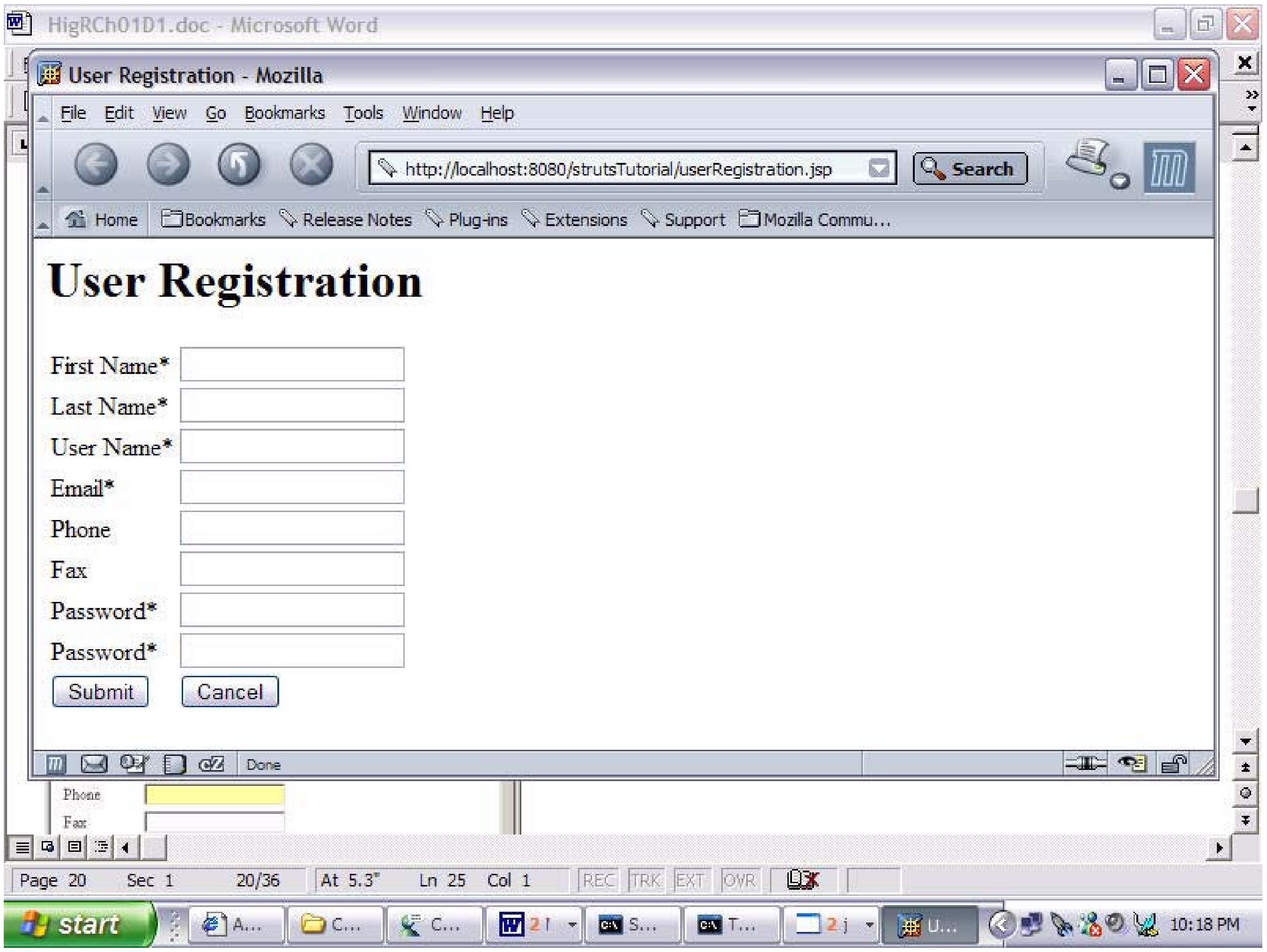
   At this point you should be able to deploy and test your Struts application. The Action has not been wired to do much of anything yet. But the form will submit to the Action. And, if a form field is invalid the control will be forwarded back to the input form. Try this out by leaving the firstName field blank.

If you are having problems, you may want to compare what you have written to the solution. Here is what the userRegistration.jsp looks like after you finish (your HTML may look different):

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<html>

  <head>
    <title>User Registration</title>
  </head>

  <body>
    <h1>User Registration</h1>

<html:errors/>

    <table>
    <html:form action="userRegistration">
      <tr>
        <td>
         <bean:message key="userRegistration.firstName" />*
        </td>
        <td>
          <html:text property="firstName" />
        </td>
      </tr>
        <td>
          <bean:message key="userRegistration.lastName" />*
        </td>
        <td>
          <html:text property="lastName" />
        </td>
      <tr>
        <td>
          <bean:message key="userRegistration.userName" />*
        </td>
        <td>
          <html:text property="userName" />
        </td>
      </tr>
      <tr>
        <td>
          <bean:message key="userRegistration.email" />*
        </td>
        <td>
          <html:text property="email" />
        </td>
      </tr>
      <tr>
```

```
          <td>
            <bean:message key="userRegistration.phone" />
          </td>
          <td>
            <html:text property="phone" />
          </td>
        </tr>
        <tr>
          <td>
            <bean:message key="userRegistration.fax" />
          </td>
          <td>
            <html:text property="fax" />
          </td>
        </tr>
        <tr>
          <td>
            <bean:message key="userRegistration.password" />*
          </td>
          <td>
            <html:password property="password" />
          </td>
        </tr>
        <tr>
          <td>
            <bean:message key="userRegistration.password" />*
          </td>
          <td>
            <html:password property="passwordCheck" />
          </td>
        </tr>
        <tr>
          <td>
            <html:submit />
          </td>
          <td>
            <html:cancel />
          </td>
        </tr>

      </html:form>
      </table>
    </body>
  </html>
```

The form should look like this when it first gets loaded. (You load the form by going to http://localhost:8080/strutsTutorial/userRegistration.jsp.)


HigRCh01D1.doc - Microsoft Word
User Registration - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/strutsTutorial/userRegistration.jsp
Search
Home Bookmarks Release Notes Plug-ins Extensions Support Mozilla Commu...
Done
Page 20 Sec 1 20/36 At 5.3" Ln 25 Col 1 REC TRK EXT OVR
start
10:18 PM

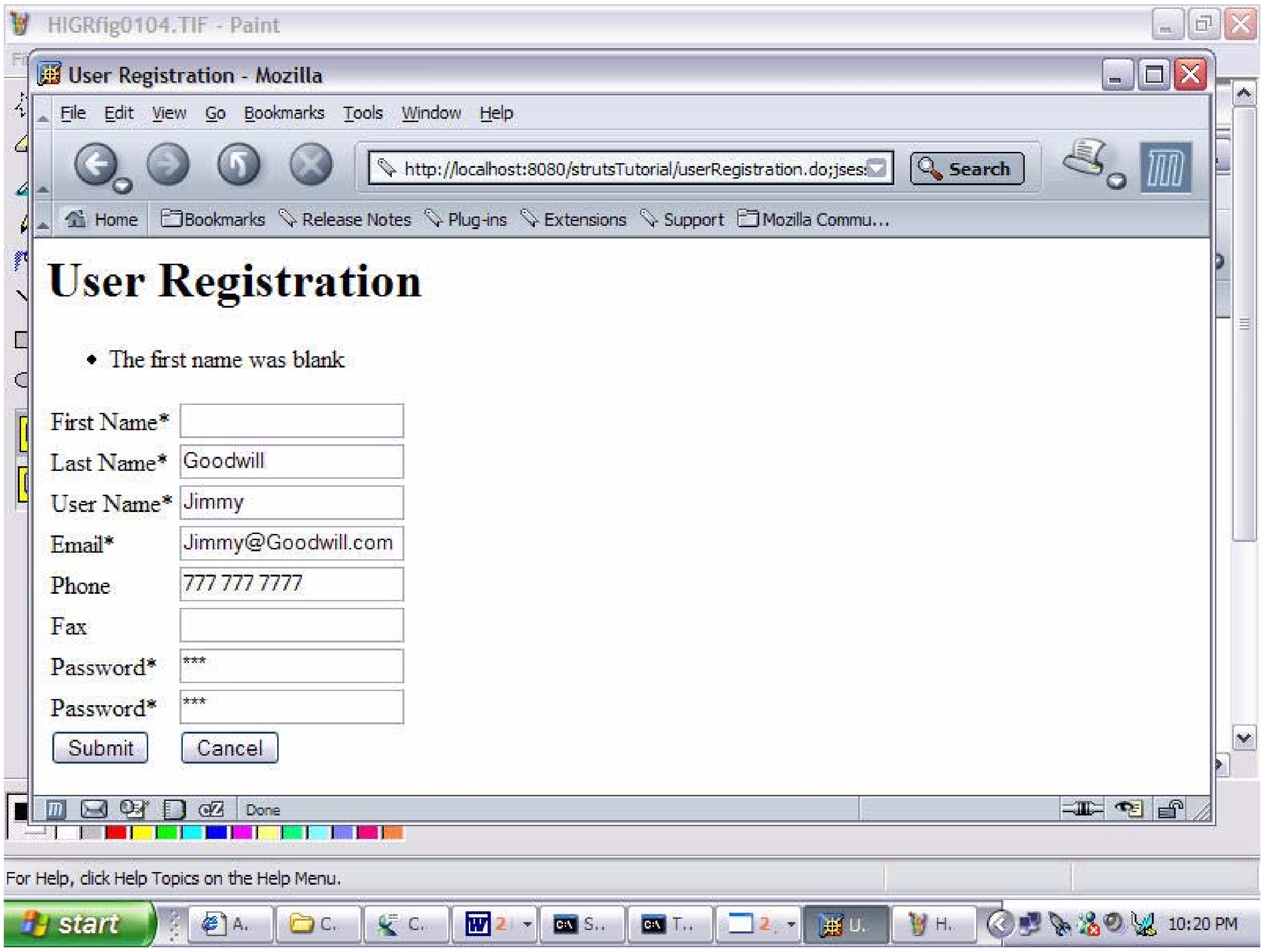
User Registration

First Name*	
Last Name*	
User Name*	
Email*	
Phone	
Fax	
Password*	
Password*	

Submit Cancel

Figure 1.4 User Registration JSP

If you leave the firstName blank, then you should get a form that looks like this.


HIGRfig0104.TIF - Paint
User Registration - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/strutsTutorial/userRegistration.do;jses
Search
Home Bookmarks Release Notes Plug-ins Extensions Support Mozilla Commu...
Done
For Help, click Help Topics on the Help Menu.
start
10:20 PM

User Registration

- The first name was blank

First Name*	
Last Name*	Goodwill
User Name*	Jimmy
Email*	Jimmy@Goodwill.com
Phone	777 777 7777
Fax	
Password*	***
Password*	***

Submit Cancel

Figure 1.5 User Registration JSP with validation errors

Notice that the error message associated with the firstName displays, since the firstName was left blank. It is instructive to view the logs as you run the example to see the underlying interactions of the Struts framework. Once you complete the form and hit the Submit button, the execute method of gets `UserRegistrationAction` invoked. Currently the execute method just forwards to `regSuccess.jsp`, which is mapped into the success forward, whether or not the Cancel button is pressed.

Update the Action to Handle the Form and Cancel Buttons

Let’s do something with the ActionForm that gets passed to the Action. Once you fill in the form correctly (no validation errors) and hit the submit button, the execute method of the `UserRegistrationAction` is invoked. Actually, the execute method gets invoked whether or not you hit the submit button or the cancel button.

You need check to see if the cancel button was clicked; it was clicked forward to welcome. The welcome forward was setup by the authors of blank.war, and it forwards to “/Welcome.do”, which forwards to /pages/Welcome.jsp. Check out the struts-config.xml file to figure out how they did this. To check and see if the cancel button was clicked, you need to use the `isCanceled` method of the Action class in the execute method as follows:

```
public ActionForward execute(...)...{
                ...
if (isCancelled(request)){
log.debug("Cancel Button was pushed!");
return mapping.findForward("welcome");
}
                ...
}
```

The `isCancelled` method takes an HttpServletRequest as an argument. The execute method was passed an HttpServerletRequest.

Next, you need to cast the ActionForm to an `UserRegistrationForm` . In order to use the form that was submitted to the action, you need to cast the ActionForm to the proper type. Thus, you will need to cast the ActionForm that was passed to the execute method to a UserRegistrationForm as follows:

```
UserRegistrationForm userForm =
                   (UserRegistrationForm) form;
```

Now you can start using the UserRegistrationForm like this:

```
log.debug("userForm firstName" + userForm.getFirstName());
```

For now, just print out the firstName with the logging utility. In the next section, you’ll do something more useful with this form—you will write it to a database.

Set up the Database Pooling with Struts

This is an optional section. You can skip this section and continue the tutorial. However, if you want to use Struts DataSources, then skipping this session is not an option.

Struts DataSources allow you to get the benefits of data sources without being tied to any one vendor's implementation. On one hand, it can make your application more portable as configuring J2EE datasources is vendor specific. On the other hand, you cannot use some of the more advanced features of your specific vendor's implementation. If you are not sure if you are going to use Struts DataSources, then follow along and make your decision after this section.

You need both commons-pooling and commons-dbcp to get Struts Datasources to work, but this is not mentioned in the online documentation, and the required jar files do not ship with Blank.war or with Struts 1.1 at all. In fact, Struts 1.1., requires the Struts Legacy jar file. Note that if you are using Tomcat5, then both commons-pooling and commons-dbcp ship in the common/lib folder. If you are using another application server with Struts Datasources, you need to download them.

Currently neither http://jakarta.apache.org/struts/userGuide/configuration.html#data-source_config nor http://jakarta.apache.org/struts/faqs/database.html make note of this.

Struts Datasources require commons-pooling and commons-dbcp. The blank.war does not have commons-pooling or commons-dbcp. If you want to use Struts datasource, you will need to download commons-pooling and commons-dbcp from the following locations:

- http://jakarta.apache.org/site/binindex.cgi#commons-dbcp
- http://jakarta.apache.org/site/binindex.cgi#commons-pool

Download the above archives, and extract them. Then, copy the jar files commons-pool-1.1.jar, and commons-dbcp-1.1.jar from the archives into the WEB-INF/lib directory of your web application. Note that Tomcat 5 ships with commons-dbcp and commons-pool so you do not need to download and install commons-dbcp and commons-pool if you are using Tomcat 5.

In addition to the above two jar files, you will need to use the struts-legacy.jar jar file that ships with Struts 1.1. Copy the struts-legacy.jar file (C:\tools\jakarta-struts-1.1\lib\ struts-legacy.jar) to the WEB-INF/lib directory of your web application.

This might be a moot point depending on how soon Struts 1.2 comes out and if commons-pooling and commons-dbcp ship with Struts 1.2.

You need to create a table in your database of choice, similar to the one whose DDL is listed below:

```
CREATE TABLE USER(
  EMAIL VARCHAR(80),
  FIRST_NAME VARCHAR(80),
  LAST_NAME VARCHAR(80),
  PASSWORD VARCHAR(11),
  PHONE VARCHAR(11),
  FAX VARCHAR(11),
  CONSTRAINT USER_PK PRIMARY KEY (EMAIL)
);
```

You need to obtain a JDBC driver for your database of choice and obtain the JDBC URL. Then using the JDBC driver name and the JDBC URL, write the following in your Struts-Config.xml file (before the formbeans element).

```
<data-sources>
  <data-source
      type="org.apache.commons.dbcp.BasicDataSource"
      key="userDB">

    <set-property property="driverClassName"
                  value="org.hsqldb.jdbcDriver" />
    <set-property property="url"
              value="jdbc:hsqldb:c:/strutsTutorial/db" />
    <set-property property="username" value="sa" />
    <set-property property="password" value="" />
  </data-source>
</data-sources>
```

Notice that the set-property sets a property called driverClassName, which is the class name of your JDBC driver, and set-property sets the url, which is the JDBC URL, of the database that has your new table. Also, note that you will need to specify the username and password for the database using set-property.

You will need to copy the jar file that contains the JDBC driver onto the classpath somehow. Most application servers have a folder that contains files that are shared by all web applications; you can put the JDBC driver jar file there.

Note I added an Ant target called builddb that creates the table with DDL using the ant task `sql`. Thus, I added the following to the Ant script to create the database table. You can use a similar technique or use the tools that ship with your database:

```
<target name="builddb"
        description="builds the database tables">
      <sql driver="org.hsqldb.jdbcDriver"
           userid="sa"
           password=""
           url="jdbc:hsqldb:c:/strutsTutorial/db">
        <classpath>
            <pathelement
              path="/tools/hsqldb/lib/hsqldb.jar"/>
        </classpath>

CREATE TABLE USER(
  EMAIL VARCHAR(80),
  FIRST_NAME VARCHAR(80),
  LAST_NAME VARCHAR(80),
  PASSWORD VARCHAR(11),
  PHONE VARCHAR(11),
  FAX VARCHAR(11),
  CONSTRAINT USER_PK PRIMARY KEY
                      (EMAIL)
);

    </sql>
</target>
```

Note that I used HSQL DB as my database engine, which can be found at http://hsqldb.sourceforge.net/. You should be able to use any database that has a suitable JDBC driver.

Once you have the database installed and configured, you can start accessing it inside of an Action as follows:

```
import java.sql.Connection;
import javax.sql.DataSource;
import java.sql.PreparedStatement;

...
public ActionForward execute(...)...{
    DataSource dataSource =
           getDataSource(request, "userDB");

     Connection conn = dataSource.getConnection();
```

Warning! At this point, I strongly recommend that you never put database access code inside of an Action's execute method, except in a tutorial. This should almost always be delegated to a Data Access Object. However, for the sake of simplicity, go ahead and insert the UserRegistrationForm into that database inside of the `execute` method as follows:

```
UserRegistrationForm userForm = (UserRegistrationForm) form;

log.debug("userForm firstName" + userForm.getFirstName());

DataSource dataSource = getDataSource(request, "userDB");
Connection conn = dataSource.getConnection();

try{
PreparedStatement statement = conn.prepareStatement(
"insert into USER " +
"(EMAIL, FIRST_NAME, LAST_NAME, PASSWORD, PHONE, FAX)" +
" values (?,?,?,?,?,?)");

statement.setString(1,userForm.getEmail());
statement.setString(2,userForm.getFirstName());
statement.setString(3,userForm.getLastName());
statement.setString(4,userForm.getPassword());
statement.setString(5,userForm.getPhone());
statement.setString(6,userForm.getFax());
statement.executeUpdate();
}finally{
conn.close();
}

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

Now, test and deploy the application. If you have gotten this far and things are working, then you have made some good progress. So far you have created an Action, ActionForm, ActionMapping and set up a Struts Data-source.

Exception Handling with Struts

Bad things happen to good programs. It is our job as fearless Struts programmers to prevent these bad things from showing up to the end user. You probably do not want the end user of your system to see a Stack Trace. An end user seeing a Stack Trace is like any computer user seeing the "Blue Screen of Death" (generally not a very pleasant experience for anyone).

It just so happens that when you enter an e-mail address into two User Registrations, you get a nasty error message as the e-mail address is the primary key of the database table. Now, one could argue that this is not a true "exceptional" condition, as it can happen during the normal use of the application, but this is not a tutorial on design issues. This is a tutorial on Struts, and this situation gives us an excellent opportunity to explain Struts declarative exception handling.

If you enter in the same e-mail address twice into two User Registrations, the system will throw a `java.sql.SQLException`. In Struts, you can set up an exception handler to handle an exception.

An exception handler allows you to declaratively handle an exception in the struts-config.xml file by associating an exception to a user friendly message and a user friendly JSP page that will display if the exception occurs.

Let's set up an exception handler for this situation. Follow these steps:

1. Create a JSP file called userRegistrationException.jsp in the root directory of the project (c:\strutsTutorial).

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>

<html>

<head>
<title>
User Registration Had Some Problems
</title>
</head>

<body>
<h1>User Registration Had Some Problems!</h1>
<html:errors/>
</body>

</html>
```

Notice the use of `html:errors` to display the error message associated with the exception.

2. Add an entry in the resource bundle under the key `userRegistration.sql.exception` that explains the nature of the problem in terms that the end user understands. This message will be used by the exception handler. Specifically, you can display this message using the html:errors tag in the userRegistrationException.jsp file. Edit the properties file associated with the resource bundle (located at C:\strutsTutorial\WEB-INF\src\java\resources\application.properties if you have been following along with the home game version of the Struts tutorial).

   ```
   userRegistration.sql.exception=There was a problem adding the User. \n The
   most likely problem is the user already exists or the email\n address is
   being used by another user.
   ```

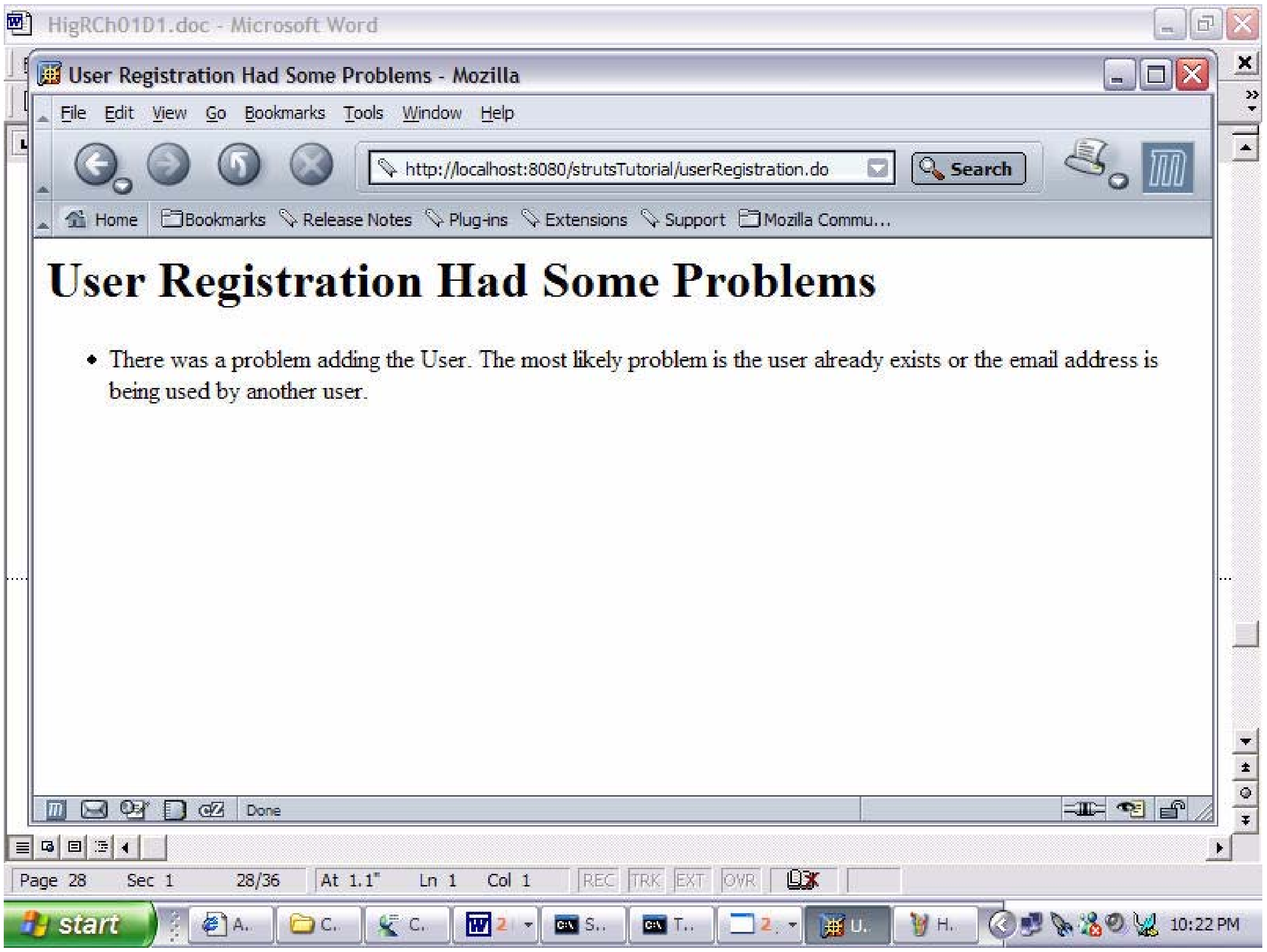
   (The code above is all one line.)

3. Add an exception handler in the action mapping for `/userRegistration` that handles `java.sql.SQLException` as follows:

   ```
   <action path="/userRegistration"
               type="strutsTutorial.UserRegistrationAction"
               name="userRegistrationForm"
               input="/userRegistration.jsp">
     <exception type="java.sql.SQLException"
               key="userRegistration.sql.exception"
               path="/userRegistrationException.jsp" />

     <forward name="success" path="/regSuccess.jsp" />
     <forward name="failure" path="/regFailure.jsp" />
   </action>
   ```

   Notice that you add the exception handler by using the exception element (highlighted above). The above exception element has three attributes: type, key and path. The type attribute associates this exception handler with the exception `java.sql.SQLException`. The key attribute associates the exception handler with a user friendly message out of the resource bundle. The path attribute associates the exception handler with the page that will display if the exception occurs.

If you do everything right, you get the following when the exception occurs.


HigRCh01D1.doc - Microsoft Word
User Registration Had Some Problems - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/strutsTutorial/userRegistration.do
Search
Home Bookmarks Release Notes Plug-ins Extensions Support Mozilla Commu...
Done
Page 28 Sec 1 28/36 At 1.1" Ln 1 Col 1 REC TRK EXT OVR
start
10:22 PM

User Registration Had Some Problems

- There was a problem adding the User. The most likely problem is the user already exists or the email address is being used by another user.

Figure 1.6 Declaritive Exception Handling

Display an Object with Struts Tags

Struts supports a Model 2 architecture. The Actions interact with the model and perform control flow operations, like which view is the next view to display. Then, Actions delegate to JSP (or other technologies) to display objects from the model.

To start using Struts with this tutorial, follow these steps:

1. Add an attribute called `attribute` to the mapping that causes the ActionForm to be mapped into scope as follows:

   ```
   <action path="/userRegistration"
           type="strutsTutorial.UserRegistrationAction"
           name="userRegistrationForm"
           attribute="user"
           input="/userRegistration.jsp">
           ...
   ```

   Notice the above action mapping uses the attribute called `attribute`. The attribute maps the ActionForm into a scope (session scope by default) under “user”. Now that the ActionForm is in session scope, you can display properties from the ActionForm in the view.

2. Edit the regSuccess.jsp that you created earlier. The regSuccess.jsp is an ActionForward for the UserRegistrationAction. The regSuccess.jsp is the output view for the Action. In order to display the ActionForm, you could use the Struts bean tag library.
3. Import the bean tag library into the JSP as follows:

   ```
   <%@ taglib uri="/tags/struts-bean" prefix="bean"%>
   ```

4. Use the `bean:write` to output properties of the ActionForm

   ```
   <bean:write name="user" property="firstName" />
   ```

   The code above prints out the `firstName` property of the user object. Use the above technique to print out all of the properties of the user object.

   When you are done with the JSP, it should look something like this:

   ```
   <%@ taglib uri="/tags/struts-bean" prefix="bean"%>

   <html>

   <head>
   <title>
   User Registration Was Successful!
   ```

```
</title>
</head>

<body>
<h1>User Registration Was Successful!</h1>
</body>

<table>
  <tr>

    <td>
    <bean:message key="userRegistration.firstName" />
    </td>

<td>
<bean:write name="user" property="firstName" />
</td>

  </tr>

  <tr>

    <td>
    <bean:message key="userRegistration.lastName" />
    </td>

<td>
<bean:write name="user" property="lastName" />
</td>

  </tr>

  <tr>

    <td>
    <bean:message key="userRegistration.email" />
    </td>

<td>
<bean:write name="user" property="email" />
</td>

  </tr>

</table>

</html>
```

Using Logic Tags to Iterate over Users

Struts provides logic tags that enable you to have display logic in your view without putting Java code in your JSP with Java scriptlets. To start using the Logic tags, follow these steps.

1. Create a JavaBean class called User to hold a user with email, firstName and lastName properties. Here is a possible implementation (partial listing):

```
package strutsTutorial;

import java.io.Serializable;


public class User implements Serializable {

private String lastName;
private String firstName;
private String email;

public String getEmail() {
return email;
}

     ...
public void setEmail(String string) {
email = string;
}
     ...
}
```

2. Create a new Action called `DisplayAllUsersAction`.

```
public class DisplayAllUsersAction extends Action {
```

In the new Action's `execute` method, complete the following steps:

3. Get the `userDB` datasource.

```
DataSource dataSource = getDataSource(request, "userDB");
```

4. Create a DB connection using the datasource.

```
Connection conn = dataSource.getConnection();
Statement statement = conn.createStatement();
```

5. Query the DB, and copy the results into a collection of the `User` JavaBean:

```
ResultSet rs =
statement.executeQuery("select FIRST_NAME, LAST_NAME, EMAIL from USER");

List list = new ArrayList(50);

while (rs.next()){
    String firstName = rs.getString(1);
    String lastName = rs.getString(2);
    String email = rs.getString(3);

    User user = new User();
    user.setEmail(email);
    user.setFirstName(firstName);
    user.setLastName(lastName);
    list.add(user);
}
                           if (list.size() > 0){
                             request.setAttribute("users", list);
                         }
```

Tip: Don’t forget to close the connection using a try/finally block.

Warning! You do not typically put SQL statements and JDBC code directly in an Action. This type of code should be in a DataAccessObject. A DataAccessObject would encapsulate the CRUD access for a particular domain object. The DataAccessObject is part of the Model of the application.

6. Create a new JSP called userRegistrationList.jsp.

In the new JSP, perform the following steps:

7. Import the logic tag library into the userRegistrationList.jsp.

```
<%@ taglib uri="/tags/struts-logic" prefix="logic"%>
```

8. Check to see if the users are in scope with the `logic:present` tag.

```
<logic:present name="users">
     ... (Step 9 goes here)
</logic:present>
```

9. If the users are in scope, iterate through them.

```
<logic:iterate id="user" name="users">
     ... (Step 10 goes here)
</logic:iterate>
```

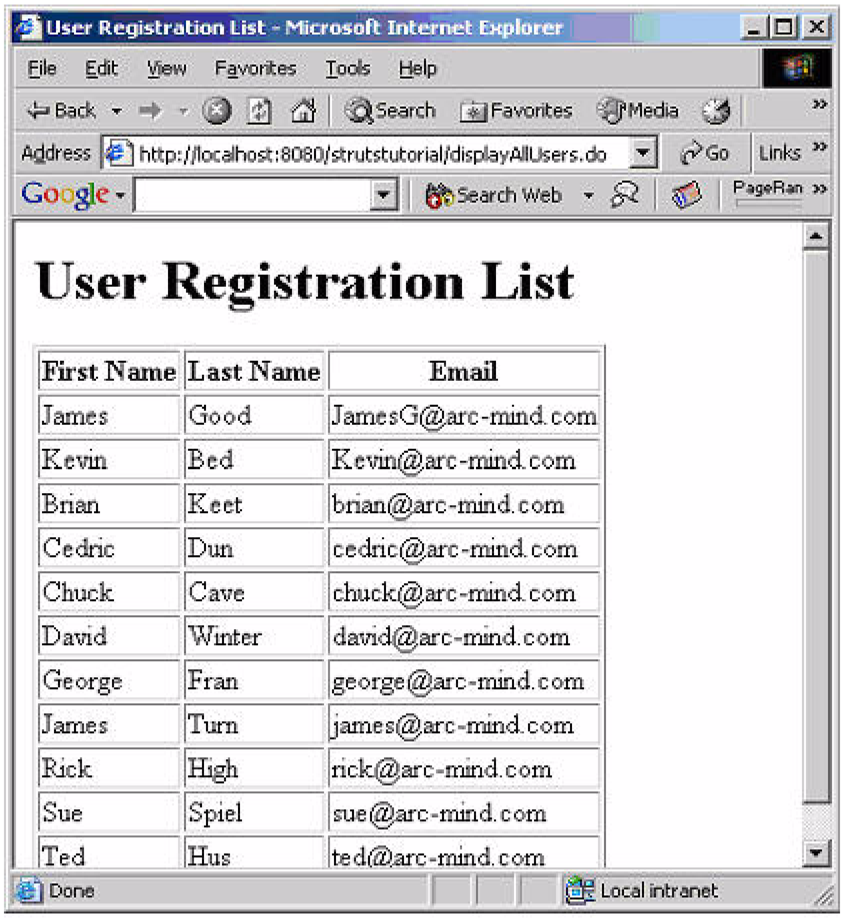
10. For each iteration, print out the firstName, lastName and email using `bean:write`

```
<bean:write name="user"
           property="firstName"/>

<bean:write name="user"
           property="lastName"/>

<bean:write name="user"
           property="email"/>
```

One possible implementation for the JSP is as follows:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<%@ taglib uri="/tags/struts-logic" prefix="logic"%>

<html>

  <head>
    <title>User Registration List</title>
  </head>

  <body>
    <h1>User Registration List</h1>

<logic:present name="users">

  <table border="1">
    <tr>
      <th>
       <bean:message
                   key="userRegistration.firstName"/>
      </th>

      <th>
       <bean:message
                   key="userRegistration.lastName" />
      </th>

      <th>
       <bean:message
                    key="userRegistration.email" />
      </th>
    </tr>
    <logic:iterate id="user" name="users">
      <tr>

         <td>
            <bean:write name="user"
```

```
                                 property="firstName"/>
                </td>

                <td>
                   <bean:write name="user"
                                    property="lastName"/>
                </td>

                <td>
                   <bean:write name="user"
                                    property="email"/>
                </td>

             </tr>

           </logic:iterate>
         </table>

       </logic:present>
         </body>
       </html>
```


User Registration List - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/strutstutorial/displayAllUsers.do
Go Links
Search Web PageRank
Done
Local intranet

User Registration List

First Name	Last Name	Email
James	Good	JamesG@arc-mind.com
Kevin	Bed	Kevin@arc-mind.com
Brian	Keet	brian@arc-mind.com
Cedric	Dun	cedric@arc-mind.com
Chuck	Cave	chuck@arc-mind.com
David	Winter	david@arc-mind.com
George	Fran	george@arc-mind.com
James	Turn	james@arc-mind.com
Rick	High	rick@arc-mind.com
Sue	Spiel	sue@arc-mind.com
Ted	Hus	ted@arc-mind.com

Figure 1.7 User Listing

11. Create a new entry in the struts-config.xml file for this new Action.

```
<action path="/displayAllUsers"
        type="strutsTutorial.DisplayAllUsersAction">

        <forward name="success"
                 path="/userRegistrationList.jsp"/>
</action>
```

Now you can deploy and test this new Action by going to: http://localhost:8080/strutstutorial/displayAllUsers.do

Adjust your domain and port number accordingly.

Summary

Now that you have completed this chapter, you have experienced many different aspects of Struts. While not going into great detail on any one topic, you should have a better idea about the breadth of Struts. The chapters that follow will expand in detail on the breadth and depth of the Struts framework.

Chapter 2

Testing Struts

Test Driven Development with Struts

Test Driven Development (TDD) is all the rage. Unlike most development fads, this one has staying power. TDD began in the Agile Development community (http://www.agilealliance.com) and Extreme Programming (XP). Although originally espoused by XP, the mainstream uses TDD, which requires fast running, automated tests. The purpose of TDD is to draw out what functionality is really needed, rather than what the programmer thinks is needed. Developing software in a J2EE environment can be fairly tough with its somewhat complex deployment requirements and dependencies on the J2EE container. Add to that the Struts framework and its complexities, and fast automated testing becomes essential. Why essential you ask? It is essential because of speed! By the time you can deploy and smoke test a feature or bug fix, you could do it four times using TDD.

This chapter covers automated testing and Struts. We will first lay the groundwork with JUnit, then cover StrutsTestCase and jWebUnit. StrutsTestCase is a testing framework that is specific to Struts. jWebUnit is a testing framework that tests web applications through the HTTP interface. StrutsTestCase and jWebUnit both build on top of JUnit. StrutsTestCase allows you to easily test Actions and other code that depends on Struts and Servlets. JWebUnit allows you to test JSPs.

Testing Model Code with JUnit

JUnit is a lightweight testing framework written by developers for developers. It is easily extensible and easy to get started with as well. With JUnit, you can write tests that you can execute. In each test, you exercise the API of your code, and you assert the expected results. The tests are written in Java.

You can use JUnit without doing TDD; however, you should consider doing TDD. If you are going to do TDD then you write your test code before you write your code. This may seem counter intuitive at first. It certainly takes a while to get use to, but it is well worth the effort as it helps you draw out the functionality of your code. Instead of creating code in a vacuum, TDD allows you to create code as it is going to be used. No more guessing what the code needs, and no more adding code that no one uses.

Note: We don't give TDD the justice it deserves in this chapter. If you are new to the concept of TDD and want to learn more, then read Kent Beck's book entitled *Test Driven Development: By Example*.

Warning! "I don't have enough time to write test code" is a self-fulfilling prophecy. The simple fact is that using JUnit and doing TDD saves time. JUnit is easy to use. You end up writing tests anyway using well-placed print statements (System.out.println calls) or adding a main method to test the API. Avoid scroll blindness by using JUnit. Scroll blindness is when you wade through the log files or console looking for you println test. The main difference is between using JUnit and what you are doing now is that you will save the test for the future. Do you want to spend your time chasing bugs or improving the quality of your code base?

How I Became Test Infected: I was never a big proponent of automated testing per se until I was forced to do it. I was working on a factory automation project in 1996. We needed 100% go code coverage and 90% exception handling coverage. I found that creating scaffolding code and automated tests to achieve this code coverage led to a cleaner code base (i.e., tighter and well-designed). If you have to test something, you don't want to test it twice, so you avoid duplication in your code base at all costs. If two things are similar, you try to abstract as much as you can to a common base class so you don't have to test twice. Later, I started using JUnit to write tests. Even later, I started doing test-driven development. I find that a test-driven development helps you to focus your efforts and design and build better code. Being a natural skeptic, it took me a while to join the TDD crowd. But now I find it quite liberating writing tests before I write the code.

Getting Started with JUnit

In order to get started with JUnit, write a simple test that tests a hash map. But before you do that you need to download JUnit and put it on your class path. You can find JUnit at http://www.junit.org. Just download the junit.jar file and put it on your classpath. Click the link that says download at http://www.junit.org. The download comes with a zip file. The zip file has the junit.jar file in it.

Using JUnit Step-by-Step

Now let's create a simple example that tests a HashMap. The objective is to show you how JUnit works, not to demonstrate TDD as the HashMap already exists. To get started with JUnit, you need to do the following steps:

1. You need to subclass `junit.framework.TestCase` as follows:

   ```
   public class HashMapTest extends TestCase{
   ```

   The TestCase class defines a fixture to run multiple tests. The setUp method gives you a chance to set up the objects that are part of the fixture under test.

2. Override the `setUp` method to setup objects you are about to test as follows:

   ```
   public class HashMapTest extends TestCase{

   private Map map;

   /** Create a HashMap to test.
    */
   protected void setUp() throws Exception {
   map = new HashMap();
   map.put("Larry", "Oracle");
   map.put("Bill", "Microsoft");
   }
         …
   ```

   The above sets up a `HashMap` and populates the map with two method calls to the put method of the map. Now that we have the objects we want to test, we can start testing.

3. Define one or more `testXXX` methods, where XXX is the name of the test as follows:

```
public void testRemove() throws Exception{
map.remove("Bill");
assertEquals(1, map.size());
}

public void testCount() throws Exception{
assertEquals(2,map.size());
}

public void testAdd()throws Exception{
map.put("Rick", "ArcMind");
assertEquals(3, map.size());
}

public void testKeyIndex()throws Exception{
assertEquals(map.get("Larry"), "Oracle");
assertNull(map.get("Rick"));
}

protected void tearDown() throws Exception {
}

     ...
```

You test the map by invoking methods on its public interface and then asserting the desired result. There are many assert methods in the JUnit framework (e.g., `assertTrue`, `assertFalse`, `assertSame`, `assertNull` and more).

Warning! The setUp method gets called before each test. The tearDown method gets called after each test. Notice that the testCount is after the testRemove method. If the setUp method did not get called before each test, then the testCount method would fail as the testRemove method removed one of the original two items that was added to the map in the setUp method.

4. Optional Release any resources by overriding the `tearDown` method as follows:

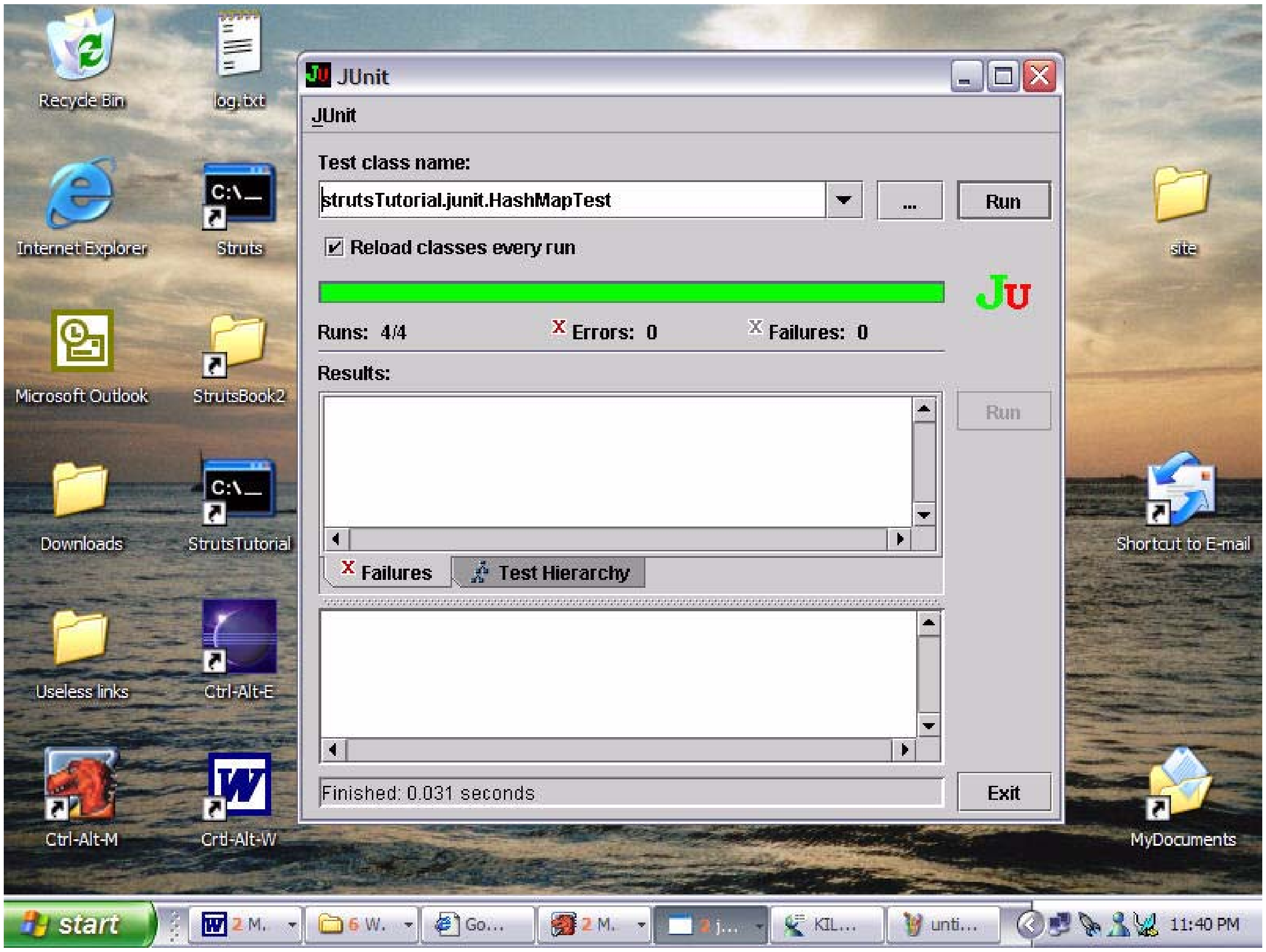
```
protected void tearDown() throws Exception {}
```

Implementing the `tearDown` method is optional. You only need to do this if you need to clean up resources like closing a connection to a database. Thus, you typically do not need to override the `tearDown` method.

5. Run the tests. Since JUnit is fairly prevalent, most IDEs (Jbuilder, Eclispe, IBM WSAD, NetBeans, IntelliJ and more) have support for JUnit. Thus, it is a simple matter of running the test case through the IDE. However for you emacs, vi, and notepad users, here is how you can run the test we just created. Add a main method to your `HashMapTest` as follows:

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

The TestRunner is defined in `junit.swingui.TestRunner`. Now you can run the test like any other Java class. The following GUI will appear (see figure 2.1):



Test class name: strutsTutorial.junit.HashMapTest

☑ Reload classes every run

Runs: 4/4 Errors: 0 Failures: 0

Results:

Finished: 0.031 seconds

Figure 2.1 Running HashMapTest

As you write tests for awhile, you want the ability to run a whole grouping of tests together. In fact, running all of the tests in your project several times a day is called continuous integration. This allows you to find problems early in the development cycle while they are easy to fix.

Tip: Most tests run very quickly, but some tests run slower. Group the tests that run faster together into suites that you can run more often. For example, a test that talks to a database may run slow, while a test that implements a business logic rule may run quickly.

6. Group related test in a test suite as follows:

```
package strutsTutorial.junit;

import junit.framework.Test;
import junit.framework.TestSuite;

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

public static void main(String[] args) {
junit.swingui.TestRunner.run(AllTests.class);
}

public static Test suite() {
TestSuite suite = new TestSuite("Sample Test ");
suite.addTest(new TestSuite(HashMapTest.class));
suite.addTest(new TestSuite(VectorTest.class));
return suite;
}
}
```

The suite method groups all of the tests in `HashMapTest` and all of the tests in `VectorTest` into one test suite. The main method is used to run this new test suite.

This concludes our coverage of JUnit. You can see how to easy it is to use JUnit to write tests. For the sake of being thorough, here is the complete listing of the `HashMapTest`:

```
/*
 *  This file was created by Rick Hightower of ArcMinds Inc.
 *
 */
package strutsTutorial.junit;

import java.util.HashMap;
import java.util.Map;

import junit.framework.TestCase;
import junit.swingui.TestRunner;


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

private Map map;

/** Create a HashMap to test.
 */
protected void setUp() throws Exception {
map = new HashMap();
map.put("Larry", "Oracle");
map.put("Bill", "Microsoft");
}

public void testRemove() throws Exception{
map.remove("Bill");
assertEquals(1, map.size());
}

public void testCount() throws Exception{
assertEquals(2,map.size());
}

public void testAdd()throws Exception{
map.put("Rick", "ArcMind");
assertEquals(3, map.size());
}

public void testKeyIndex()throws Exception{
assertEquals(map.get("Larry"), "Oracle");
assertNull(map.get("Rick"));
}

protected void tearDown() throws Exception {
}

public static void main (String [] args){
TestRunner.run(HashMapTest.class);
}

}
```

Now that we have covered the basics of JUnit, it is time to see it applied to our application.

Tip: You can also group tests together using Ant's batchtest subtask under the junit task. I usually use both. I use batchtest to generate reports, and I use test suites to quickly run related tests in my IDE.

Applying JUnit to Our Struts Tutorial

When we last left our Struts application from Chapter 1, we had no testing at all. We were able add users to the database and get a list of users from the database. However, if you recall, we had our database access code mixed right into the action. This is a bad practice; actions are supposed to provide control flow only and select the next view. Actions are not supposed to have JDBC code inside of them, as they are merely the glue between the view and the model.

In order to separate the Action code from the JDBC code, we will use the DAO pattern. DAO stands for database access object. The DAO pattern is part of Sun's blueprint for Java and J2EE. (See http://java.sun.com/blueprints/patterns/catalog.html and http://java.sun.com/blueprints/corej2eepatterns/ for more detail.)

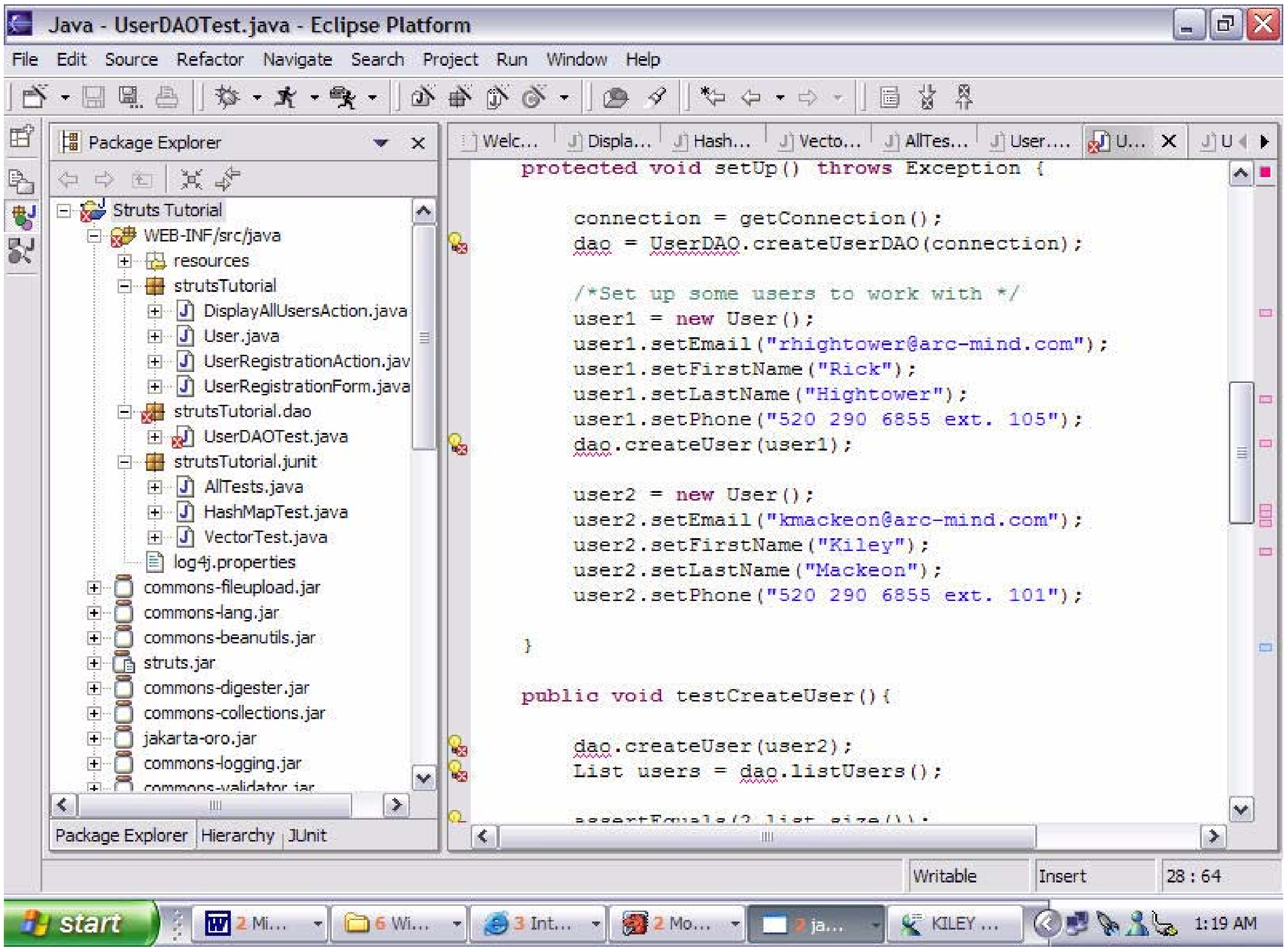
We want to write a test, against a DAO object that does not exist, to add a user to the application as follows:

```
[UserDAOTest.java]
public static final String EMAIL="rhightower@arc-mind.com";
protected void setUp() throws Exception {

connection = getConnection();
dao = DAOFactory.createUserDAO(connection);

/*Set up some users to work with */
user1 = new User();
user1.setEmail(EMAIL);
user1.setFirstName("Rick");
user1.setLastName("Hightower");
user1.setPhone("520 290 6855 ext. 105");
dao.createUser(user1);

user2 = new User();
user2.setEmail("kmackeon@arc-mind.com");
user2.setFirstName("Kiley");
user2.setLastName("Mackeon");
user2.setPhone("520 290 6855 ext. 101");

}

public void testCreateUser(){

dao.createUser(user2);
List users = dao.listUsers();

assertEquals(2,users.size());

boolean found = false;
for (Iterator iter = users.iterator(); iter.hasNext();) {
User user = (User) iter.next();
if (user.getEmail().equals(EMAIL)){
found = true;
                        break;
}
}

assertTrue(found);
}
```

At this point, we are calling methods that do not exist yet. In fact, the `UserDAO`, does not exist yet. This is a common technique in TDD. You define the test before you define the code. Now since this will not compile, we cannot run the test yet. The first step then is to get this to compile.

Tip: Many IDEs allow you to stub out methods that do not exist yet. You write your test code and then when the compile complains, you tell the browser to implement the missing method. The IDE will implement a stubbed out version, i.e., an empty method. See figure 2.2.

Tip: You can use DBUnit to populate your database before running tests instead of using the technique above. We will cover using DBUnit, when we cover AppFuse in Chapter 14.



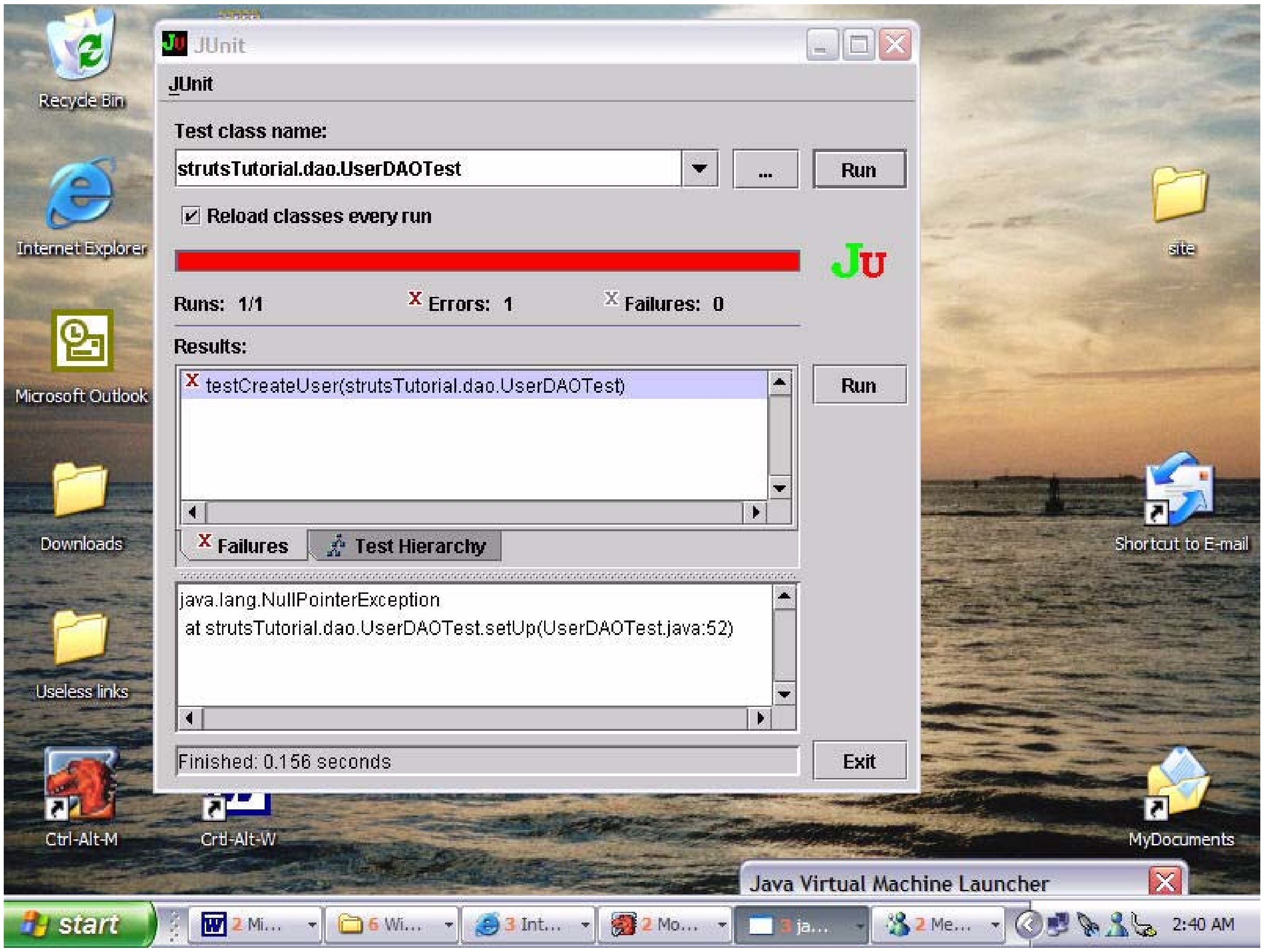
```
    protected void setUp() throws Exception {

        connection = getConnection();
        dao = UserDAO.createUserDAO(connection);

        /*Set up some users to work with */
        user1 = new User();
        user1.setEmail("rhightower@arc-mind.com");
        user1.setFirstName("Rick");
        user1.setLastName("Hightower");
        user1.setPhone("520 290 6855 ext. 105");
        dao.createUser(user1);

        user2 = new User();
        user2.setEmail("kmackeon@arc-mind.com");
        user2.setFirstName("Kiley");
        user2.setLastName("Mackeon");
        user2.setPhone("520 290 6855 ext. 101");

    }

    public void testCreateUser(){

        dao.createUser(user2);
        List users = dao.listUsers();
```

Figure 2.2 Test Fails because code does not compile yet

Add the stub out methods to get the above to compile as follows:

```
[DAOFactory.java]

public class DAOFactory {
/**
 * Factory method to create a user DAO.
 * @param connection
 * @return
 */
public static UserDAO createUserDAO(Connection connection){
             return null;
      }

}
[UserDAO.java]

public interface UserDAO {


/**
 * Create a User.
 * @param user
 */
public void createUser(User user);

/**
 * List all users in the system.
 * @return
 */
public List listUsers() ;

}
```

Now rerun the test. You should get the red bar. See figure 2.3.



```
strutsTutorial.dao.UserDAOTest
Runs: 1/1    Errors: 1    Failures: 0
testCreateUser(strutsTutorial.dao.UserDAOTest)
java.lang.NullPointerException
 at strutsTutorial.dao.UserDAOTest.setUp(UserDAOTest.java:52)
Finished: 0.156 seconds
```

Figure 2.3 Test Fails because stubbed our methods not implemented yet

The objective of TDD is to write the test first. One strategy in TDD is to let the IDE define the stubbed out methods. By writing the test first, you only add the methods you actually need for the object under test (less is more). The IDE stubbing out the methods for you is just a bonus.

Note: *TDD maxim:* Code that does not have a test does not exist!

Now that we have the test and the stubbed out code, it is time to write the code. Here is the new implementation of the DAOUser as follows:

```
[SQL92UserDAO.java]

public class SQL92UserDAO implements UserDAO {

private Connection connection;

private static Log log = LogFactory.getLog(SQL92UserDAO.class);

public SQL92UserDAO(Connection connection) {
this.connection = connection;
}

public void createUser(User user) {
log.trace("In createUser method");

PreparedStatement statement;
try {
statement =
connection.prepareStatement(
"insert into USER " +
       "(EMAIL, FIRST_NAME, LAST_NAME, PASSWORD, PHONE, FAX)"
+ " values (?,?,?,?,?,?)");
statement.setString(1, user.getEmail());
statement.setString(2, user.getFirstName());
statement.setString(3, user.getLastName());
statement.setString(4, user.getPassword());
statement.setString(5, user.getPhone());
statement.setString(6, user.getFax());
statement.executeUpdate();

} catch (SQLException e) {
log.error("Unable to create User", e);
throw new NestableRuntimeException(e);
}

}

public List listUsers() {
log.trace("In listUsers method");
List list = new ArrayList(50);
try {
Statement statement = connection.createStatement();

ResultSet rs =
statement.executeQuery(
"select FIRST_NAME, LAST_NAME, EMAIL from USER");
while (rs.next()) {
String firstName = rs.getString(1);
String lastName = rs.getString(2);
String email = rs.getString(3);

if (log.isDebugEnabled()) {
log.debug(
"just read firstName="
+ firstName
+ ", lastName="
+ lastName
+ ", email="
+ email);
}

User user = new User();
user.setEmail(email);
user.setFirstName(firstName);
user.setLastName(lastName);
list.add(user);
}
} catch (Exception e) {
log.error("Unable to list Users", e);
throw new NestableRuntimeException(e);
}
return list;

}

}
```

The SQL92UserDAO is for working with databases that are fairly SQL compliant. If we need to branch to support other databases, then we may add, for example, OracleUserDAO or SQLServerUserDAO, which would likely subclass SQL92UserDAO and override the methods that change for that particular vendor. Here is the newly updated DAOFactory that uses SQL92UserDAO as follows:

```
[DAOFactory]
public class DAOFactory {
/**
 * Factory method to create a user DAO.
 * @param connection
 * @return
 */
public static UserDAO createUserDAO(Connection connection){
return new SQL92UserDAO(connection);
}

}
```

Notice that we have not implemented all of the CRUD (create, read, update, and delete) features of `UserDAO`. With TDD, you only implement the features when you are ready to use them. That way if later you decide that you do not need a feature, then it was never added to the project.

Now that we have our new DAO class that creates and lists users, you need to change the action that have JDBC call to use these new DAO classes. Of course if we are following the TDD paradigm, then we cannot modify the code until we have a test in place for it. In the next section, you use Struts Test Case to test the actions that you wrote during the first chapter. Then, you modify the Actions to use the new DAO object.

Testing Struts Actions with StrutsTestCase

StrutsTestCase is a JUnit extension that allows you to test code that relies on the Struts framework. StrutsTestCase works in two modes: in container testing and simulated container testing. You can run StrutsTestCase test inside of a running J2EE container. This support is provided by StrutsTestCase's integration with Cactus. StrutsTestCase extends Cactus to provide in container testing of Struts Actions. Cactus is a framework for in container unit testing. Cactus extends JUnit as well. StrutsTestCase also allows you to use a simulated container test via its Mock Object approach. You can find more information about StrutsTestCase at http://strutstestcase.sourceforge.net/. You can find more information about Cactus at http://jakarta.apache.org/cactus/.

Using either mode, StrutsTestCase uses the Struts ActionServlet controller to test your Action objects, mappings, form beans and forward declaration. StrutsTestCase provides special assertion methods to assert, for example, that an action returned a certain forward.

You typically combine StrutsTestCase with Cactus when you have code that depends on J2EE container resources. For example, assume that you have an Action object that depends on local EJB Entity bean. Suffice to say, it is a good idea *whenever possible* not to rely too heavily on container resources, as your code gets tightly coupled with the J2EE environment and harder to test. The problem with Cactus based StrutsTestCase tests is that Cactus is hard to set up (but you only have to do it once), and the tests take longer to run than the Mock Object equivalent test.

On the flip side, container simulation—Mock Object based tests require that your action code is not tightly coupled with the underlying J2EE container. This requires a cleaner design and cleaner delineation between your model and the container. Typically, you can get a fairly clean design by using an Abstract Factory from your Actions. Thus, your Actions never directly deal with J2EE; they talk to the Abstract Factory to get model objects. The Abstract Factory may in turn talk to the J2EE services, like JNDI, to look up and interact with EJBs. Then, for testing, you create an Abstract Factory that delivers Mock Objects for testing. This is the approach I usebecause the advantage of using container simulation is you don't have to deploy your code to the container to test it. In short, the reason to use container simulation is speed. Tests that take a long time to run don't get run as often. TDD favors fast running tests.

Using StrutsTestCase (Mock Mode) Step-by-Step

Let's get started with StrutsTestCase. To download StrutsTestCase, go to http://sourceforge.net/project/showfiles.php?group_id=39190. The examples in this book are based on the 2.0 release of StrutsTestCase. You need to download the file strutstest200-1.1_2.3.zip. The 1.1 portion of the file name refers to Struts version 1.1. The 2.3 portion of the file name refers to the fact that StrutsTestCase simulates the Servlet container at the 2.3 version of the specification. Extract the ZIP file and add the strutstest-2.0.0.jar file to the classpath of your IDE for the strutsTutorial project.

Now, let's create an example that tests the `UserRegistrationForm` action. Copy the junit jar file to ant/lib. To get started with StrutsTestCase you need to do the following steps:

1. Subclass `MockStrutsTestCase`.

   ```
   public class UserRegistrationActionTest extends
                                              MockStrutsTestCase {

         private Connection connection;

   public UserRegistrationActionTest(String testName) {
   super(testName);
   }
   ```

   Notice we subclass `MockStrutsTestCase` and create a constructor that calls its superclass constructor.

2. Override the `setUp` method to set up objects you are about to test (be sure to call the superclass version of `setUp`).

   In the `setUp` method, we need to put a `DataSource` into scope underneath the key for the datasource ("userDB"). This is where the action gets its datasource. Now, we could just allow Struts to create a datasource for us since we are executing the action through the `ActionServlet`, and we already configured a datasource. But doing it this way allows us to pass the in-memory datasource, which goes away when we are done testing (nice for clean up). If you leave this out, your tests will only run once. (Once it runs, the primary key, e-mail, is already taken for that user.) Here is the `setUp` method:

```
public void setUp() throws Exception {

/* Always call the superclass setUp method! */
super.setUp();

/* Get the Sevlet Context from the ActionServlet */
ServletContext context =
                      getActionServlet().getServletContext();

/* Grab the moduleConfig from the ServletContext */
ModuleConfig moduleConfig =
               RequestUtils
               .getModuleConfig(this.getRequest(),context);

/* Create a Mock DataSource */
DataSource source =
                JDBCTestUtils.getMockedDataSource();

/* Get the connection from the DataSource */
connection = source.getConnection();

/* Create the database table for User */
JDBCTestUtils.createDB(connection);

/* Put the datasource where Struts will find it
               and deliver it to the Action */
context.setAttribute("userDB" +
                 moduleConfig.getPrefix(),source);

}
```

Notice that the `MockStrutsTestCase` has a reference to the `ActionServlet` and a mock request object that you can access through the `getActionServlet` and `getRequest` method. The `JDBCTestUtils.getMockedDataSource()` method is a utility method I wrote that creates a mock datasource (`javax.sql.DataSource`) that returns a connection to the in-memory database. The good thing about this datasource is that the data will go away when we are done testing.

3. Define one or more testXXX methods where XXX is the name of the test. Now, we need to define our test method as follows:

```
public void testUserRegistration() {
/* Primary key */
String email = "TEST1rhightower@arc-mind.com";

/* Set the action path */
setRequestPathInfo("/userRegistration");

populateUserForm(email);

/* Run the action through the ActionServlet */
actionPerform();

/* Make sure that the user was created in the
              system*/
assertTrue(JDBCTestUtils.userExists(connection,
                                                     email));

/* Verify that the action forwarded to success */
verifyForward("success");

/* Verify that there were no exceptions */
verifyNoActionErrors();
}

private void populateUserForm(String email) {
addRequestParameter("userName", "KingAdRock");
addRequestParameter("firstName", "Rick");
addRequestParameter("lastName", "Hightower");
addRequestParameter("email", email);
addRequestParameter("lastName", "Hightower");
addRequestParameter("phone", "555-1212");
addRequestParameter("fax", "555-1212");
addRequestParameter("password", "555-1212");
addRequestParameter("passwordCheck", "555-1212");
}
```

The above test accesses the action located at path `"/userRegistration"`. The test populates form parameters (request parameters) with the `addRequestParameter` method via the `populateUser-Form` helper method. It then runs the action with the actionPerform method. The test verifies the action forwarded to "success" and that there were no ActionErrors with the `verifyNoActionEr-rors` method. The call to `JDBCTestUtils.userExists(connection, email)` verifies that the user was created in the system. Creating a utility class like `JDBCTestUtils` is a good practice—much better than repeating the code to perform the tests in every test in every layer.

Tip: Refactor your test code as you would with any code. Create utility classes to make testing easier.

The code above only tests the “happy-go-lucky” route. What if the user already exists in the system? What if the user hits the Cancel button? With TDD, you test anything that could possibly break. Here is the test case to see if a user already exists:

```
public void testDuplicateUserRegistrations(){

/* Create the first registration */
String email = "rhightower@arc-mind.com";
JDBCTestUtils.createUser(connection, email);

/* Set the action path */
setRequestPathInfo("/userRegistration");
populateUserForm(email);

/* Run the action through the ActionServlet */
actionPerform();

/* Make sure that the Exception Handler error
              message is present */
verifyActionErrors(new
                 String[]{"userRegistration.sql.exception"});

/* Make sure the Exception object got put into
              request scope */
Exception e = (Exception) getRequest()
                .getAttribute(Globals.EXCEPTION_KEY);
assertNotNull(e);

}
```

The code above ensures that the Action forwards to the exception handler if there are duplicate users in the system. Notice the use of `verifyActionErrors` to see if the action handler's error message is present. You may recall that we registered an exception handler with this action as follows:

```
<action path="/userRegistration"
  type="strutsTutorial.UserRegistrationAction"
  name="userRegistrationForm" attribute="user"
  input="/userRegistration.jsp">

      <exception type="java.sql.SQLException"
  key="userRegistration.sql.exception"
  path="/userRegistrationException.jsp" />

     <forward name="success"
  path="/regSuccess.jsp" />
                <forward name="failure"
  path="/regFailure.jsp" />
</action>
```

Now, notice that we are using the key (`"userRegistration.sql.exception"`) from the exception handler in the String array that you pass to `verfyActionErrors`.

The following code tests to see if the user hit the Cancel button in the middle of a user registration:

```
public void testCancelUserRegistration(){

/* Primary key */
String email = "TEST1rhightower@arc-mind.com";

/* Set the action path */
setRequestPathInfo("/userRegistration");


populateUserForm(email);

/* Simulate hitting the cancel key */
getRequest()
.setAttribute(Globals.CANCEL_KEY,"CANCEL");

/* Run the action through the ActionServlet */
actionPerform();

/* Make sure the user was not created */
assertFalse(
JDBCTestUtils.userExists(connection, email));

/* Verify we were forwarded to the right place */
verifyForward("welcome");


verifyNoActionErrors();
}
```

The general rule is that you should test anything that could possibly break. Can you think of anything that could possibly break? If you can, write another test for it using the techniques above.

4. Release any resources by overriding the `tearDown` method (be sure to call the superclass version of `tearDown`). In the `tearDown` method, we can remove any connection that we create.

```
public void tearDown() throws Exception {
/* Always call the super class method*/
super.tearDown();

/* Clear users is needed for non-in-memory
              databases */
JDBCTestUtils.clearUsers(connection);

/* Close the database connection */
connection.close();
}
```

   Since we used an in-memory database (HyperSonic SQL), we don't have to release any resources per se, but we do for good measure.

5. Run the tests. If you want to run the test from your IDE, you need to add the /strutsTutorial directory to your classpath (the root directory of the project where the jsp files are located). We do this because Struts will try to load the relative path /WEB-INF/web.xml and /WEB-INF/struts-config.xml files from the Servlet resource area, and StrutsTestCase simulates the Server resource area with as a regular class resource—a bit of needed subterfuge to get things moving along.

   If you want to run and compile your test from Ant, you need to make the following changes to the compile.classpath:

```
<!-- classpath for Struts 1.1 -->
<path id="compile.classpath">

    <!--Added this for StrutsTestCase -->
    <pathelement path =".."/>

    <pathelement path ="classes"/>
    <pathelement path ="${classpath}"/>
    <pathelement path ="${servlet.jar}"/>

    <!-- Added these for StrutsTestCase -->
    <pathelement path ="/tools/junit3.8.1/junit.jar"/>
<pathelement path ="/tools/strutstest/strutstest-2.0.0.jar"/>
<pathelement path ="/tools/hsqldb/lib/hsqldb.jar"/>
<pathelement path ="/tomcat5/common/lib/jsp-api.jar"/> //


    <!-- Added this to simplfy classpath creation-->
    <fileset dir="lib">
    <include name="*.jar"/>
    </fileset>
```

```
</path>
```

Notice the comments in the Ant script state these jar files are needed for StrutsTestCase to run. This means that the StrutsTestCase framework has dependencies on these jar files.

Just make sure yours matches the code above, and everything should be good.

Next, you will need to add a test target to run the test as follows:

```
<target name="test">
<junit  >
<classpath refid="compile.classpath"/>
<formatter type="plain" usefile="no"/>
<batchtest >
<fileset dir="./src/java">
<include name="strutsTutorial/UserRegistrationActionTest.java"/>
</fileset>
</batchtest>
</junit>
</target>
```

Run the compile and test target. If the test fails, you will get a message saying that the test failed. that the message will look something like this:

```
c:\strutsTutorial\WEB-INF\src>ant compile test
…
    [junit] TEST strutsTutorial.UserRegistrationActionTest FAILED

BUILD SUCCESSFUL
Total time: 3 seconds
```

Notice a few things. First, notice that the build succeeded but the test failed. Don’t get caught thinking your test passed just because your build passed. One does not mean the other.

If the tests pass, the output will look like this:

```
c:\strutsTutorial\WEB-INF\src>ant compile test
…
    [junit] Testcase: testUserRegistration took 0.906 sec
    [junit] Testcase: testDuplicateUserRegistrations took 0.219 sec
    [junit] Testcase: testCancelUserRegistration took 0.156 sec


BUILD SUCCESSFUL
Total time: 3 seconds
```

6. Now that we have written our tests and they running and passing, we can refactor our Actions to use our model objects (`UserDAO`, `User`, etc.), and get rid of that ugly JDBC code in our Action. And we can sleep sound at night knowing that our test case still runs after we refactor the code. Having automated test cases takes the fear out of refactor because you no longer have to worry if your refactoring broke any working code.

   Now, it is time to refactor. Compare the following to what you created for the tutorial in chapter 1:

```
public class UserRegistrationAction extends Action {

private static Log log =
          LogFactory.getLog(UserRegistrationAction.class);

public ActionForward execute(
ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response)
throws Exception {
log.trace("UserRegistrationAction.execute()");

if (isCancelled(request)){
log.debug("Cancel Button was pushed!");
return mapping.findForward("welcome");
}
UserRegistrationForm userForm =
                               (UserRegistrationForm) form;

log.debug("userForm email "
                               + userForm.getEmail());

DataSource dataSource =
                            getDataSource(request, "userDB");
Connection conn = dataSource.getConnection();

/* Create UserDAO */
UserDAO dao = DAOFactory.createUserDAO(conn);

/* Create a User DTO and copy the properties
                   from the userForm */
User user = new User();
BeanUtils.copyProperties(user, userForm);

/* Use the UserDAO to insert the new user into
                   the system */
dao.createUser(user);


return mapping.findForward("success");
}

}
```

Now we rerun the test and everything works fine, right? Wrong! Bingo, Banjo, Blamo! One of our test cases fail. The `testUserRegistration` and `testCancelUserRegistration` tests pass, and `testDuplicateUserRegistrations` fails. The test fails because our DAO objects throw a wrapper exception instead of the original `SQLException`. The handler we configured was set up to work with `SQLException`. Would you have caught that without this test? Can you see the benefits of testing?

Note: The code now uses BeanUtils(import org.apache.commons.beanutils.BeanUtils) to copy the properties from the userForm ActionForm to User DTO (Data Transfer object). BeanUtils ships with Struts.

To get all the tests to run, you need to change the exception handler as follows:

```
<action path="/userRegistration"
 type="strutsTutorial.UserRegistrationAction"
 name="userRegistrationForm" attribute="user"
 input="/userRegistration.jsp">

              <exception type="org.apache.commons.lang.exception.NestableRunt-
imeException"
key="userRegistration.sql.exception"
path="/userRegistrationException.jsp" />

   <forward name="success"
path="/regSuccess.jsp" />
              <forward name="failure"
path="/regFailure.jsp" />
</action>
```

The key lesson here is that using StrutsTestCase allows us to test our code in an integrated manner.

Some Design Notes: In the above example, we talk directly to our DAO object in our Action. We removed all of the JDBC code out of the Action. You can take this a step further by creating a Service Layer. The Service Layers act as a facade separating the controller tier, which is dependent on Struts and the Servlet API, and your true domain model. For more information on designing and implementing an industrial strength application, please read *Patterns of Enterprise Application Architecture* by Martin Fowler. Also, look into Chapter 14 of this series which covers AppFuse, which implements many of the patterns in Fowler's book.

Using StrutsTestCase (Cactus Mode)

In order to test our tests inside of the container, we would have to subclass CactusStrutsTestCase instead of MockStrutsTestCase and then follow the directions for setting up Cactus located at http://jakarta.apache.org/cactus/integration/howto_config.html. The rest of the test code should be identical. You would use Cactus if, for example, your Action talked directly to an EJB or some other object or service provided by the J2EE container.

Note: To learn more about Cactus, JUnit and Ant read the book *Java Tools for Extreme Programming* by Richard Hightower et al.

We have tested our Model code (DAO objects). We tested our controller code (Action objects). But, we have not tested our JSPs. JSP testing is often overlooked because it is thought to be relatively hard to accomplish.

Testing JSP with jWebUnit

jWebUnit is another testing framework built on top of JUnit. It also uses HttpUnit, which is a framework for parsing HTML easily.

jWebUnit makes testing JSP easy by providing a high-level API for navigating web applications. Like StrutsTestCase, jWebUnit provides methods to verify and assert valid conditions. JWebUnit assertion methods relate to navigation and the HTML DOM including form entry and submission, validation of table contents and many common scenarios.

Note: The book *Java Tools for Extreme Programming* by Richard Hightower et al covered using JUnit and HttpUnit together to tests Servlets, JSP, Custom Tags and Filters. The chapter on HttpUnit is good background information for understanding jWebUnit. Using jWebUnit is much easier than using JUnit and HttpUnit together.

Using jWebUnit Step-by-Step

You can find jWebUnit at http://jwebunit.sourceforge.net/. You can download jWebUnit from http://sourceforge.net/project/showfiles.php?group_id=61302. Download the file jWebUnit-1.1.1.zip. Extract the file and put the jwebunit.jar on the classpath of your IDE project.

The jWebUnit framework depends on the HttpUnit framework, so you will need to download HttpUnit. The HttpUnit web site can be found at http://httpunit.sourceforge.net/.

To download HttpUnit go to http://prdownloads.sourceforge.net/httpunit/httpunit-1.5.4.zip?download. Extract the ZIP file and put the following jar files on your project's classpath js.jar, junit.jar, nekohtml.jar, Tidy.jar, xercesImpl.jar, xmlParserAPIs.jar (found under jars) and of course httpunit.jar (found under lib).

Now let's create an example that tests the user userRegistration.jsp JSP.

Before we get started, add an action mapping to userRegistration.jsp. Generally speaking, it is bad form to link directly to a JSP file. So instead of linking directly to the JSP file, we will add the following entry in our struts-config.xml file.

```
<action path="/userRegForm" forward="/userRegistration.jsp" />
```

This will allow us to link to /userRegForm.do instead of directly to the JSP. This is a good MVC practice that will allow us to add an Action that forwards to this JSP in the future without breaking all of the links.

To get started with jWebUnit, you need to complete the following steps:

1. Subclass net.sourceforge.jwebunit.WebTestCase.

   ```
   public class UserRegistrationJSPTest extends WebTestCase {

   public UserRegistrationJSPTest(String name) {
   super(name);
   }
   ```

   Not much new here.

2. Override the setUp method to set up objects you are about to test.

   ```
   public void setUp() {
   getTestContext().setBaseUrl("http://localhost:8080/strutsTutorial");
   }
   ```

   Here, we set the base URL of our web application.

3. Define one or more testXXX methods where XXX is the name of the test.

   ```
   public void testFormSubmit() {
   beginAt("/userRegForm.do");
   populateUserForm("5999rhightower@arc-mind.com");
   submit();
   assertTitleEquals("User Registration Was Successful!");
   }

   public void testFormSubmitDuplicates() {
      beginAt("/userRegForm.do");
      populateUserForm("5666rhightower@arc-mind.com");
      submit();

      beginAt("/userRegForm.do");
      populateUserForm("5666rhightower@arc-mind.com");
      submit();

      assertTitleEquals("User Registration Had Some Problems");
   }


   private void populateUserForm(String email) {
   setFormElement("userName", "KingAdRock");
   setFormElement("firstName", "Rick");
   setFormElement("lastName", "Hightower");
   setFormElement("email", email);
   setFormElement("lastName", "Hightower");
   ```

```
setFormElement("phone", "555-1212");
setFormElement("fax", "555-1212");
setFormElement("password", "555-1212");
setFormElement("passwordCheck", "555-1212");
}
```

4. Before you run the tests, please remove the spaces and linefeeds from the title in regSuccess.jsp and userRegistrationException.jsp.

   Note that you will have to have jWebUnit and the HttpUnit jar files on your class path to get these tests to run. Also, don't forget to deploy your web application and start your web application server (e.g., Resin or Tomcat).

5. Now, we write our tests by navigating to the form and populating form data. As you can see, jWebUnit provides a high level API for testing our web application. The `beginAt` positions jWebUnit to a particular URL. In our case, the URL is linked to our userRegistration form. The populateUserForm is a helper method that uses the `setFormElement` method to populate the form with some default data. An interesting thing about the `setFormElement` is that it will not allow you to set a field that does not exist on the form. The Submit button submits the populated data to the web application.

   The approach above just scratches the surface on what you can do with jWebUnit. To find out about all of the assertion methods that come with `WebTestCase`, take a look at the API documents at http://jwebunit.sourceforge.net/api/.

   The unfortunate thing about this test is that we can only run it once. You would need to add a way to clean up the test data in the system so the test can be run again. This task is left up to the reader to accomplish.

Note: Another approach to testing JSP is to use Canoo. We will cover Canoo in chapter 14, when we cover using AppFuse. Canoo allows you to specify a test purely via an Ant build file. Thus, you write your test in XML, not in Java. This allows you to write your tests much faster!

Summary

This chapter covered automated testing in Struts. We started by testing our model with JUnit. Then we covered using StrutsTestCase to test the Action from our controller. Lastly, we used jWebUnit to test our JSP views. StrutsTestCase is a testing framework that is specific to Struts. jWebUnit is a testing framework that tests web applications through the HTTP interface. StrutsTestCase and jWebUnit both build on top of JUnit. StrutsTest-Case allows you to easily test Actions and other code that depends on Struts and Servlets. JWebUnit allows you to test JSPs. Testing is an important part of the development process that often gets overlooked, but this chapter should help you understand its importance and offer some effective ways to implement proper testing procedures.

Chapter 3

Working with ActionForms and DynaActionForms

ActionForms function as data transfer objects to and from HTML forms.ActionForms populate HTML forms to display data to the user and also act like an object representation of request parameters, where the request parameters attributes are mapped to the strongly typed properties of the ActionForm. ActionForms also perform field validation.

This chapter is divided into two sections. The first section covers the theory and concepts behind ActionForms. The second part covers common tasks that you will need to perform with ActionForms like:

- Creating a master detail ActionForm (e.g., Order has LineItems)
- Creating an ActionForm with nested JavaBean properties
- Creating an ActionForm with nested indexed JavaBean properties
- Creating an ActionForm with mapped backed properties
- Loading form data in an ActionForm to display
- Configuring DynaActionForms

Defining an ActionForm

An ActionForm is an object representation of an HTML form (or possibly several forms in a wizard-style interface). ActionForms are a bit of an anomaly in the MVC realm. An ActionForm is not part of the Model. An ActionForm sits between the View and Controller acting as a transfer object between the two layers. An ActionForm represents not the just the data, but the data entry form itself.

ActionForms have JavaBean properties to hold fields from the form. An ActionForm's JavaBean properties can be primitive types, indexed properties, Maps (i.e., HashMap), or other JavaBeans (nested beans). Thus, ActionForms do not have to be one-dimensional; they can consist of master detail relationships and/or can have dynamic properties. (Examples of indexed properties, dynamic properties, and master detail relationships can be found in the tutorial section of this chapter.)

ActionForms are configured to be stored by Struts in either session or request scopes. Session scope is the default scope. Struts automatically populate the ActionForm's JavaBean properties from corresponding request parameters, performing type conversion into primitive types (or primitive wrapper types) if needed. You typically use Session scope for wizard-style interfaces and shopping carts.

With ActionForms, you use JavaBean properties to represent the fields in the HTML form. You can also use JavaBean properties to represent buttons and controls; this helps when deciding which button or control the user selected.

Understanding the Life Cycle of an ActionForm

The ActionServlet handles requests for Struts (i.e., requests ending in *.do are common). The ActionServlet looks up the RequestProcessor associated with the module prefix. The RequestProcessor implements the handling of the life cycle and uses RequestUtils as a façade to Struts objects mapped into Servlet scopes (request, session, and application).

When a form gets submitted, Struts looks up the action mapping for the current request path from the ModuleConfig. The ModuleConfig is the object manifestation of the struts-config.xml file, each module gets its own ModuleConfig. (Recall that the action attribute in the html:form tag specifies the path of the action to invoke.)

Struts locates the form-bean mapping from the action mapping associated with the request path. (This occurs in the `processActionForm` method of the `RequestProcessor` by calling the `createActionForm` method of `RequestUtils.)`

If Struts does not find an ActionForm in the scope specified by the action mapping, it will create an instance of the ActionForm identified form-bean mapping.

Struts populates the ActionForm by mapping the request parameters from the HTML form variables to the JavaBean properties of the ActionForm instance. Before it populates the form, Struts calls the reset() method of the ActionForm. (This occurs in the `processPopulate` method of the `RequestProcessor` by calling the `populate` method of `RequestUtils.)`

If validation is required and is successful, an instance of the Action class will be invoked. Validation is required if the validate attribute of the action mapping is not false (the default is true). If validation fails, control will be returned to the submitting form (the input JSP) where the JSP form fields will be populated by the ActionForm. The ActionForm is valid if the validate() method returns null or an empty ActionErrors collection. (This occurs in the `processValidate` method of the `RequestProcessor.)`

The life cycle of an ActionForm is demonstrated in the following diagram.

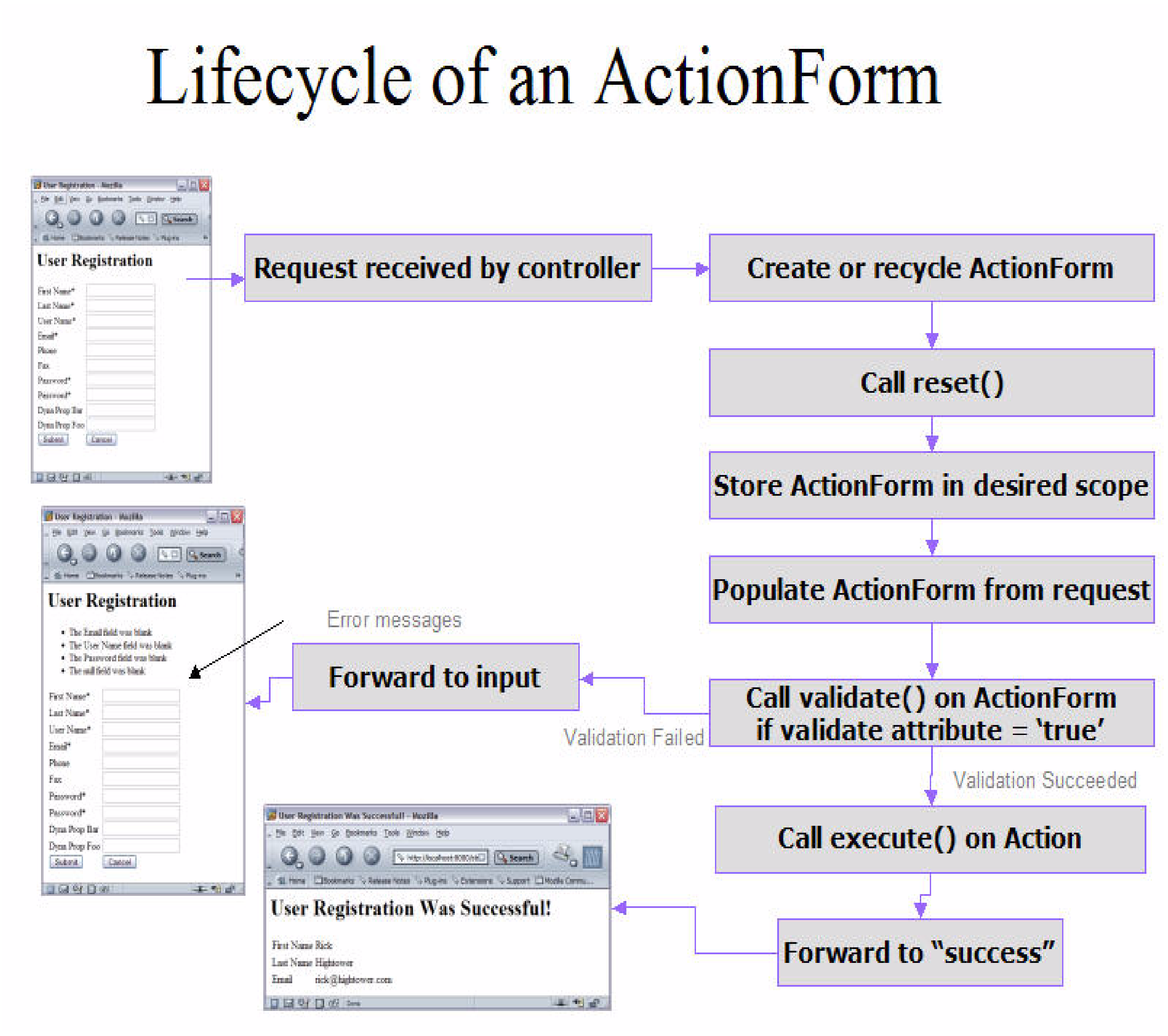


Figure 3.1 Life Cycle of an ActionForm

Understanding ActionForm's reset() Method

The reset() method allows you to set properties to default values. The ActionForm is a transfer object; therefore, you should not deal with the Model from the reset() method, and don't initialize properties for an update operation in the reset() method.

The reset() method was mainly added so you could reset check boxes to false. Then, the selected check boxes will be populated when Struts populates the form. The HTTP protocol sends only selected check boxes. It does not send unselected check boxes. (Examples of working with check boxes in the reset() method are in the tutorial section of this chapter.)

Understanding ActionForm's validate() Method

The purpose of the validate() method is to check for field validation and relationships between fields. Do not perform business logic checks in the validate() method; it is the job of the action to work with the Model to perform business logic checks.

A field validation would check to see if a field is in a certain range, if a field was present, a certain length, and more. A relationship validation would check the relationship between the fields. Checking to see if the start date is before the end date is a good example of a relationship validation. Another relationship validation is checking to see if the password and the check password field are equal. (Examples of performing validation can be found in the tutorial section of this chapter.)

The Do’s and Don’ts of Automatic Type Conversion

ActionForms can be strongly typed. Struts will convert Strings and String Arrays into primitive and primitive arrays.

Struts converts the request parameters into a HashMap and then uses common BeanUtils to populate the ActionForm with the request parameters. (This occurs in the `processPopulate` method of the `RequestProcessor` by calling the `populate` method of `RequestUtils.)` The interesting thing about this is that BeanUtils will perform type conversion from the strings coming from the request parameter to any primitive or primitive wrapper class.

At first, this seems like a boon. Problems arise when you implement validation. Let’s say a user mistypes an integer field with the letters “abc”. BeanUtils will convert “abc” to 0 if it corresponds to an int property of the ActionForm. This is bad news. Even if you did bounds checking in the validate() method of the ActionForm and the 0 field was not allowed, when control forwarded back to the input JSP the user will not see the “abc” they typed in; they will see 0. Even worse is if 0 is a valid value for your application, then there is no way to check to see if the user entered in the right number for the field.

Thus, you have to follow this rule when using the automatic type conversion: if the user types in the value, then make the ActionForm property representing the field a string. Usually this means if the field is rendered with html:text, html:textarea, or html:password, then make the field a string. This does not apply to drop-down boxes (html:select), check boxes (html:checkbox), and the like.

Tip: If the user types in the value, then make the property a string.

What an ActionForm Is

Data Supplier: Supplies Data to html:form

An ActionForm supplies data to be displayed by JSP pages. In the CRUD realm, the ActionForm would be used to transfer data to the html:form tag of an update.jsp page. In fact, the html:form tag will not work unless the ActionForm is present and in the correct scope. In this role, the ActionForm supplies data to the html:form.

Data Collector: Processes Data from html:form

An ActionForm receives form data (request parameters) from browsers usually with forms that were rendered with html:form. Struts converts that data into an ActionForm. Thus, instead of handling request parameters directly, you would work with the ActionForm (possibly strongly typed).

Action Firewall: Validates Data before the Action Sees It

An ActionForm acts like a traffic cop. If validation is turned on for an action, the action will never be executed unless the ActionForm's validate() method says the ActionForm is valid. The action in turn deals with the Model; the Model is never passed bad data from the view. This is a boon from an architecture standpoint. Your Model just needs to worry about business rule violations not fumbling finger violations. This turns out to be a good separation of concern that makes the Model easier to test and validate.

What an ActionForm is Not

ActionForms can be abused and used in manners that were not intended by the Struts framework. There are some things to keep in mind when using ActionForms.

Not Part of the Model or Data Transfer Object

ActionForms are not part of the Model and should not be used as data transfer objects between the controller (Actions) and the Model. The first reason is that the Model should be “Struts agnostic.” The second reason is that ActionForms are not strongly typed as they are used to perform field validations and reflect bad fields back to the user to see and fix. Often times there are one-to-one relationships between ActionForms and the Model DTOs (data transfer objects). In that case, you could use BeanUtils.copyProperties to move and convert data from the ActionForm to the Model DTOwhere appropriate.

Not an Action, Nor Should It Interact with the Model

The ActionForm should not deal with the Model in the reset() method or the validate() method. This is the job of the Action. ActionForms have a limited set of responsibilities: act as a transfer object, reset the fields to default values, and validate the fields. If you are doing more than that in your ActionForm, then you are breaking how Struts delimits the areas of concern. (I am okay with breaking the rules, as long as you know what the rules are and have a good reason for breaking them.)

Reducing the Number of ActionForms

A common concern with Struts is the number of ActionForms that need to be created. Essentially, you have to create an ActionForm for each HTML form. There are many strategies to get around this.

Super ActionForms

One common strategy to get around the number of ActionForms is to use a super class ActionForm that has many of the fields that each of the other HTML forms need. This works out well if a lot of forms are similar, which can be the case with some web applications.

Advantage

The advantage to this approach is that you reduce the number of fields you need to add to each ActionForm by having a super class ActionForm that contains a lot of the common fields.

Delta

One of the disadvantages to this approach is you end up carrying around a lot of fields that some Actions don’t care about. Essentially, you have opted to trade the cohesiveness of an ActionForm to reduce the number of classes in your system.

This only works where a lot of the forms are similar.

Mapped Back ActionForms

ActionForms can be mapped back. A mapped back ActionForm has one or more mapped back properties. A mapped back property is like an indexed property, but instead of indexing the property with an int, you index it with a string. You can reference objects in the property map using a special syntax.

Advantage

The advantage of using mapped back ActionForms is that you can create dynamically extensible ActionForms. The ActionForms can receive and transmit properties at runtime. If you need extra properties for a form, you can have a mapped back field that represents the data for that form instead of creating an ActionForm for every possible combination of fields.

Delta

Maps are no substitute for Java classes in many scenarios. The use of mapped back ActionForms should be a last option, not a first choice. They are harder to document, as the items in the map are truly dynamic. However, when you need a form to have dynamic ActionForms (possibly option fields), nothing can take the place of mapped back ActionForms.

(Examples of mapped back dynamic properties can be found in the tutorial section of this chapter.)

DynaActionForms

In teaching, consulting Struts, and developing with Struts, I have found that DynaActionForms are either readily embraced or consistently avoided. The idea behind DynaActionForms is that instead of creating an ActionForm for each HTML form, you instead configure an ActionForm for each HTML form.

Advantage

Some folks feel creating an ActionForm class for each HTML form in your Struts application is time-consuming, maintenance-intensive, and plain frustrating. With DynaActionForm classes, you don't have to create an Action-Form subclass for each form and a bean property for each field. Instead, you configure a DynaActionForm's properties, type, and defaults in the Struts configuration file.

Delta

You still have to create the DynaActionForm in the Struts configuration file. When you use the DynaAction-Form, you have to cast all the properties to their known type. Using a DynaActionForm is a lot like using a Hash-Map. In your Action, if you are accessing a DynaActionForm and misspell a property name, the compiler will not pick it up; instead, you will get a runtime exception. If you cast an integer to a float by mistake, the compiler will not pick it up; you will get a runtime exception. DynaActionForms are not type safe. If you use an IDE, code completion does not work with DynaActionForms. If you override the reset() method or validate() method, you defeat the purpose of having a DynaActionForm. Finally, DynaActionForms are not really dynamic, as you still have to change the configuration file and then restart the web application to get Struts to recognize an additional field.

Tip: As you can probably tell, I think using DynaActionForms is less than ideal. However, I will cover them. I feel DynaActionForms are not really dynamic at all since you have to restart the web application when you change them. I prefer to code my forms in Java instead of XML. I find DynaActionForms no more dynamic than using Java classes. I prefer to create my ActionForms by subclassing ActionForm and using bean properties. I find that modern IDEs make short work of adding JavaBean properties. I see no real advantage to using DynaActionForms over ActionForms. If I want to make an ActionForm dynamic, I add a mapped back property. I have worked on projects that forced DynaActionForms. I much prefer regular ActionForms. With subclassing Action-Forms, you get strongly typed properties, IDE code completion, and XDoclet support. The XDoclet material is covered when you cover the Validator framework.

Session vs. Request Scope ActionForms

A reoccurring question about ActionForms is which scope should I put them in: session or request? The answer is: it depends. Putting ActionForms in session scope may work out really well for web applications with rich user interfaces. ActionForms in the session scope will ease the development process.

Whether you put ActionForms in session scope or request scope depends on what type of application you are building. If you are building an application similar to eBay or Amazon.com, then a different set of rules will apply than if you are writing an intranet application for a company with 500 workers. Don't exclude putting anything in session scope as a knee jerk reaction. Putting objects like ActionForms into session scope can make it easier to create a rich GUI environment.

However, as a general rule, you should limit how much you put into session scope as it uses up memory resources until either you remove the object from scope or the user's session times out. Resources are usurped further when you implement session sharing via a cluster of J2EE application servers because you are now eating up both memory resources and network bandwidth. This is not a suggestion that you refrain from putting anything into session scope, but simply a warning that you are careful with what you put into session scope.

To determine how much you can put into session scope, you should do some capacity planning and hardware architecture planning for your web application. How many users will use the application? Is hardware failover required? Is session failover required? Will you use load balancing? Will you focus on scaling up or scaling out? How much down time is allowed? How much money can be spent on hardware?

If you decide to put ActionForms into session scope, you can help conserve resources in a few ways. If you are building a wizard-style interface like a multi-step User Registration form, be sure to remove the ActionForm from session scope on the last step of the wizard or when the user presses the Cancel button. For a shopping cart, make sure you remove the shopping cart ActionForm from session scope after the user finishes checking out. Always implement a log out feature. Study how the web application is going to be used and only make the session timeout as long as is needed; this process can be refined once the site goes live by studying how users are using the system. (Ex. One company I consulted with set the session time out to six minutes, which was perfect for their application.)

Note: I helped create an ASP (application service provider systems) that almost always put ActionForms into request scope. We used hidden fields and cookies to help manage state. I've also helped create a B2B application with a known number of users with a rich HTML GUI that almost always put ActionForms into session scope.

Continue the Tutorial

This section adds some code examples to the Struts concepts you just covered. If you have not read the first chapter, you may need to as it covers simple uses of ActionForms.

Make User Registration a Multi-Step Process: Part 1

The purpose of this tutorial is to demonstrate implementing ActionForms into session scope for wizard-style interfaces.

When you create a wizard-style interface, it impacts the way that you do validation. For one thing, you need to be careful how you implement validation. You cannot validate the whole form in one step, as the form fields are spread across multiple pages.

You are going to create a multi-step user registration wizard. First, the user will enter in a username, email, and password. Second, the user will enter in the rest of their information. The first step puts a User object in session scope. The second step puts the completed User into the database.

To turn the user registration example into a multi-step process, you need to do the following:

1. Create a Struts unit test for the new action.

   Writing a unit test for the action will help you visualize what you need the action to do.

   Create a test that subclasses MockStrutsTestCase (see Chapter 2 for more detail) :

   ```
   public class UserRegistrationMultiActionTest extends MockStrutsTestCase
   ```

   Override the setup() method to mock the database:

   ```
   public void setUp() throws Exception {

      /* Always call the superclass setUp method */
      super.setUp();

      /* Get the Servlet Context from the ActionServlet */
      ServletContext context =
                     this.getActionServlet().getServletContext();

      /* Grab the moduleConfig from the ServletContext */
      ModuleConfig moduleConfig =
                  RequestUtils
                  .getModuleConfig(this.getRequest(),context);

      /* Create a Mock DataSource */
   ```

```
    DataSource source =
                JDBCTestUtils.getMockedDataSource();

    /* Get the connection from the DataSource */
    connection = source.getConnection();

    /* Create the database table for the User */
    JDBCTestUtils.createDB(connection);

    /* Put the datasource where Struts
                 will find it and deliver it to the Action */
    context.setAttribute("userDB" +
                 moduleConfig.getPrefix(),source);

}
```

This is similar to what you did before. (I opted not to create a super class, which I would do if the tests were very similar, as is the case with this and the last StrutsTestCase you wrote.) You may want to review Chapter 2 for more details on what this code does.

Create two tests. The first test simulates what the user does in the first step of the wizard:

```
public void testUserRegistrationFirstPage() {
    /* Primary key */
    String email = "TEST1rhightower@arc-mind.com";

    /* Set the action path */
    setRequestPathInfo("/userRegistrationMultiPage1");

    populateUserForm(email);
    addRequestParameter("page", "1");

    /* Run the action through the ActionServlet */
    actionPerform();

    /* Make sure that the user was not
                created in the system yet */
    assertTrue(JDBCTestUtils.userNotExists(connection,
                                                email));

    /* Make sure the user data is put into scope */
    Object user =
                  this.getSession()
                  .getAttribute(
                      UserRegistrationMultiAction.USER_KEY);

    assertNotNull(user);

    /* Make sure the user form is put into scope */
    ActionForm form = (ActionForm)
                        this.getSession().getAttribute("user");

            assertNotNull(form);


    /* Verify that the action forwarded to success */
    verifyForward("success");

    /* Verify that there were no exceptions */
    verifyNoActionErrors();
}
```

The first thing you do is specify the path of the Action: `setRequestPathInfo("/userRegistrationMultiPage1")`.

Future Step: This means that later you will need to add an action mapping for /userRegistrationMultiPage1. The test is like a template for what you will need to do in the code.

Next, you use the `populateUserForm(email)` helper method, which simply populates the form using `addRequestParameters` of `MockStrutsTestCase` as follows:

```
private void populateUserForm(String email) {
    addRequestParameter("userName", "KingAdRock");
    addRequestParameter("firstName", "Rick");
    addRequestParameter("lastName", "Hightower");
    addRequestParameter("email", email);
    addRequestParameter("phone", "555-1212");
    addRequestParameter("fax", "555-1212");
    addRequestParameter("password", "555-1212");
    addRequestParameter("passwordCheck", "555-1212");
}
```

The test denotes that this is the first step in the multi-step process by adding a page request parameter (`addRequestParameter("page", "1")`). Future Step: To do this, you will add a page property to the ActionForm.

Once you set the path and the request parameters, you need to run the action through the ActionServlet by using the actionPerform() method of `MockStrutsTestCase`.

You will need to create a helper method to check to see if the user is not in the system. This requires some knowledge of JDBC and DAO (data access objects) that you created earlier. I created a helper method called JDBCTestUtils.userNotExists. Future step: you want to make sure the user is not in the system until the second step. To do this, you try to grab the user out of session scope, and then just check if the user object is null (`assertNotNull(user)`). Then, you do the same thing with the ActionForm.

Note: In the real world, you can use User Stories (XP), Feature Stories (FDD), Specifications, or Use Case Scenarios (RUP), to help you write your test case. You should have a test for each feature and scenario described.

Last, you can check to see if the action forwarded to success and that the there were no errors reported (`verifyForward("success"),verifyNoActionErrors()`).

The next step tests the second step of your user registration wizard, which is similar to the first step. (Read the code comments.)

```
public void testUserRegistrationSecondPage() {

   /* Primary key */
   String email = "TEST1rhightower@arc-mind.com";

   /* Set the action path */
   setRequestPathInfo("/userRegistrationMultiPage2");

   populateUserForm(email);

   /* Set the page to the second page */
   addRequestParameter("page", "2");

   /* Put an empty user into session scope,
               simulate step 1 of wizard */
   User theUser = new User();
   this.getSession()
            .setAttribute(
               UserRegistrationMultiAction.USER_KEY, theUser);

   /* Run the action through the ActionServlet */
   actionPerform();

   /* Make sure that the user was
              created in the system */
   assertTrue(JDBCTestUtils.userExists(connection,
                                  email));

   /* Make sure the user data is no longer put
               in session scope. */
   Object user = this.getSession().
            getAttribute(
               UserRegistrationMultiAction.USER_KEY);
   assertNull(user);

   /* Make sure the user form is no longer in
                                  session scope. */
   ActionForm form = (ActionForm)
              this.getSession().getAttribute("user");
   assertNull(form);
   /* Verify that the action forwarded to success */
   verifyForward("success");

   /* Verify that there were no exceptions */
   verifyNoActionErrors();
}
```

This wizard is a two-step process: after this step the user should be in the database, so the test checks to make sure the new user is in the database. Now, you would want to write test cases for the user hitting the Cancel button and the user entering invalid data. Check out the example code to see these tests.

The tests above show that you configured Struts correctly (struts-config.xml) and you wrote your action correctly. The nice thing about these tests is that they will always fail unless you implement the action and configure Struts (struts-config.xml) correctly. Later on when you make changes to the code, you can rerun the test to make sure that you did not break the test.

The problem with the tests above is that they do not test the JSPs that are involved. In order to do so, you could write a jWebUnit test as follows:

```
package strutsTutorial.testJSP;

import net.sourceforge.jwebunit.WebTestCase;

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

public UserRegistrationMultiJSPTest(String name) {
   super(name);
}

public void setUp() {
   getTestContext()
             .setBaseUrl(
                "http://localhost:8080/strutsTutorial");
}

private void populateUserForm1(String email) {
   setFormElement("email", email);
   setFormElement("userName", "KingAdRock");
   setFormElement("password", "555-1212");
   setFormElement("passwordCheck", "555-1212");

}
```

```
    private void populateUserForm2() {

       setFormElement("firstName", "Rick");
       setFormElement("lastName", "Hightower");
       setFormElement("phone", "555-1212");
       setFormElement("fax", "555-1212");
    }

    public void testMultiStepUserReg() {

       /* Step 1 */
       beginAt("/userRegWizard.do");
       populateUserForm1("test@test.com"
                            + System.currentTimeMillis());
       submit();
       assertTitleEquals("User Registration Page 2");

       /* Step 2 */
       populateUserForm2();
       submit();
       assertTitleEquals(
                  "User Registration Was Successful!");

    }

    public void testMultiNoEmail() {

       /* Step 1 */
       beginAt("/userRegWizard.do");
       populateUserForm1(null);
       submit();
       assertTitleEquals("User Registration Page 1");


    }

    public void testFormSubmitDuplicates() {
       String email = "test@test2.com" +
                              System.currentTimeMillis();

       for (int i = 0; i < 2; i++) {

             beginAt("/userRegWizard.do");
             populateUserForm1(email);
             submit();

             populateUserForm2();
             submit();
```

```
            }

            assertTitleEquals(
                      "User Registration Had Some Problems");
        }
              <... other tests ...>
        }
```

The nice thing about the test above is that it tests the complete application (all tiers). See Chapter 2 for more detail on jWebUnit.

Tip: Make a test sandwich. When I write Struts code, I always create a test sandwich. I read the User Story. Then, I write my Model tests with JUnit, and implement that part of the Model. I write my test for the action using StrutsTestCase, then implement that part of the controller. The nice thing about StrutsTestCase is that I can test a single action in a multi-step process. (I might use Mock objects to simulate parts of the Model.) I create JSP level tests with jWebUnit, then I write my JSPs. You can do this one JSP at a time.

Now that you have written your tests, you can start to code and configure Struts.

2. Create an action mapping for each step.

   After completing your tests, you have a pretty good idea how to implement the code. You will need three mappings. One mapping will load the first form (JSP page) of the wizard. The second mapping will handle form submission from the first step and load the second form of the wizard. The third mapping handles form submission from the second step.

   The first mapping just loads the first form.

```
<action path="/userRegWizard"
        forward="/userRegistrationPage1.jsp" />
```

   Notice that this mapping just forwards to the JSP page `/userRegistrationPage1.jsp`.

The second mapping handles the form submission from the first form.

```
<action path="/userRegistrationMultiPage1"
        type="strutsTutorial.UserRegistrationMultiAction"
        name="userRegistrationForm"
        attribute="user"
        scope="session"
        input="/userRegistrationPage1.jsp">

          <exception
            type="org...NestableRuntimeException"
            key="userRegistration.sql.exception"
            path="/userRegistrationException.jsp" />

       <forward name="success"
                path="/userRegistrationPage2.jsp" />

</action>
```

The nice thing about writing the action mapping before you get started is that the action mapping has all the information about the classes (Action handler and ActionForm) and the forwards involved. Notice that this is associated with the ActionForm you created in the first tutorial.

Review: The path attribute specifies that this action mapping will handle requests from the path `/userRegistrationMultiPage1`. The type attribute specifies that the action handler of this mapping is `strutsTutorial.UserRegistrationMultiAction`. If you are having a problem understanding this, read the first chapter, which explains all of the pieces.

The third mapping handles the form submission from the second registration form.

```
<action path="/userRegistrationMultiPage2"
        type="strutsTutorial.UserRegistrationMultiAction"
        name="userRegistrationForm"
        attribute="user"
        scope="session"
        input="/userRegistrationPage2.jsp">


         <exception
            type="org... NestableRuntimeException"
            key="userRegistration.sql.exception"
            path="/userRegistrationException.jsp" />

          <forward name="success"
                   path="/regSuccess.jsp" />

</action>
```

Notice that the use of the action handler to handle this step.

3. Modify the ActionForms validate() method to handle multiple steps.

   The issue for validation is that the form gets submitted multiple times. Thus, you need to keep track of which step you are on and only validate the fields for that step. To keep track of the step, create a property to hold the current page number as follows:

```
public class UserRegistrationForm extends ActionForm {
     ...
     private int page;

     public int getPage() {
   return page;
     }

     public void setPage(int i) {
   page = i;
     }
     ...
```

The validate() method of the ActionForm gets passed an action mapping and an HttpServletRequest. You can use the request object to check the value of the page parameter as follows:

```
public ActionErrors validate(
   ActionMapping mapping,
   HttpServletRequest request) {
   ActionErrors errors = new ActionErrors();

   String strPage = request.getParameter("page");
   int page = 0;
   boolean all=false;

   if (strPage != null) {
        page = Integer.parseInt(strPage);
   } else {
        all=true;
   }
```

Then only validate the fields for that step as follows:. (The page denotes the step.)

```
if (page == 1 || all) {
     log.debug("validating page 1");
     addErrorIfBlank(errors, email, "email", request);
     addErrorIfBlank(errors, userName,
                                      "userName", request);
     addErrorIfBlank(errors, password,
                                     "password", request);
     addErrorIfBlank(errors, passwordCheck,
                                 "passwordCheck", request);

     if (!errors.isEmpty())
          return errors;

     log.debug("check if password fields match");
     if (!password.equals(passwordCheck)) {
          errors.add(
               "password",
               new
                       ActionError("user.passwrd.nomatch"));
     }
} else if (page == 2 || all) {
     log.debug("validating page 2");
     addErrorIfBlank(errors, firstName,
                                    "firstName", request);
     addErrorIfBlank(errors, firstName,
                                     "lastName", request);
     addErrorIfBlank(errors, phone,
                                      "phone", request);
} else {
     throw new IllegalStateException(
                              "not a valid page");
}
```

Notice that you validate the email, userName, password, and passwordCheck for the first step. The firstName, lastName, and phone get validated for the second step. Both steps use the addErrorIfBlank helper method, which is implemented as follows:

```
private void addErrorIfBlank(
   ActionErrors errors,
   String fieldValue,
   String fieldName,
   HttpServletRequest request) {

        /* Check to see if the field is blank */
        if (fieldValue == null || fieldValue.trim().equals("")) {

        /* Grab the default resource bundle out
                     of request scope */
        MessageResources defaultBundle =
              (MessageResources)
                           request.getAttribute(Globals.MESSAGES_KEY);

        log.debug("looking up label userRegistration." +
                               fieldName);

        String label =
          defaultBundle.getMessage("userRegistration." +
                          fieldName);

        log.debug("label = " + label);

        /* Add the error to errors .*/
        errors.add(fieldName, new
                     ActionError("errors.blank",
                                        label));

   }

}
```

In order for the code above to work, you need to add a new message named errors.blank to the resource bundle as follows:

```
errors.blank=The {0} field was blank
```

You look up the label (`defaultBundle.getMessage()`), and then pass the label as the first argument of the message (`new ActionError("errors.blank", label)`).

4. Create an Action that uses this ActionForm.

Now that you have the implemented the form, you can write a test that uses this form to handle step 1 and step 2. (Read the code comments.)

```
public class UserRegistrationMultiAction extends Action {

private static Log log = LogFactory.getLog(...);
protected static final String USER_KEY =
   "strutsTutorial.UserRegistrationMultiAction.USER";

public ActionForward execute(
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
   throws Exception {
   log.trace("In execute of UserRegistrationAction");

   if (isCancelled(request)) {
         log.debug("Cancel Button was pushed!");
         request.getSession()
                  .removeAttribute(USER_KEY);

                 request.getSession()
                 .removeAttribute(
                  mapping.getAttribute());

                  return mapping.findForward("welcome");
   }
   UserRegistrationForm userForm =
                             (UserRegistrationForm) form;

   log.debug("userForm email " + userForm.getEmail());

   int page = userForm.getPage();
   if (page == 1) {
         return processPage1(mapping, userForm,
                                    request, response);
   } else if (page == 2) {
         return processPage2(mapping, userForm,
                                    request, response);
   } else {
         throw new IllegalStateException(
               "Page number not supported");
   }

}

public ActionForward processPage1(
   ActionMapping mapping,
   UserRegistrationForm userForm,
   HttpServletRequest request,
   HttpServletResponse response)
   throws Exception {

   log.trace("In processPage1 of UserRegistration");

   /* Create a User DTO and copy
               the properties from the userForm */
   User user = new User();
   BeanUtils.copyProperties(user, userForm);

   request.getSession().setAttribute(USER_KEY, user);

   return mapping.findForward("success");

}

public ActionForward processPage2(
   ActionMapping mapping,
   UserRegistrationForm userForm,
   HttpServletRequest request,
   HttpServletResponse response)
   throws Exception {
   log.trace("In processPage2 of UserRegistration");

   /* Get the user in session scope,
             fail if not present */
   User user = (User)
                       request
                       .getSession().getAttribute(USER_KEY);
   if (user == null)
         throw new java.lang.IllegalStateException(
               "Missing user in session scope");

   /* Grab the datasource */
   DataSource dataSource =
                      getDataSource(request, "userDB");
   Connection conn = dataSource.getConnection();

   /* Create UserDAO */
   UserDAO dao = DAOFactory.createUserDAO(conn);

   /* Create a User DTO and copy
                           the properties from the userForm */
   BeanUtils.copyProperties(user, userForm);

   /* Use the UserDAO to insert the
                              new user into the system */
   dao.createUser(user);
   /* Clean up objects you no longer use */
   request.getSession()
                   .removeAttribute(USER_KEY);
   request.getSession()
                   .removeAttribute(mapping.getAttribute());

   /* Put item in request scope */
   request
            .setAttribute(mapping.getAttribute(),userForm);

   return mapping.findForward("success");

}

}
```

The functionality is very similar to the one step user registration action that you developed earlier. Notice that the execute() method uses the page property of the ActionForm to decide to execute processPage1 or processPage2 as follows:

```
int page = userForm.getPage();
if (page == 1) {
     return processPage1(mapping, userForm,
                              request, response);
} else if (page == 2) {
     return processPage2(mapping, userForm,
                              request, response);
} else {
     throw new IllegalStateException(
          "Page number not supported");
}
```

5. Create two JSPs for each step that pass hidden page parameters.

   Now that you are done writing the action and your unit tests pass, you need to write the JSP pages. Basically, you are going to split the JSP page you had before into two JSP pages. The first JSP page will have five fields: userName, email, password, passwordCheck, and page (the hidden field) and will submit to `/userRegistrationMultiPage1` as follows:

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<html>

  <head>
    <title>User Registration Page 1</title>
  </head>

  <body>
    <h1>User Registration</h1>

<html:errors/>

    <table>
    <html:form action="userRegistrationMultiPage1">
      <tr>
        <td>
          <bean:message key="userRegistration.userName" />*
        </td>
        <td>
          <html:text property="userName" />
        </td>
      </tr>
      <tr>
        <td>
          <bean:message key="userRegistration.email" />*
        </td>
        <td>
          <html:text property="email" />
        </td>
      </tr>
      <tr>
        <td>
          <bean:message key="userRegistration.password" />*
        </td>
        <td>
          <html:password property="password" />
        </td>
      </tr>
      <tr>
        <td>
          <bean:message key="userRegistration.password" />*
        </td>
```

```
          <td>
            <html:password property="passwordCheck" />
          </td>
        </tr>
        <tr>
          <td>
            <html:submit />
          </td>
          <td>
            <html:cancel />
          </td>
        </tr>

<!-- Added hidden variable so Action execute,
           and ActionForm validate knows which page its on -->
<html:hidden property="page" value="1"/>

    </html:form>
    </table>
  </body>
</html>
```

Notice you added the hidden page property using the html:hidden custom tag. The second page has the rest of the fields and submits to the action mapping linked to `/userRegistrationMultiPage2.do` as follows:

```
...
<table>
    <html:form action="userRegistrationMultiPage2">
      <tr>
        <td>
          <bean:message key="userRegistration.firstName" />*
        </td>
        <td>
          <html:text property="firstName" />
        </td>
      </tr>
      <tr>
        <td>
          <bean:message key="userRegistration.lastName" />*
        </td>
        <td>
          <html:text property="lastName" />
        </td>
      </tr>
      ... rest of the fields
      <tr>
        <td>
          <html:submit />
        </td>
        <td>
          <html:cancel />
        </td>
      </tr>
<!-- Added hidden variable so action and
           actionform knows which page its on -->
<html:hidden property="page" value="2"/>

    </html:form>
    </table>
  </body>
</html>
```

If you finished this, then you successfully created a multi-step user registration wizard. You can use this same technique to create other multi-step workflows.

Create the Address JavaBean Property: Part 2 (Nested Beans)

The forms you create so far are pretty one-dimensional. ActionForms can use nested JavaBean properties. For example, let's say that you wanted to collect address information from several different forms. Instead of repeating the fields in every form, you will create a DTO that can be used in multiple locations.

Your Address DTO is a JavaBean like the following:

```
public class Address implements Serializable {
private String line1;
private String line2;
private String postalCode;
      ...


public String getLine1() {
   return line1;
}

public String getLine2() {
   return line2;
}
public String getPostalCode() {
   return postalCode;
}

public void setLine1(String string) {
   line1 = string;
}
public void setLine2(String string) {
   line2 = string;
}

public void setPostalCode(String string) {
   postalCode = string;
}
      ...

}
```

You can declare a property in the ActionForm of type address as follows:

```
public class UserRegistrationForm extends ActionForm {
      ...
private Address address = new Address();

public Address getAddress() {
   return address;
}

public void setAddress(Address address) {
   this.address = address;
}
     ...
```

Now you can add the following fields to the JSP page's html:form to access the nested properties you just created as follows:

```
<html:form action="userRegistrationMultiPage2">
  ...
  <tr>
    <td>
      <bean:message key="userRegistration.line1" />
    </td>
    <td>
      <html:text property="address.line1" />
    </td>
  </tr>
  <tr>
    <td>
      <bean:message key="userRegistration.line2" />
    </td>
    <td>
      <html:text property="address.line2" />
    </td>
  </tr>
  <tr>
    <td>
      <bean:message key="userRegistration.postalCode" />
    </td>
    <td>
      <html:text property="address.postalCode" />
    </td>
  </tr>
        ...
```

Notice that you use the notation address.line1, which causes Struts to populate the line1 field associated with the nested Address bean (`form.getAddress().setLine1("value entered")`). (Remember to add the new address labels to the resource bundle.)

Using nested properties can help you reuse common properties.

Create the Phone JavaBean Indexed Property: Part 3 (Indexed Properties)

In addition to using JavaBean nested properties, you can use indexed properties.

The forms you created so far are pretty one-dimensional. Let's say that a user can have multiple phone numbers (fax, phone, mobile, etc.). Instead of one nested JavaBean, you want several. You could do this by using a nested number of Phone DTOs.

Your Phone DTO is a JavaBean that looks like the following code:

```
public class Phone implements Serializable {
private String countryCode;
private String areaCode;
private String phone;
private String extention;

public String getAreaCode() {
   return areaCode;
}

public String getCountryCode() {
   return countryCode;
}

public String getExtention() {
   return extention;
}

public String getPhone() {
   return phone;
}

public void setAreaCode(String string) {
   areaCode = string;
}

public void setCountryCode(String string) {
   countryCode = string;
}
public void setExtention(String string) {
   extention = string;
}

public void setPhone(String string) {
   phone = string;
}

}
```

You can declare an indexed property in the ActionForm of type Phone as follows:

```
public class UserRegistrationForm extends ActionForm {
      ...
private Phone [] phones = new Phone[]
                 {new Phone(), new Phone(), new Phone()};

public Phone getPhones(int index) {
   return phones[index];
}

public void setPhones(int index, Phone value) {
   this.phones[index] = value;
}
```

Now you can use the fields as follows:

```
<html:form action="userRegistrationMultiPage2">
  ...
  <tr>
    <td>
      <bean:message key="userRegistration.mobilePhone" />
    </td>
    <td>
      <html:text property="phones[1].areaCode" />
    </td>
  </tr>
  ...
```

Notice that you use the notation **`phones[1].areaCode`**, which causes Struts to populate the first phone's area-Code field associated with the indexed phones property (form.getPhones(1).setAreaCode("value entered")).

Using nested indexed properties can help create forms that have one-to-many relationships.

Convert the User Registration to Use DynaActionForms: Part 4 (DynaActionForms)

To configure a DynaActionForm in struts-config.xml, you use a form-bean element as with normal Action-Forms. The type of the form bean must be org.apache.struts.action.DynaActionForm or a derived class. You then add form-property elements to declare the properties of the form.

To use a DynaActionForm, add the following to your form-beans element:

```
<form-bean name="userRegistrationDynaForm"
          type="org.apache.struts.action.DynaActionForm">

    <form-property name="userName"
                   type="java.lang.String" />
    <form-property name="email"
                   type="java.lang.String" />
    <form-property name="password"
                   type="java.lang.String" />
    <form-property name="passwordCheck"
                   type="java.lang.String" />
    <form-property name="firstName"
                   type="java.lang.String" />
    <form-property name="lastName"
                   type="java.lang.String" />
    <form-property name="phone"
                   type="java.lang.String"
                   initial="(520) "/>
    <form-property name="fax" type="java.lang.String" />
    <form-property name="page"
                             type="java.lang.Integer" />
 </form-bean>
```

Then create an action mapping that uses this ActionForm:

```
<action path="/userRegistrationDyna"
        type="strutsTutorial.UserRegistrationAction"
        name="userRegistrationDynaForm"
        attribute="user"
        input="/userRegistrationDyna.jsp">
          ...
          <forward name="success"
                   path="/regSuccess.jsp" />
          <forward name="failure"
                   path="/regFailure.jsp" />
</action>
```

Create userRegistrationDyna.jsp and an action mapping to get to the page:

```
<action path="/userDynaRegForm" forward="/userRegistrationDyna.jsp"/>
```

Then modify the action to use the new DynaActionForm:

```
public class UserRegistrationAction extends Action {

     ...
public ActionForward execute(...
                  /* Cast the form into a DynaActionForm */
           DynaActionForm userDynaForm =
                     (DynaActionForm) form;
        log.debug("userForm email " +
                  userDynaForm.get("email"));
```

Notice how the action accesses the email property of the DynaActionForm (`userDynaForm.get("email")`). Working with DynaActionForms is a lot like using a HashMap.

The nice thing about DynaActionForms is that the BeanUtils framework treats them just like regular JavaBeans. Thus, the following code does not really change from the static version of ActionForm, as shown below:

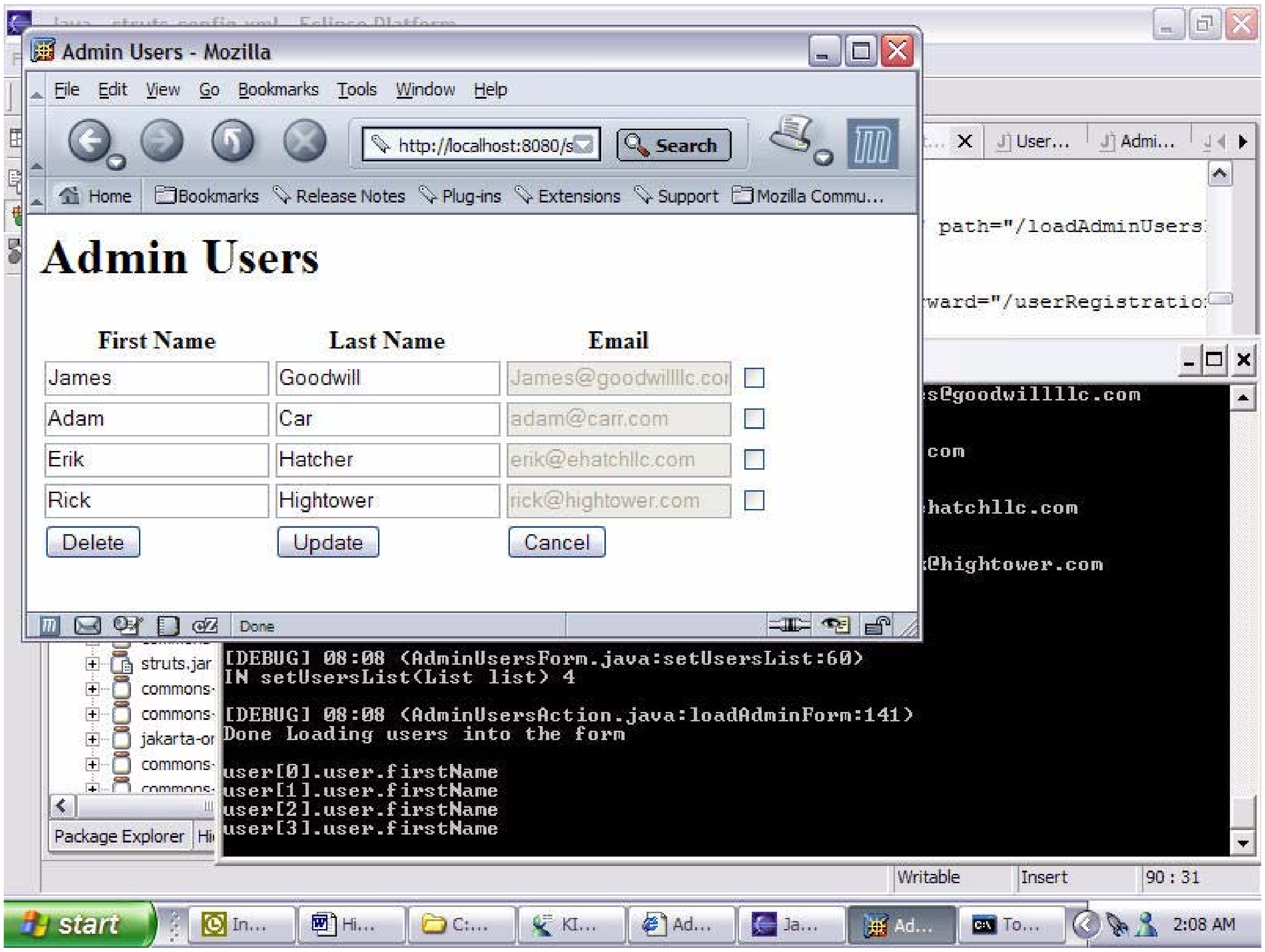
```
/* Create a User DTO and copy the properties
     from the userForm */
User user = new User();
BeanUtils.copyProperties(user, form);
```

Create the Management Feature to Edit/Delete Listings

This example covers creating a master detail form. Basically, you create a master ActionForm that has details (indexed properties). This example will demonstrate how to clear radio check boxes in the reset() method of the ActionForm.

You will create a user administration form that administers users. You will be able to edit multiple users at once, and delete users out of the system. Each user will have a check box by them. If you click the check box and click the Delete button, the user will be deleted. You will also be able to update multiple users.

The administration form for this feature is as follows:



Admin Users - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/s
Search
Home Bookmarks Release Notes Plug-ins Extensions Support Mozilla Commu...
Done
Package Explorer
Writable Insert 90 : 31
start
2:08 AM

Admin Users

First Name	Last Name	Email	
James	Goodwill	James@goodwilllc.coı	☐
Adam	Car	adam@carr.com	☐
Erik	Hatcher	erik@ehatchllc.com	☐
Rick	Hightower	rick@hightower.com	☐

Delete Update Cancel

```
path="/loadAdminUsers
ward="/userRegistratio
```

```
s@goodwillllc.com

com

hatchllc.com

@hightower.com

[DEBUG] 08:08 (AdminUsersForm.java:setUsersList:60)
IN setUsersList(List list) 4

[DEBUG] 08:08 (AdminUsersAction.java:loadAdminForm:141)
Done Loading users into the form

user[0].user.firstName
user[1].user.firstName
user[2].user.firstName
user[3].user.firstName
```

Figure 3.2 User Admin

You need to create an ActionForm that represents the form in the picture. You need to add a property for each button and a property that represents the list of users you are editing. The list will be implemented as an indexed bean property.

Here is the new ActionForm for administering users:

```
public class AdminUsersForm extends ActionForm {
private UserLineItem[] users = new UserLineItem[0];
private String deleteButton;
private String updateButton;

public UserLineItem getUser(int index) {
   return users[index];
}

public void setUser(int index, UserLineItem user) {
   this.users[index] = user;
}

public String getDeleteButton() {
   return deleteButton;
}
public void setDeleteButton(String string) {
   deleteButton = string;
}

public String getUpdateButton() {
   return updateButton;
}

public void setUpdateButton(String string) {
   updateButton = string;
}
      ...
}
```

Notice a few things. First, the user indexed property is implemented with an array of UserLineItems. The UserLineItem class represents a row in your listing. A row consists of a check box and user fields. The UserLineItem is implemented as follows:

```
public class UserLineItem implements Serializable {

private User user = new User();
private boolean checked = false;


public User getUser() {
   return user;
}

public void setUser(User user) {
   this.user = user;
}

public boolean isChecked() {
   return checked;
}

public void setChecked(boolean b) {
   checked = b;
}

}
```

Now, one of the problems with handling the form is that HTTP only sends the checked items. It does not send the check boxes that are not checked.. Thus, you have to clear the check boxes in the reset() method of the ActionForm.

```
public void reset(ActionMapping mapping,
                        HttpServletRequest request) {
   log.trace("IN reset");

   if ("submit".equals(mapping.getParameter())) {
         log.trace("reset for submit");
         for (int i = 0; i < users.length; i++) {
               users[i].setChecked(false);
         }
   }

}
```

The above iterates through all of the UserLineItems and unchecks the check box (`users[i].setChecked(false)`) associated with the UserLineItem. Recall that the reset() method gets called before Struts populates the ActionForm; thus, Struts will populate the check boxes that are checked. Notice that the reset() method checks to see if the mapping parameter is equal to submit. You can use the action mapping to denote that the form is being submitted as opposed to being loaded.

Now the missing link to this ActionForm is loading it. You may be tempted to load the ActionForm in the reset() method—don't. You should instead load the ActionForm in an Action.

Create an action that both loads the form and handles the form submission. To do this you will map the same action twice. Set the parameter of the action mapping to "load" to denote you are loading the form data. Set the parameter of the action mapping to "submit" to denote you are handling the form submission as follows:

```
<form-bean name="adminUsersForm"
            type="strutsTutorial.AdminUsersForm" />
     ...
<action path="/loadAdminUsersForm"
        type="strutsTutorial.AdminUsersAction"
        parameter="load"
        name="adminUsersForm"
        validate="false">
        <forward name="success"
                 path="/adminUsersForm.jsp" />
</action>

<action path="/submitAdminUsersForm"
        type="strutsTutorial.AdminUsersAction"
        parameter="submit"
        name="adminUsersForm">
        <forward name="success"
                 path="/loadAdminUsersForm.do" />
</action>
```

The load action mapping turns form validation off using the validate attribute (validate="false"). You turn the form validation off because you are not handling form submission—you are handling loading the form. Struts still loads the form in the right scope. Then the load forwards to the adminUserForm, which will display the form data.

Both of the action mappings are associated with the action handler class strutsTutorial.AdminUsersAction. The AdminUsersAction loads the form by checking to see if the parameter of the action mapping is set to load and then invoke the loadAdminForm helper method as follows:

```
public ActionForward execute(
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
   throws Exception {
   log.trace("In Execute method of AdminUsersAction");

   DataSource dataSource = getDataSource(request,
                                                "userDB");
   Connection conn = dataSource.getConnection();

   /* Create UserDAO */
   UserDAO dao = DAOFactory.createUserDAO(conn);

   AdminUsersForm adminForm = (AdminUsersForm) form;

   if ("load".equals(mapping.getParameter())) {

         return loadAdminForm(mapping, dao, adminForm);

   }else if("submit".equals(mapping.getParameter())) {

         return doSubmitAdminForm(mapping, dao,
                                            adminForm, conn);

   } else {
         throw new IllegalStateException();
   }

}
```

The loadAdminForm use the UserDAO object to look up the users in the system and load them into the Admin-UsersForm as follows:

```
     [AdminUsersAction]
private ActionForward loadAdminForm(
   ActionMapping mapping,
   UserDAO dao,
   AdminUsersForm adminForm) {
   log.debug("Loading users into the form");
   List list = dao.listUsers();

   adminForm.setUsersList(list);

   log.debug("Done Loading users into the form");
   return mapping.findForward("success");
}
```

Notice the loadAdmin() method invokes the setUserList of the AdminUsersForm which is implemented as follows:

```
          [AdminUsersForm]
public void setUsersList(List list) {
   log.trace("IN setUsersList(List list)");

   List newList = new ArrayList(list.size());

   Iterator iter = list.iterator();
   while (iter.hasNext()) {
         UserLineItem userLineItem =
                             new UserLineItem();
         User user = (User) iter.next();
         userLineItem.setUser(user);
         newList.add(userLineItem);
   }

   users =
         (UserLineItem[])
                   newList.toArray(
                     new UserLineItem[newList.size()]);
   log.trace("IN setUsersList(List list) " +
                       users.length);
}
```

This populates the ActionForm. Then, in the JSP form you would add the following code to render the individual rows:

```
[adminUsersForm.jsp]
 <logic:iterate id="lineItem" indexId="index"
               name="adminUsersForm" property="usersList">
   <tr>
     <td>

     <%
      String slineItem = "user[" + index + "]";
      String nest= slineItem + ".user.";

      String checked = slineItem + ".checked";
      String firstName = nest + "firstName";
      String lastName = nest + "lastName";
      String email = nest + "email";
     %>
       <html:text property="<%=firstName%>" />

     </td>
     <td>
       <html:text property="<%=lastName%>" />
     </td>
     <td>
       <html:text disabled="true" property="<%=email%>" />
     </td>
     <td>
     <html:checkbox property="<%=checked%>" />
     </td>
   </tr>
  </logic:iterate>
```

The JSP scriptlet generates the property accessory strings for the current row of the users. Thus, the firstName attribute for the first iteration would be user[0].user.firstName; the second iteration would be user[1].user.first-Name, and so on. If you use JSTL tag libs, you can get rid of the JSP scriptlet. (You will do this when you cover JSTL and Struts EL in Chapter 13.)

The AdminUserAction uses the properties of the button to see if the Update button or the Delete button was clicked as follows:

```
public class AdminUsersAction extends Action {

private static Log log =
              LogFactory.getLog(AdminUsersAction.class);

public ActionForward execute(
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
   throws Exception {
   log.trace("In Execute method of AdminUsersAction");

   DataSource dataSource = getDataSource(
                                   request, "userDB");
   Connection conn = dataSource.getConnection();

   /* Create UserDAO */
   UserDAO dao = DAOFactory.createUserDAO(conn);

   AdminUsersForm adminForm = (AdminUsersForm) form;

   if ("load".equals(mapping.getParameter())) {

        return loadAdminForm(mapping, dao, adminForm);

   } else if ("submit".equals(mapping.getParameter())) {

        return doSubmitAdminForm(mapping, dao,
                   adminForm, conn);

   } else {
        throw new IllegalStateException();
   }

}

private ActionForward doSubmitAdminForm(
   ActionMapping mapping,
   UserDAO dao,
   AdminUsersForm adminForm,
   Connection connection) throws Exception{
   log.debug("User submitted form");

   log.debug("DELETE BUTTON " +
                            adminForm.getDeleteButton());
   if (adminForm.getDeleteButton() != null) {

      return doDelete(mapping, dao, adminForm, connection);

   }else if (adminForm.getUpdateButton()!=null){

      return doUpdate(mapping, dao, adminForm, connection);

   }

   return mapping.findForward("success");
}

private ActionForward doUpdate(ActionMapping mapping,
            UserDAO dao, AdminUsersForm adminForm,
             Connection connection) throws Exception{

   UserLineItem[] items = adminForm.getUsersList();for (int i = 0; i <
items.length; i++) {

         UserLineItem lineItem = items[i];
         log.debug("Line Item Email" +
                  lineItem.getUser().getEmail());

         dao.updateUser(lineItem.getUser());

   }

   connection.close();

   return mapping.findForward("success");
}

private ActionForward doDelete (
   ActionMapping mapping,
   UserDAO dao,
   AdminUsersForm adminForm,
    Connection connection) throws Exception{

   UserLineItem[] items = adminForm.getUsersList();
   for (int i = 0; i < items.length; i++) {

         UserLineItem lineItem = items[i];
         log.debug("Line Item Email" +
                   lineItem.getUser().getEmail());
         if (lineItem.isChecked()) {
              dao.deleteUser(
                        lineItem.getUser().getEmail(),false);
         }
```

```
            }

            connection.close();

            return mapping.findForward("success");

        }

        ...
        }
```

Now the action can handle updates as well as deleting users from the system.

Using Dynamic Properties: Part 5

The forms you created so far are non-dynamic (even the DynaActionForm example was mostly static). ActionForms can use mapped back properties. Mapped back properties have no parallel in the JavaBean world—it is a Struts invention.

To create a mapped back property, declare a “property” in the ActionForm as follows:

```
public class UserRegistrationForm extends ActionForm {
      ...
private HashMap dynamicProps = new HashMap();

public Object getDynamicProps(String key) {
   return dynamicProps.get(key);
}

public void setDynamicProps(String key, Object value) {
   dynamicProps.put(key, value);
}
```

To access the new dynamic property, you modify the JSP page by adding the following field:

```
<html:form action="userRegistration">
  ...
  <tr>
    <td>
      Dyna Prop Bar
    </td>
    <td>
      <html:text property="dynamicProps(bar)" />
    </td>
  </tr>
  <tr>
    <td>
      Dyna Prop Foo
    </td>
    <td>
      <html:text property="dynamicProps(foo)" />
    </td>
  </tr>

  ...
```

Notice that you use the notation `dynamicProps(bar)`, which causes Struts to add a field to the dynamicProps property called bar (form.setDynamicProps(“foo”, “value entered”).

To access the above field from an Action, you would do the following:

```
        String foo = (String) userForm.getDynamicProps("foo");
```

Remember, in the earlier example the string that represents the property could be dynamically generated. Thus, you can really have forms that have dynamic fields (fields created at run time).

Summary

ActionForms function as data transfer objects to and from HTML forms. ActionForms populate HTML forms to display data to the user and also act like an object representation of request parameters, where the request parameters attributes are mapped to the strongly typed properties of the ActionForm. ActionForms are also responsible for performing field validation.

This chapter included two sections. The first section covered the theory and concepts behind ActionForms. The second section covered common tasks that you will need to perform with ActionForms.

The first section covered the life cycle of an ActionForm, which is important when you are debugging your application. The first section also covered key concepts regarding ActionForms summarized as follows: the reset() method is used to prepare the ActionForm that has check boxes. The validate() method is used to validate fields on the ActionForm. If the user can mess something up, they will; therefore, use strings for all fields that the user must type. Otherwise, the Struts validation mechanism will not work. The ActionForm supplies data to the view and collects data from the view; thus, it is a DTO between the Controller and the View. ActionForms also act as a firewall, as you can set them up so that the Actions will not be invoked unless the corresponding ActionForm is valid. Your model should be Struts agnostic, so don't use ActionForms as Model objects.

The second section consisted mostly of examples. The examples covered creating a master detail ActionForm (e.g., Order has LineItems), creating an ActionForm with nested JavaBean properties, creating an ActionForm with nested indexed JavaBean properties, creating an ActionForm with mapped back properties, and loading form data in an ActionForm to display and configure DynaActionForms.

The Validator Framework

The Validator framework is used to validate fields. Validation is done in many forms—for example, a zip code in both a user registration form and an order form. Instead of writing the zip code validation twice (in each of the validate methods of the form beans corresponding to these forms), you can create a general validation mechanism for all zip code numbers in the system.

Since Struts was built with i18N in mind, the validation mechanism that ships with Struts has support for internationalization. The Validator framework provides many common validation rules. In addition to the common validation rules, you can write your own rules using the Validator framework. By the end of this chapter, you will be able to use the common validation rules to validate form fields and create your own validation rules.

This chapter covers the following topics:

- Understanding how the Validator framework integrates with Struts
- Using the Validator framework with static ActionForms and with DynaActionForms
- Working with common validation rules
- Building and using your own validation rules (zip code with plus 4)
- Working with wizards (multistep user registration) by employing the page attribute
- Using the Validator framework and your own custom validation at the same time
- Employing JavaScript validation on the client side

Getting Started with the Validator Framework

It’s hard to get started with the Validator framework, so let’s go right into the tutorial.

Let’s use the Validator framework to validate fields from the user registration form. As part of this user registration, you want the end user to enter a username. The username should start with a letter, consist of at least 5 alphanumeric characters or underscores, and be no longer than 11 characters.

To use the Validator framework, follow these steps:

1. Add the Validator plug-in to the Struts configuration file. The Validator framework integrates with Struts via this plug-in, which is responsible for reading the configuration files for the Validator rules. To use the Validator framework with Struts, you need to add this plug-in after any message resource elements in the Struts configuration file as follows:

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

2. Copy the validator-rules.xml and validation.xml files into WEB-INF (from the blank Struts web application). The validator-rules.xml file serves as the deployment descriptor for validation rule components. Since the user registration is based on the blank WAR file, you don’t have to perform this step for the tutorial.

    For now, you will use a common, preexisting validation rule component so that you do not have to modify the validator-rules.xml file. The validator.xml file enables you to set up the mappings from the ActionForm's property to the rules and any error message for the rules. Examples of both the validator-rules.xml file and the validation.xml file are in the blank starter web application that ships with Struts.

3. Change the ActionForm class (UserRegistrationForm.java) to the subclass ValidatorForm (org.apache.struts.validator.ValidatorForm). The ValidatorForm is the Struts hook into the Validator framework. The ValidatorForm overrides the validate() method of the ActionForm and communicates with the Validator framework to validate the fields of this form.

   Here are the changes you need to make to UserRegistrationForm.java:

```
import org.apache.struts.validator.ValidatorForm;


public class UserRegistrationForm extends ValidatorForm {
…
private String userName;

public String getUserName() {
return userName;
}
public void setUserName(String string) {
userName = string;
}
…
```

4. Add a form to the form set in the validation.xml file. You may recall that the userRegistrationForm is mapped into the struts-config.xml file as follows:

```
<form-beans>
   <form-bean name="userRegistrationForm"
             type="strutsTutorial.UserRegistrationForm" />
```

   Then, the above form is used by the action mapping as follows:

```
<action path="/userRegistration"
        type="strutsTutorial.UserRegistrationAction"
        name="userRegistrationForm"
        attribute="user"
        input="/userRegistration.jsp">
...
          <forward name="success" path="/regSuccess.jsp" />
          <forward name="failure" path="/regFailure.jsp" />
</action>
```

   You must add the mapping from the userRegistrationForm bean definition in the struts-config.xml file to the rules that should be invoked for the individual properties of the userRegistrationForm bean:

```
<formset>
      <form name="user">
       ...
      </form>
</formset>
```

This code states that you are going to write rules for the properties of the form bean that is associated with the attribute user as defined in the struts-config.xml file. The name has to match the attribute value of the form bean for a mapping that you defined in the struts-config.xml file earlier.

Warning! Do not match the name of the form, but the value of the attribute. This gets confusing because if you do not give an action mapping an attribute (attribute="user") , then the value of the attribute defaults to the name of the ActionForm (name="userRegistrationForm"). Thus, if you did not specify the attribute, the attribute would have been userRegistrationForm and you would use userRegistrationForm for the name of the form in the formset in the validation.xml file.

5. Add a field to the form declaration in the validation.xml file corresponding to the userName field. Now that you specified which form you want to associate with rules, you can start associating fields (also known as bean properties) with the predefined rules: the field sub-element maps inputForm's userName property to one or more rules.

```
<form name="user">
     <field property="userName" ...>
             ...
     </field>
</form>
```

6. Specify that the userName field corresponds to the *required, minlength*, and *maxlength* rules:

```
<form name="user">
     <field property="userName"
          depends="required,minlength,maxlength">
             ...
     </field>
</form>
```

The depends attribute of the field element takes a comma-delimited list of rules to associate with this property. Therefore, this code associates the userName property with the required, minlength, and maxlength rules. These rules are some of the many rules built into the Validator framework. These rules are associated with rule handlers and an error message key. For example, if you looked up the minlength rule in the validator-rules.xml file, you would see the following:

```
<validator name="minlength"
   classname="org.apache.struts.validator.FieldChecks"
   method="validateMinLength"
   ...
   depends=""
   msg="errors.minlength">
     ...

</validator>
```

Notice that the validator element defines the minlength rule. It also uses the classname attribute to specify the rules handler, the class that implements the rule. In the example, the handler for this rule is implemented in the class org.apache.struts.validator.FieldChecks by the validateMinLength() method. The msg attribute specifies the key for the message that the framework will look up in the resource bundle to display an error message if the associated fields do not validate. You will need to add this message to the resource bundle.

7. Add the error message for the rules to the resource bundle. Because you are using the common rules, you must import its associated message into the resource bundle for this web application. The validator-rules.xml file has sample messages for all of the rules in a commented section. You may change the wording, but the sample messages are a good guide. Find the sample messages (e.g., errors.minlength) in the comments of validator-rules.xml, and copy and paste the errors to the resource bundle. (If you are using the blank WAR file as the base, the error messages are already in the resource bundle.) The resource bundle for the tutorial is located in resources/application.properties in the src/java directory. Ensure the resource bundle contains the following messages:

```
errors.invalid={0} is invalid.
errors.maxlength={0} cannot be greater than {1} characters.
errors.minlength={0} cannot be less than {1} characters.
errors.range={0} is not in the range {1} through {2}.
errors.required={0} is required.
errors.byte={0} must be a byte.
errors.date={0} is not a date.
errors.double={0} must be a double.
errors.float={0} must be a float.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.short={0} must be a short.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid e-mail address.
```

Notice the three error messages below correspond to the three rules that you are configuring: *required*, *minlength*, and *maxlength*.

```
errors.maxlength={0} cannot be greater than {1} characters.
errors.minlength={0} cannot be less than {1} characters.
errors.required={0} is required.
```

Notice that the maxlength rule message (`errors.maxlength`) takes two arguments ({0} and {1}) as does the minlength rule. The required rule message only takes one argument. You will need to configure values for these arguments. The first argument of all three rules corresponds to the label of the field. The second argument for the length rules corresponds to cardinality of the characters. See java.text.MessageFormat API docs for more information on working with messages and arguments to messages.

8. Specify that the username label is the first argument to the error message. Notice for example that the errors.minlength message takes two arguments. The first argument is the name of the field as it appears to the end user (i.e., the label). The second argument is the value of the minlength variable (you will set up the second argument later). To set up the first argument, use the arg0 element as follows:

```
<form name="user">
     <field property="userName"
             depends="required,minlength,maxlength">
          <arg0 key="userRegistration.userName"/>
         …
     </field>
</form>
```

   The arg0 element passes the key of the message resource, `userRegistration.userName`. Therefore, the error message in this example will display the `userRegistration.userName` message, which is the label for the userName field.

9. Configure the value of the minlength value to 5 and the maxlength to 11. Rules take parameters. This particular rule takes a parameter that tells it what the numeric value of the minimum length is. To set a parameter, you use the var element as follows:

```
<form name="user">
     <field property="userName"
       depends="required,minlength,maxlength,match">
          <arg0 key="userRegistration.userName"/>
          <var>
               <var-name>minlength</var-name>
               <var-value>5</var-value>
          </var>
     </field>
</form>
```

   The minlength rule has a variable called minlength (rule names and variable names do not always match). The var element has two sub-elements that specify the name of the parameter and the value of the parameter. In addition to setting up the minlength rule to 5, you need to set up the maxlength rule to 11 as follows:

```
<var>
<var-name>maxlength</var-name>
     <var-value>11</var-value>
</var>
```

10. Specify that:

    - The value of the rules' minlength variable is the second argument (arg1) to the error message if the minlength rule gets actuated.
    - The maxlength variable is the second argument (arg1) if the maxlength rule is actuated.

    Therefore, instead of getting the argument from the resource bundle, you want to get it from the variable that you just defined. To do this, you must specify another argument. This time, use the arg1 element:

```
<field property="userName"
       depends="required,minlength,maxlength">
     <arg0 key="userRegistration.userName"/>

     <arg1 key="${var:minlength}"
           name="minlength"
           resource="false"/>

     <var>
          <var-name>minlength</var-name>
          <var-value>5</var-value>
     </var>
     <var>
     <var-name>maxlength</var-name>
          <var-value>11</var-value>
     </var>
</field>
</form>
```

    Notice that the code sets the resource attribute equal to false (**`resource="false"`**), which means that the second argument will not be looked up in the resource bundle. Instead, the second argument will use the minlength variable defined in the previous step. To do this, the key attribute is set to ${var:minlength) (**`key="${var:minlength}`**), which states that the value of the second argument is equal to the value of the minlength variable (in kind of a bastardized JSTL expression).

    The name attribute of arg1 states that this second argument is appropriate only for the minlength rule (**`name="minlength")`**. Thus, the second argument will be the value of the minlength variable only if there is a validation problem with the minlength rule. Remember that the property can be associated with many rules because the depends attribute of the field element takes a comma-delimited list of rules to associate with the property. Therefore, the name attribute specifies which rule this argument is used with.

    Now set the first argument for the maxlength rule as follows:

```
<arg1 key="${var:maxlength}"
        name="maxlength"
        resource="false"/>
```

If the above seems like a lot of work, don't fret. Once you set up the framework, using additional rules is easy. The following listing shows the rules for your user registration form so far:

```
<formset>
      <form name="user">
           <field property="userName"
                  depends="required,minlength,maxlength">
                <arg0 key="userRegistration.userName"/>
                <arg1 key="${var:maxlength}"
                        name="maxlength"
                        resource="false"/>
                <arg1 key="${var:minlength}"
                        name="minlength"
                        resource="false"/>
                <var>
                     <var-name>minlength</var-name>
                     <var-value>5</var-value>
                </var>
                <var>
                 <var-name>maxlength</var-name>
                     <var-value>11</var-value>
                </var>
           </field>
      </form>
</formset>
```

Now to get this to run, you will need to comment out your old validate() method so that the validate() method defined by ValidatorForm can execute. As part of this user registration, the code above makes the username a required field, consist of at least 5 characters, and no longer than 11 characters. It does not ensure that the first character is a letter and the remaining characters are alphanumeric, you will do that later after you cover the mask rule.

The last step before you start testing what you have done is to turn on logging. Edit your log4j.properties file (see chapter 1 if you don't know what this is), and add these two entries:

```
#For debugging validator configuration
log4j.logger.org.apache.commons.validator=DEBUG
log4j.logger.org.apache.struts.validator=DEBUG
```

Now read through the log file as your web application gets loaded and the user registration gets submitted. Doing this will help you understand how the Validator framework works and will help you debug it when things go wrong. When you are done with this chapter, you can set the above back to WARN.

Common Validator Rules

Before you start writing your own rules, you should become familiar with Table 4.1, which describes the standard rules included with the framework. As you can see, you get a lot of functionality with very little work.

Table 4.1: Common Validator Rules

Rule Name(s)	Description	Variable(s)
required	The field is required. It must be present for the form to be valid.	None
minlength	The field must have at least the number of characters as the specified minimum length.	The minlength variable specifies the minimum allowed number of characters.
maxlength	The field must have no more characters than the specified maximum length.	The maxlength variable specifies the maximum number of characters allowed.
intrange, floatrange, doublerange	The field must equal a numeric value between the min and max variables.	min variable specifies start of the range max variable specifies end of the range
byte, short, integer, long, float, double	The field must parse to one of these standard Java types (rules names equate to the expected Java type).	None
mask	The field must match the specified regular expression (Perl 5 style).	The mask variable specifies the regular expression.
date	The field must parse to a date. The optional variable datePattern specifies the date pattern (see java.text.SimpleDateFormat in the JavaDocs to learn how to create the date patterns). You can also pass a strict variable equal to false to allow a lenient interpretation of dates (i.e., 05/05/99 = 5/5/99). If you do not specify a date pattern, the short date form for the current locale is used (i.e., DateFormat.SHORT).	datePattern datePatternStrict
creditCard	The field must be a valid credit card number.	None

Let's cover using several combinations of the rules from the table and see the ramifications of doing so. Not all of the rules are covered—just the most useful ones.

Mask Rule

The mask rule is fairly flexible and eliminates the need for writing a lot of custom validation rules. Because the mask rule uses Perl 5-style regular expressions, it is very powerful—assuming, of course, that you know regular expressions well. If you don't, now is a good time to learn. If you have used regular expressions with Perl, Python, or JavaScript, then you are in good shape. Also, if you have used the regular expression package that ships with JDK 1.4 and greater, you will do fine because most of the syntax works the same.

With the mask rule, you are essentially writing your own rule as a Perl 5-style regular expression. Since you are writing your own rule, starting with a blank slate if you will, the default message often does not make sense. You see, having a simple default message—like those with minlength, maxlength, and date—will probably not work as you are writing you own custom rule with a regular expression. Thus, you need to specify your own custom message that the mask rule will use. The message should explain what the regular expression validation routine does so that the end user can understand how to fix the field validation problem.

To demonstrate the mask rule, let's say that the username has to begin with a letter and that it can contain letters, numbers, and underscores.

Before you get started, let’s take a look at the complete work. When you are done, the form element in validation.xml will look like this:

```
<form name="user">
     <field property="userName"
       depends="required,minlength,maxlength,mask">
          <msg name="mask"
               key="userRegistration.userNameMask"/>
          <arg0 key="userRegistration.userName"/>
          <arg1 key="${var:maxlength}"
                name="maxlength" resource="false"/>
          <arg1 key="${var:minlength}"
                name="minlength" resource="false"/>
          <var>
               <var-name>minlength</var-name>
               <var-value>5</var-value>
          </var>
          <var>
          <var-name>maxlength</var-name>
               <var-value>11</var-value>
          </var>
          <var>
<var-name>mask</var-name>
              <var-value>^[a-zA-Z]{1}[a-zA-Z0-9_]*$
              </var-value>
</var>
     </field>
</form>
```

The steps for using the mask rule and defining your own custom message are:

1. Specify that the userName field corresponds to the mask rule:

```
<form name="user">
     <field property="userName"
        depends="required,minlength,maxlength,mask">
```

   This step is similar to the one you performed in the previous example. Now the userName is associated with four rules: required, minlength, maxlength, and mask.

   If you look up the mask rule in the validator-rules.xml file, you will notice that it is associated with a message called errors.invalid as follows:

```
<validator name="mask"
     classname="org.apache.struts.validator.FieldChecks"
         method="validateMask"
         …
        depends=""
        msg="errors.invalid">
        …
```

   Looking at the default message (errors.invalid) in the resource bundle yields:

```
errors.invalid={0} is invalid.
```

   Look at the above error message, and put yourself in the shoes of the end user. The above is not very specific about why the field is invalid. To be more specific, you need to specify a custom message associated with the mask rule. Just like you can specify custom arguments (arg0, arg1) to messages, you can also specify custom messages.

2. Specify an error message key for the mask rule. Because a generic message for a mask rule would not make sense, you need to specify an error key for this rule using the msg element as follows:

```
<form name="user">
     <field property="userName"
        depends="required,minlength,maxlength,mask">

        <msg name="mask"
               key="userRegistration.userNameMask"/>
```

   The msg element takes two attributes. The name attribute specifies which rule you want to associate the message key with. The key element specifies a key for the message in the resource bundler. Thus, the above message will only be used when there is a mask rule validation error.

3. Add the message to the resource bundle:

   ```
   userRegistration.userNameMask={0} must start with a letter and contain
   only letters, numbers, and underscores.
   ```

   Notice how specific this error message is. Essentially, you are describing what the regular expression does in plain English. You can see how the message has to change each time you use the mask rule with other fields. For example, the message for a mask rule applied to a phone number is going to be very different from a message for a mask rule applied to a zip code.

4. Configure the value of the mask variable to a pattern that implements your business rules. The mask variable of the mask rule contains the regular expression that you want to apply to the field. If the field does not match this rule, then it will not validate. Here is the mask for the userName field:

   ```
   <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z]{1}[a-zA-Z0-9_]*$</var-value>
   </var>
   ```

Regular expressions are a necessary weapon in your developer arsenal. For the regular expression neophytes, the ^ specifies that you want to match the start of the line. Thus, ^[a-zA-Z]{1} means that you expect the first character to be an uppercase or lowercase letter. The {1} means you expect one of the items in the list. The expression [a-zA-Z0-9_]* specifies that you expect many (zero or more) letters (a-zA-Z), numbers (0-9), and underscores (_) after the first character. You could change the * to a {10}, which would mean that you expect 10 characters after the first character, and then you would not need the maxlength rule. However, the error messages are clearer if you can split them up among rules (i.e., the error message for maxrule is more specific than the error message for mask).

What if you want to use the same mask on more than one form? For example, the registration form and the login form will both have usernames on them. In the next section, you will look at a way to globally define constants that you can reuse.

It is interesting to note that the rules are applied in the order they are listed. Consider this: if you enter *hi in the username field and the minlength rule is listed first in the value of the depends attribute, you'll see the minlength error. If mask is listed first, you'll see the mask error.

Constants

The Validator framework allows you to define constants that can be used elsewhere in the file. If you have ever used Jakarta Ant, then using global constants is a lot like using Ant properties (but not exactly). First, you define the constant in the global area as follows:

```
<form-validation>
    <global>
        <constant>
            <constant-name>userNameMask</constant-name>
            <constant-value>
               ^[a-zA-Z]{1}[a-zA-Z0-9_]*$
            </constant-value>
        </constant>
    </global>
    ...
```

Then, you refer to the constant using the ${} syntax, just as you would refer to an Ant property:

```
<var>
    <var-name>mask</var-name>
    <var-value>${userNameMask}</var-value>
</var>
```

You can use this constant again and again—for instance, you can use it with the userName field on the Login-Form. In fact, this is the point behind constants.

Note that you can also define constants inside formsets by using the same constant element structure. This is important because constants might be used only for one locale, and formsets can be keyed to a certain local.

```
<formset>
  <constant>
    <constant-name>userNameMask</constant-name>
    <constant-value>^[a-zA-Z]{1}[a-zA-Z0-9_]*$
    </constant-value>
  </constant>
...
```

Working with Dates

Suppose that you want to track the birth dates of your users with the user registration form. To do this, you could use the date rule.

The date rule checks to see if the field is a valid date. The datePattern variable specifies the pattern used to parse the date; if it is not specified, the short form of the date is used. The datePattern format is specified in the JavaDocs for java.text.SimpleDateFormat because the underlying implementation uses java.text.SimpleDateFormat.

Instead of datePattern, you could specify a datePatternStrict variable. The datePatternStrict variable will ensure that 5-29-70 does not work; when the pattern MM-dd-yyyy is used, only 05-29-1970 works (notice the 0 before the 5).

The following code ensures that users enter dates using the pattern MM-dd-yyyy:

```
<field property="birthDate"
            depends="date">
     <arg0 key="userRegistration.birthDate" />
    <var>
       <var-name>datePattern</var-name>
       <var-value>MM-dd-yyyy</var-value>
    </var>
</field>
```

Thus, 05-29-1970 would be a valid date but 99-29-1970 would be an invalid date. This code assumes that you have added a string property to your userRegistration form bean.

To make 5-29-1970 an invalid date, you would use the strict form as follows:

```
<field property="birthDate"
            depends="date">
     <arg0 key="userRegistration.birthDate" />
    <var>
       <var-name>datePatternStrict</var-name>
       <var-value>MM-dd-yyyy</var-value>
    </var>
</field>
```

Notice with both date validations, the user could decide to enter no date at all. Hence, the way you coded the code above implies that birth date is an optional field. Essentially, this rule determines the date only if it is present; otherwise, it is happy to validate a blank field. To make the date required, you would add the required rule to the front of the depends attribute.

Working with E-Mail and Credit Cards

Working with e-mail and credit card validation rules is quite easy because neither rule takes arguments. The email and credit card validation rules were converted from working routines that were used by JavaScript and Perl developers, respectively. Ted Husted translated these routines to Java and added them to the Validator framework. Here's how you use the email validation rule in the field declaration in validation.xml:

```
<field property="email"
            depends="required,email">
     <arg0 key="userRegistration.email" />
</field>
```

This code maps the email rule to the email property in your ActionForm.

Note: You can find the implementation for all the rules that ship with the Validator framework in org.apache.struts.validator.FieldChecks and org.apache.commons.validator.GenericValidator.

Putting It All Together

For completeness, add all of the validation to all of the fields that need it. Validate the phone numbers and fax number fields if present. Make the password fields required. Make the firstName and lastName fields required. Here is the complete validation.xml file:

```
<form-validation>

    <global>

        <!-- An example global constant
        <constant>
            <constant-name>postalCode</constant-name>
            <constant-value>^\d{5}\d*$</constant-value>
        </constant>
        end example-->

        <constant>
            <constant-name>userNameMask</constant-name>
            <constant-value>
               ^[a-zA-Z]{1}[a-zA-Z0-9_]*$
            </constant-value>
        </constant>

    </global>

    <formset>
      <constant>
      <constant-name>countryCode</constant-name>
      <constant-value>[0-9]?</constant-value>
      </constant>
      <constant>
      <constant-name>areaCode</constant-name>
      <constant-value>
                [ |(]*[0-9]{3}[ |)]*
                </constant-value>
      </constant>
      <constant>
      <constant-name>phone</constant-name>
      <constant-value>
                 [0-9]{3}[-| ]*[0-9]{4}
                </constant-value>
      </constant>

          <form name="user">
               <field property="userName"
                  depends="required,minlength,maxlength,mask">
                    <msg name="mask"
                key="userRegistration.userNameMask" />
           <arg0 key="userRegistration.userName" />
            <arg1 key="${var:maxlength}"
                  name="maxlength" resource="false" />
            <arg1 key="${var:minlength}"
                  name="minlength" resource="false" />
            <var>
                 <var-name>minlength</var-name>
                 <var-value>5</var-value>
            </var>
            <var>
            <var-name>maxlength</var-name>
                 <var-value>11</var-value>
            </var>
            <var>
                <var-name>mask</var-name>
                <var-value>${userNameMask}</var-value>
            </var>
       </field>
       <field property="birthDate"
                   depends="date">
            <arg0 key="userRegistration.birthDate" />
            <var>
              <var-name>datePatternStrict</var-name>
              <var-value>MM-dd-yyyy</var-value>
            </var>
       </field>
       <field property="email"
                   depends="required,email">
            <arg0 key="userRegistration.email" />
       </field>

       <field property="firstName"
                   depends="required">
            <arg0 key="userRegistration.firstName" />
       </field>
       <field property="lastName"
                   depends="required">
            <arg0 key="userRegistration.lastName" />
       </field>
       <field property="passwordCheck"
                   depends="required">
            <arg0 key="Password Check"
                  resource="false"/>
       </field>
       <field property="password"
                   depends="required">
            <arg0 key="userRegistration.password" />
       </field>
       <field property="phone"
```

```
                                   depends="mask">
                            <arg0 key="userRegistration.phone" />
                            <var>
                              <var-name>mask</var-name>
                              <var-value>
                                ${countryCode}${areaCode}${phone}
                              </var-value>
                            </var>
                       </field>
                       <field property="fax"
                                   depends="mask">
                            <arg0 key="userRegistration.fax" />
                            <var>
        <var-name>mask</var-name>
                              <var-value>
                                ${countryCode}${areaCode}${phone}
                              </var-value>
                            </var>
                       </field>
                       <field property="dynamicProps(bar)"
                                   depends="required">
                           <arg0
                            key="userRegistration.dynamicProps.bar" />
                       </field>
                  </form>
            </formset>

        </form-validation>
```

Don't forget to add the birthDate field to the UserRegistrationForm and userRegistration.jsp, and add the label to the resource bundle.

Using More than One Constant to Create Regular Expressions

Creating regular expressions can quickly become unwieldy. One strategy is to use constants to divide the job into smaller jobs. Below you define three regular expression constants: countryCode, areaCode, and phone as follows:

```
...
<formset>
  <constant>
  <constant-name>countryCode</constant-name>
  <constant-value>[0-9]?</constant-value>
  </constant>
  <constant>
  <constant-name>areaCode</constant-name>
  <constant-value>
            [ |(]*[0-9]{3}[ |)]*
            </constant-value>
  </constant>
  <constant>
  <constant-name>phone</constant-name>
  <constant-value>
             [0-9]{3}[-| ]*[0-9]{4}
            </constant-value>
  </constant>
```

The **countryCode** regular expression constant checks to see if the country code is present but allows it to be missing. (If you are in the US, US phone numbers do not need a country code.) The **areaCode** regular expression constant looks for a number as follows: “(520)” or “609”. The **phone** regular expression constant looks for a number like “333-4444” or just “333 4444”.

Note: Notice that the way you are validating the phone number is very US-centric; therefore, you put the constants as part of the formset instead of globals because you can internationalize formsets in the validation.xml file.

The phone field and the fax field use this regular expression constants by stringing the constants together (**${countryCode}${areaCode}${phone}**) as follows:

```
<form name="user">
            ...
            <field property="phone"
                        depends="mask">
                 <arg0 key="userRegistration.phone" />
                 <var>
                   <var-name>mask</var-name>
                   <var-value>
                     ${countryCode}${areaCode}${phone}
                   </var-value>
                 </var>
            </field>
            <field property="fax"
                        depends="mask">
                 <arg0 key="userRegistration.fax" />
                 <var>
<var-name>mask</var-name>
                   <var-value>
                     ${countryCode}${areaCode}${phone}
                   </var-value>
                 </var>
            </field>
    </form>
 </formset>

</form-validation>
```

Note: For a quick overview of regular expressions, go to: http://etext.lib.virginia.edu/helpsheets/regex.html

The nice thing about constants is that you can divide and conquer a regular expression by breaking it down into smaller regular expressions. Later, if you want to enhance the way part of the regular expression works, you can do so more easily.

Working with Dynamic Properties (Mapped Back Properties)

In addition to validating static properties, you can validate dynamic properties (mapped back properties) as follows:

```
<field property="dynamicProps(bar)"
            depends="required">
    <arg0
     key="userRegistration.dynamicProps.bar" />
</field>
```

Recall that you have a mapped back property in your ActionForm. The code above ensures that your mapped back property has a value called “bar”.

Specifying Hard-Coded Arguments

Notice that the passwordCheck field uses a hard-coded label instead of the one from the resource bundle as follows:

```
<field property="passwordCheck"
            depends="required">
     <arg0 key="Password Check"
           resource="false"/>
</field>
```

In the code above, arg0 uses the key as a literal string instead of looking up the arg0 (which is the label of the field) in the resource bundle. This may at first seem somewhat evil, but you have to keep in mind that formsets are also internationalized, which you would use if the validation rule setup is different for different locales.

i18n for Validation

Validation requirements often vary quite a bit based on locale (like how you validate a phone number). For example, if you design a web site for users from France, Canada, and the US, the forms these users fill out might have unique validation needs. In this case, you need to create formset for each language and country combination. If you do not specify a locale, it becomes the default locale. You specify the locale with the language attribute of the formset as follows:

```
<formset country="fr" language="fr" >
</formset>

<formset country="ca" language="fr" >
</formset>
```

Extending Validate and Custom Validation in the Validate() Method of an ActionForm

There are two main types of form validation. The first type is per field—in other words, the validation takes into account only one field at a time. The Validator framework excels with this type of form validation.

The other type of form validation is the relationship between two fields. For example, let's say that your product form has a price and a list price field. The rule is that the list price should never be less than the price. You can add this validation in the ActionForm's validate method, but you could argue that this type of validation is more of a business rule than true validation.

Another example comes to mind. Let's say that you need the end user to enter two passwords. One is the password itself, and the second is to ensure that users entered what they thought they did. To write this, you could choose to override the validate() method of the ValidatorForm class.

The ValidatorForm is needed to integrate the Validator framework to Struts. The ValidatorForm subclasses ActionForm and overrides the validate() method to use the framework. You can take this a step further by overriding the validate() method of ValidatorForm and extending the functionality of the ValidatorForm to provide your own custom validation by following these steps:

1. Override the validate method.
2. Call the superclass validate method.
3. Save a reference to the errors (if any) that the ValidatorForm superclass produces.
4. Perform additional error checking; add errors.
5. Return the errors.

To add password support to your ActionForm, add the following code:

```
public class UserRegistrationForm extends ValidatorForm {
private String password;
private String passwordCheck;
      ...

public ActionErrors validate(
ActionMapping mapping,
HttpServletRequest request) {

log.trace("validate");
ActionErrors errors =
                         super.validate(mapping, request);
if (errors==null) errors = new ActionErrors();
if (!password.equals(passwordCheck)) {
errors.add(
"password",
new ActionError (
                      "userRegistration.password.nomatch"));
}
return errors;

}
```

You can use this approach anytime you want to add validation that the Validator framework does not yet provide. Now, this is one way to extend the Validator framework without writing your own custom rules. However, you could just write your own validation rule; in the next section, you'll learn how.

Writing Your Own Rules

Let's begin with a simple rule for validating a zip code. To create a validation rule, you follow these steps:

1. Create a Java class. This class does not have to subclass any special class.
2. Add a static() method to the Java class that implements the validation routine. The validation routine gets passed a bean, the ValidatorAction, the Field, and the servlet request as follows:

```
public class ValidationRules {

public static boolean validateZip(
Object bean,
ValidatorAction va,
Field field,
ActionErrors errors,
HttpServletRequest request){
...
```

The ValidatorAction contains information for dynamically instantiating and running the validation method. The ValidatorAction is the rule. This is the object representation of the XML validator element that you will define in step 3.

The Field contains the list of rules (pluggable validators such as ValidatorAction), the associated form property, message arguments, messages, and variables used to perform the validations and generate error messages. This is the object representation of the field element you've been using all along.

Inside the validator method, you need to get the string value of the current field. Then, you must see if that string value is a valid zip code. If the string value is not a valid zip code, you need to add an error to the ActionErrors as follows (read the comments):

```
package strutsTutorial.validation;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

...
import org.apache.commons.validator.Field;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.validator.Resources;

/**
 * @author Richard Hightower
 * ArcMind Inc. http://www.arc-mind.com
 */
public class ValidationRules {
      ...
/* Accepts zip codes like 85710 */
private static final String ZIP_REGEX = "[0-9]{5}";

public static boolean validateZip(
Object bean,
ValidatorAction va,
Field field,
ActionErrors errors,
HttpServletRequest request){


 /* Create the correct mask */
 Pattern mask  = Pattern.compile(ZIP_REGEX);

       /* Get the string value of the current field */
 String zipField = ValidatorUtil.
getValueAsString(bean, field.getProperty());

 /* Check to see if the value is a zip code */
 Matcher matcher = mask.matcher(zipField);

 if (!matcher.matches()){
errors.add(field.getKey(),
Resources.getActionError(request,va,field));
return false;

 }

 return true;
}
```

This code checks to see if the string value of the field matches the regular expression pattern of `ZIP_REGEX` (`[0-9]{5}`). The regular expression states the zip code consists of 5 numbers.

Note: You may have noticed that this chapter uses the regular expression library that ships with JDK 1.4 and higher. If you are not familiar with this library, go to http://java.sun.com/docs/books/tutorial/extra/regex/ for more details.

3. Create an entry in the validator-rules.xml file that associates the static() method to a rule. The validator-rules.xml file serves as a deployment descriptor for validation rules. This is the entry you need to add:

   ```
   <validator name="zip"
      classname="strutsTutorial.validation.ValidationRules"
      method="validateZip"
      methodParams="java.lang.Object,
              org.apache.commons.validator.ValidatorAction,
               org.apache.commons.validator.Field,
               org.apache.struts.action.ActionErrors,
               javax.servlet.http.HttpServletRequest"
              msg="errors.zip">
   </validator>
   ```

   The validator element defines the rule. The name attribute specifies the name of the new rule. The classname specifies the name of the new class you just created that contains the static() method validateZip (i.e., the rule handler). The method attribute defines the method that will be called (i.e., validateZip). The methodParams specifies the arguments that will be passed to the validateZip method. Lastly, the msg attribute specifies the resource key of the message that will be used from the resource bundle if a validation failure occurs.

That's it. Now, to actually use the validator rule, you need to do the following:

1. Create a field entry in the user form in the validator.xml file that associates the zip property of the userRegistration with the new zip validator rule.

   ```
   <field property="zip"
        depends="required,zip">
        <arg0 key="userRegistration.zip" />
   </field>
   ```

2. Add the errors.zip message that corresponds to this rule to the resource bundle (application.properties) file:

   ```
   errors.zip={0} must be a valid zip code.
   ```

Warning! You need to add a zip property to the ActionForm if you want this tutorial to work correctly.

Defining Rule Variables

Now, what you have added is not very complicated. In fact, you could have just used the match pattern like you did with userName earlier. What if this rule was more complicated? Let's say that the rule optionally supported plus-four zip code extensions (i.e., 85710-1111). Assume that this was a general-purpose rule. Some developers using the rule want the plus-four extension to be optional (i.e., if it's there, it's okay; but if it's missing, that's okay too). Other developers want the plus-four extension to be required. In order to do this, you need to define two rule variables called plus4Required and plus4Optional. You need to get the variables by using the field() method getVarValue as follows (read the code comments):

```
public class ValidationRules {

/* Accepts zip codes like 85710 */
private static final String ZIP_REGEX = "[0-9]{5}";

/* Accepts zip code plus 4 extensions like "-1119" or "
        1119" */
private static final String PLUS4_REQUIRED_REGEX =
                                        "[ |-]{1}[0-9]{4}";

/* Optionally accepts a plus 4 */
private static final String PLUS4_OPTIONAL_REGEX =
                                      "([ |-]{1}[0-9]{4})?";

public static boolean validateZipPlus4(
Object bean,
ValidatorAction va,
Field field,
ActionErrors errors,
HttpServletRequest
                                                 request){


 /* Get the plus4 options from the
              validation xml file */
 String sPlus4Required =
                   field.getVarValue("plus4Required");
 boolean plus4Required =
                      "true".equals(sPlus4Required);

 String sPlus4Optional =
                   field.getVarValue("plus4Optional");

 boolean plus4Optional =
                   "true".equals(sPlus4Optional);

 /* Create the correct mask */
 Pattern mask =  null;
```

```
         if (plus4Required){
         mask =  Pattern.compile(
                             ZIP_REGEX + PLUS4_REQUIRED_REGEX);
         } else if (plus4Optional){
        mask = Pattern.compile(
                             ZIP_REGEX + PLUS4_OPTIONAL_REGEX);
         } else if (plus4Required && plus4Optional){
         throw new IllegalStateException(
                             "Plus 4 is either optional or required");
         }
         else {
        mask = Pattern.compile(ZIP_REGEX);
         }

         /* Get the string value of
                            the current field */
         String zipField = ValidatorUtil.
                         getValueAsString(bean, field.getProperty());

         /* Check to see if the value is a zip code */
               Matcher matcher = mask.matcher(zipField);

               if (!matcher.matches()){
        errors.add(field.getKey(),
        Resources.getActionError(request,va,field));
           return false;

               }

              return true;
        }
```

Warning! For simplicity, the code above does not keep a cache of the compiled regular expressions. You should. Use lazy initialization to compile the regular expressions one time. Keep static copies of precompiled regular expressions.

The following code would declare the zipPlus4 rule available in the validator-rules.xml file.

```
<validator name="zipPlus4"
   classname="strutsTutorial.validation.ValidationRules"
           method="validateZipPlus4"
           methodParams="java.lang.Object,
           org.apache.commons.validator.ValidatorAction,
           org.apache.commons.validator.Field,
           org.apache.struts.action.ActionErrors,
           javax.servlet.http.HttpServletRequest"
           msg="errors.zip">
</validator>
```

To optionally allow plus-four extensions (85710 and 85710-1111), you would use the new rule as follows:

```
<field property="zip"
     depends="required,zipPlus4">
    <arg0 key="userRegistration.zip" />
    <var>
        <var-name>plus4Optional</var-name>
        <var-value>true</var-value>
    </var>

</field>
```

To require plus-four extentions (85710-1111 only), you would use the new rule as follows:

```
<field property="zip"
     depends="required,zipPlus4">
    <arg0 key="userRegistration.zip" />
    <var>
        <var-name>plus4Required</var-name>
        <var-value>true</var-value>
    </var>

</field>
```

To require only 5 digits (85710 only), you would use the new rule as follows:

```
<field property="zip"
     depends="required,zipPlus4">
    <arg0 key="userRegistration.zip" />
</field>
```

Warning! Remember to use strings and not other primitive types for properties. You should use strings for form property types when you are doing any type of validation with any type of text field. The problem with using other types is that Struts does conversion. Therefore, if you have an integer property, Struts will convert the incoming request parameter into an integer. If the incoming type is invalid, Struts will convert it to a 0, and it is likely that your validation rule will either not run or not run correctly— the users will just see the field as a 0 instead of the text that they typed in the text field.

Working with DynaActionForms

Working with DynaActionForms is quite easy. When you set up the dynamic form in the Struts configuration file, use the DynaValidatorForm class instead of the DynaActionForm class as follows (in struts-config.xml):

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


    <form-property name="userName"
                   type="java.lang.String" />
    <form-property name="email"
                   type="java.lang.String" />
    <form-property name="password"
                   type="java.lang.String" />
    <form-property name="passwordCheck"
                   type="java.lang.String" />
    <form-property name="firstName"
                   type="java.lang.String" />
    <form-property name="lastName"
                   type="java.lang.String" />
    <form-property name="phone"
                   type="java.lang.String"
                   initial="(520) " />
    <form-property name="fax" type="java.lang.String" />
    <form-property name="page"
                   type="java.lang.Integer" />
    <form-property name="birthDate"
                   type="java.lang.String" />
    <form-property name="zip"
                   type="java.lang.String" />
    <form-property name="dynamicProps"
                   type="java.util.HashMap" />

</form-bean>
```

The validate() method of the DynaValidatorForm provides the necessary hooks into the Validator framework to tie the validation rules you define in validation.xml to the fields you define in your DynaActionForm. The Validator framework and DynaActionForms have a special relationship. DynaActionForms typically rely on the Validator framework to perform validation. All of the rules you configured and wrote would work the same way with a DynaValidatorForm—you simply have to configure the form-property for each field.

Notice that you added three additional fields (birthDate, zip, and dynamicProps) to work with the validation rules that you already wrote. Nothing needed to change in the validation.xml or validator-rules.xml.

These fields must also be added to userRegistrationDyn.jsp. In addition, make sure the hidden property 'page' is added to the jsp (or remove the page form-property from the form bean).

Working with Client-Side JavaScript

It's very useful to validate some fields on the client with JavaScript: it saves the user a round trip to the server. The Validator framework provides a mechanism to generate JavaScript code. The validator-rules.xml file can have JavaScript associated with each validator. The JavaScript is the client-side equivalent of the validator rule, if applicable. To use the JavaScript validator routines, you need to do the following:

1. Add the html:javascript tag to your input JSP page as follows:

   ```
   <head>
   <title>User Registration</title>
   <html:javascript formName="user"/>
   </head>
   ```

   Notice that you need to specify the formName. The formName will be used by the tag to generate the JavaScript validation routine dynamically. If your formName is user (set up in the Struts configuration file), the html:javascript will generate a JavaScript function called validateUser. The validateUser will validate the form and return true if the validation succeeded; otherwise it will return false.

   Much of the JavaScript code generated by html:javascript is static. You can optimize what gets sent to the browser by setting the staticJavaScript attribute to false and including a JavaScript file that contains all of the static JavaScript. This allows the browser to cache the JavaScript routines instead of downloading them for each form in your web application. The JavaScript routines are quite large. In your input JSP, create an HTML script tag and add a src attribute that points to the JavaScript file that contains all of the validation routines. You can even generate these routines with the html:javascript tag.

2. In order to tie the validateUser to the form submission, you need to add a JavaScript event handler to the form using the onsubmit attribute:
   ```
   <html:form action="userRegistration"
              onsubmit="return validateUser (this)">
   ```

   If validateUser returns false, then the form submission will not occur. The validateUser function will display a validation error message and cause the form not to be submitted. Thus, the end user will need to handle all of the validation errors before they can submit the form to the server.

   Some validator rules are too complex to easily express in JavaScript. It is up to the rules validator developer to write the JavaScript equivalent. If there is no JavaScript equivalent or if the end user has JavaScript disabled, the Java validator will run on the server instead.

   A common mistake is to forget to use the return statement in the onsubmit. If you forget the return statement, then both of the validators (JavaScript and Java) will execute, and you will not save the roundtrip to the server.

Canceling the JavaScript Validation

Adding a cancel to your form can cause problems if not handled correctly. The end user clicks on the Cancel button, but instead of canceling the operation as expected, it runs the JavaScript validation routines. The generated validate[FormName] function checks to see the bCancel variable was set. If it was set, it does not perform the JavaScript validation routine as follows:

```
var bCancel = false;

function validateUserAll(form) {
    if (bCancel)
       return true;
    else
       return validateMaxLength(form)
       && validateRequired(form) ...
}
```

Thus, you need to set the bCancel variable when the Cancel button is clicked as follows:

```
<html:cancel onclick="bCancel=true"/>
```

Recall that when the Cancel button is clicked, the request parameter org.apache.struts.action.CANCEL is sent (Globals.CANCEL_KEY). You can check to see if the form has been cancelled using the Action’s isCancelled() method in the Action’s execute() method as follows:

```
if (this.isCancelled(request)){
    log.debug("this has been cancelled");
    return mapping.findForward("welcome");
}
```

Using a Workflow Form

At times you will want to develop a wizard-style workflow as you did in chapter 3. You can do this with the Validator framework by using the page attribute of the field element. To make the page attribute work, the ValidatorForm includes a page property. You must include the current page in your HTML form (your JSP input page) with a hidden field that corresponds to the page property. The field described with the field's element will only be evaluated if the field element's "page" attribute is less than or equal to the page property of the ActionForm.

Since the ValidatorForm already defines a page property, comment out the page property that you added for the static ActionForm—UserRegistrationForm. (Leave the page property for the DynaActionForm.)

Modify the validation.xml file by adding page attributes for the fields in the form. You need to add page attributes to each of the fields that correlate to the two steps in the wizard. For fields that you want to be validated on the first form in the wizard submission, add a page='1' attribute. For fields that you want to be validated on the second step of the wizard, add a page='2' attribute. The entry for this new workflow user registration (validation.xml) is as follows:

```
<form name="user">
     <field property="userName" page="1"
        depends="required,minlength,maxlength,mask">
          <msg name="mask"
          key="userRegistration.userNameMask" />
          <arg0 key="userRegistration.userName" />
          <arg1 key="${var:maxlength}"
            name="maxlength" resource="false" />
          <arg1 key="${var:minlength}"
            name="minlength" resource="false" />
          <var>
               <var-name>minlength</var-name>
               <var-value>5</var-value>
          </var>
          <var>
          <var-name>maxlength</var-name>
               <var-value>11</var-value>
          </var>
          <var>
<var-name>mask</var-name>
              <var-value>${userNameMask}</var-value>
</var>
     </field>
     <field property="email"
            page="1"
            depends="required,email">
          <arg0 key="userRegistration.email" />
     </field>
     <field property="passwordCheck"
            page="1"
           depends="required">
          <arg0 key="Password Check"
                resource="false" />
     </field>
     <field property="password"
            page="1"
            depends="required">
          <arg0 key="userRegistration.password" />
     </field>

     <field property="birthDate"
            page="2"
            depends="date">
          <arg0 key="userRegistration.birthDate" />
         <var>
            <var-name>datePatternStrict</var-name>
            <var-value>MM-dd-yyyy</var-value>
         </var>
     </field>
     <field property="firstName"
            page="2"
            depends="required">
          <arg0 key="userRegistration.firstName" />
     </field>
     <field property="lastName"
            page="2"
            depends="required">
          <arg0 key="userRegistration.lastName" />
     </field>
     <field property="phone"
            page="2"
            depends="mask">
          <arg0 key="userRegistration.phone" />
          <var>
<var-name>mask</var-name>
             <var-value>
      ${countryCode}${areaCode}${phone}
             </var-value>
</var>
     </field>
     <field property="fax"
            page="2"
            depends="mask">
          <arg0 key="userRegistration.fax" />
          <var>
<var-name>mask</var-name>
           <var-value>
      ${countryCode}${areaCode}${phone}
           </var-value>
</var>
     </field>
```

```
            <field property="dynamicProps(bar)"
                   page="2"
                   depends="required">
                 <arg0
               key="userRegistration.dynamicProps.bar" />
            </field>


        <field property="zip"
           page="2"
           depends="required,zipPlus4">
           <arg0 key="userRegistration.zip" />
                  <var>
        <var-name>plus4Optional</var-name>
                  <var-value>true</var-value>
        </var>

        </field>

        </form>
```

Notice how the page attributes for the fields email, userName, password, and passwordCheck are set to 1, and the page attributes for the rest of the fields are set to 2. This means that email, userName, password, and password-Check will be validated during the first step. The second step will validate the rest of the fields and the fields that were validated on the first step (cumulative).

Summary

This chapter discussed the core functionality of the Validator framework and provided step-by-step examples for using the Validator framework. The more useful validator rules were covered, as well as advanced topics like how to create and validate a wizard-style application. You also learned how to create your own validator rule in Java, and how to perform validation in both the validate() method of an ActionForm and with the Validator framework.

Chapter 5

Actions

Getting things done with actions

Struts developers spend most their time creating actions (instances of the org.apache.struts.action.Action class). Actions are the glue between the Model and View of you application. This chapter will cover all of the standard actions and the helper methods of the Action class.

Actions should communicate with the Model, invoke business rules, and return Model objects to the View to display. Actions also prepare error messages to display in the View.

You can also create utility actions. Utility actions can be wired to other actions using action chaining. This allows you to extend the behavior of an action without changing an action or having deeply nested class hierarchies of actions.

Actions have a life cycle similar to servlets. Actions are like servlets in that they're multithreaded. However, you must use caution when working with member variables of an action as they are not by default thread safe. You can think of an action as a singleton in the system.

Struts has a servlet called the ActionServlet. The ActionServlet inspects the incoming request and delegates the request to an action based on the incoming request path. The object that relates the action to the incoming request path is the action mapping.

Your actions are part of the Struts Controller. You have to extend actions to add behaviors to your Struts application. You can define a collection of these actions to define the behaviors of your web application.

Creating an Action

Below is the action from the tutorial in Chapter 1. Take a look at the DisplayAllUsers action. To create an action, you must follow these steps:

1. Create an action by subclassing org.apache.struts.action.Action as follows:

   ```
   package strutsTutorial;

   import org.apache.struts.action.Action;
   import org.apache.struts.action.ActionForm;
   import org.apache.struts.action.ActionForward;
   import org.apache.struts.action.ActionMapping;

   ...


   import strutsTutorial.dao.DAOFactory;
   import strutsTutorial.dao.UserDAO;

   public class DisplayAllUsersAction extends Action {

   ...
   }
   ```

   There is nothing special here; simply subclass Action and you're good to go.

2. Override the execute() method.

   The execute() method is where you define the behavior of the current action. It is a collection of actions that define the behaviors of your application.

   There are three primary things that you want to do with the execute() method:

   - Work with the Model to perform business logic
   - Report any errors to the View
   - Select the next View of the application

   Granted, you could add validate preconditions to the list above, but you can validate preconditions in better ways by using a custom request processor or creating a servlet filter. You will cover custom request processors in Chapter 10.

The most commonly used execute implementation that the Action class defines it is HTTP-centric. The signature of this execute() method is as follows:

```
public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws IOException, ServletException
```

Notice that the execute() method takes four arguments. The action mapping is passed to the execute() method, and it contains everything configurable in the action element of the Struts config file. The ActionForm represents the incoming request parameters. The HttpServletRequest references the current request object. It is unlikely that you will use the HttpServletResponse object, but it is passed to you just in case.

As you can see, Struts does not try to hide the Servlet API from the developer. This is why Struts is called a “lightweight framework.” It provides structure and reusable workflows for developers, but it does not try to mask that you are dealing with a Java web application. One of the jobs of an action is to translate data and events from the web application to the Model without passing Servlet API objects to the Model. The Model should be completely “agnostic,” with regards to the Servlet API.

Notice that the execute() method returns an ActionForward. The ActionForward is the logical next View or really the logical next step as it can be mapped to any resource including another action. It is helpful when you get started using Struts to think of the ActionForward as the output View. The ActionForward is used by the Struts RequestProcessor to determine the next step to handle the request (e.g., forward to a JSP or an action). The following code shows a sample execute() method:

```
public class DisplayAllUsersAction extends Action {

     private static Log log = LogFactory.getLog(DisplayAllUsersAc-
tion.class);

     public ActionForward execute(
          ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
          throws Exception {

          log.trace("DisplayAllUsersAction");

          DataSource dataSource =
                    getDataSource(request, "userDB");
          Connection conn = dataSource.getConnection();

          List list =null;
          try{
                /* Create UserDAO */
                UserDAO dao = DAOFactory.createUserDAO(conn);
                list = dao.listUsers();
           }finally{
                conn.close();
           }


           if (list.size() > 0){
                request.setAttribute("users", list);
           }

           return mapping.findForward("success");
      }

}
```

Most of your actions will perform similar steps. For example, actions always cast the ActionForm to the current ActionForm subclass. Then, the action interacts with Model objects. Lastly, the action will return the next View or step with the action mapping findForward() method.

3. Configure the action in the Struts config file.

   Now that you have created the action, you need to configure the action in the Struts config file as follows:

```
<action
      path="/displayAllUsers"
      type="strutsTutorial.DisplayAllUsersAction">

      <forward name="success"
               path="/userRegistrationList.jsp"/>
</action>
```

   As you can see, you configure a Struts action by using the <action> element. The action handler is specified with the type attribute. The action handler is the class that you just wrote. The path attribute specifies the incoming request path that this action will handle. You can reuse the action handler with multiple <action> elements; thus, the same action handler can handle multiple request paths. In fact, if you recall from the ActionForm chapter, you used this technique to implement a multistep wizard.

Table 5.1 Action Attributes

Attribute	Description
attribute	The attribute specifies the name of the ActionForm that is mapped to a given scope. If you do not specify an attribute, the name attribute becomes the value of the attribute. Thus, if the attribute is not specified, the attribute defaults to the name of the ActionForm.
className	The className attribute is used to specify custom config object. We will cover these in Chapter 10 in more detail. For now, you can replace any config object with a custom config object. The custom config object can have additional properties.
include	You can use the include attribute to specify a JSP that will handle this request. The JSP is the only handler of this request. Using this should be the exception, not the norm. This attribute is mutually exclusive with the forward attribute and the type attribute, meaning that only one of the attributes can be set.
input	The input attribute is used to specify the input View for an action. The input attribute usually specifies a JSP page that has an HTML form that submits to this action. If there are any problems with form validation, then control of the application will forward back to this input View. This attribute is optional, but if it's not included, an error occurs when Struts attempts to display validation errors.
name	The name attribute identifies the ActionForm that is related to this action. The html:form tag uses this name to pre-populate the html form. The request processor uses this form name to create and populate an ActionForm to pass to the execute() method of the action handler.
path	The path attribute represents the incoming request path that this action maps to.
parameter	The parameter attribute is similar in concept to servlet init-parameters. Its use is application-specific. You can think of it as a general-purpose init-parameter.
roles	The roles attribute is used to specify J2EE security roles that are allowed to access this action. If this attribute is present, then only those roles specified in the comma-delimited list can use this action.
type	The type attribute specifies the action handler for this action. You must specify a fully qualified class name. The type attribute is mutually exclusive with the include and forward attribute.
scope	The scope specifies in which scope (request or session) the ActionForm will be placed. The scope attribute defaults to session.
Unknown	The unknown attribute is used to specify a default action mapping to handle unknown request paths. Obviously, you can only have one of these for each Struts config file.
validate	The validate attribute is used when you want to validate the ActionForm. This is the case when you are handling form submissions. By specifying validate equals=true, you are causing the request processor to call the ActionForm.validate() method to see if there are any field/form validation errors. The execute() method of the action will not be called if form validation fails and validate equals true. Instead, control of the application would return to the input View. You typically set the validate attribute to false, if you want to pre-populate an HTML form (e.g., an action that loads a form with form data from a database).

ForwardAction

It’s generally a bad idea to link directly to a JSP page from inside another JSP page. At times, however, all you really need is just a plain link; you don't want (or need) an action to execute first. Perhaps there are no objects from the domain that need to be mapped into scope in order for the View to display. Alternatively, maybe the page is very simple. In these cases, a better approach is to use the ForwardAction. (These are exceptions to the rule.)

The ForwardAction acts as a bridge from the current View (JSP) and the pages it links to. It uses the RequestDispatcher to forward to a specified web resource. It's the glue that allows you to link to an action instead of directly to a JSP.

Later, if you need to, you can change the action mapping in the Struts configuration file so that every page that linked to that action will link to the new action. In addition, you can change the action to a custom one that you write instead of using the ForwardAction that Struts provides.

To use the ForwardAction, follow these steps:

1. Using the html:link tag with the action attribute, add a link to the JSP page that points to the action.
2. Create an action mapping in the Struts configuration file that uses the ForwardAction with the parameter attribute to specify the JSP path.

Let's say you have a JSP page that has a direct link to another JSP page:

```
<html:link page="/index.jsp">Home</html:link>
```

You have recently converted to the MVC/Model 2 architecture to simplify your design and to encourage a “divide and conquer” spirit. You want to change the html:link tag to link to an action. Because you already have a link, simply change it as follows:

```
<html:link action="home">Home</html:link>
```

All you do is remove the page attribute and add an action attribute that points to the home action. Now, you have to create the home action mapping. (The link in the previous code snippet would be expanded to a URL like http://localhost:8080/strutsTutorial/home.do.)

To add an action mapping to the home action that you referenced in your html:link tag, use this code:

```
<action
    path="/home"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/index.jsp"
    />
```

The ForwardAction uses the parameter attribute so that you can specify the path. Notice the parameter is set to / index.jsp, which was what the page attribute of the html:link tag was originally set to. Thus, the parameter attribute indicates to where you want the ForwardAction to forward.

The Forward Attribute vs. ForwardAction

For better or for worse, ForwardActions get used quite a bit—so much so that the Struts configuration file includes support for them. Thus, rather than doing this:

```
<action
    path="/home"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/index.jsp"
    />
```

you can do this instead:

```
<action
    path="/home"
    forward="/index.jsp"
    />
```

These two mappings are functionally equivalent. However, those in the know prefer the shorter of the two (use the forward attribute).

IncludeAction

The IncludeAction is similar to the ForwardAction but is not used as much. You could use the include action as follows:

```
<action
    path="/legacy"
    type="org.apache.struts.actions.IncludeAction"
    parameter="/legacy/roar"
    input="/form/userForm2.jsp"
    name="userForm"
    validate="true"
    scope="request"
    />
```

This shorter form uses the include attribute:

```
<action
    path="/legacy"
    include="/legacy/roar"
    input="/form/userForm2.jsp"
    name="userForm"
    parameter="/legacy/roar"
    validate="true"
    scope="request"
    />
```

So, what is the difference between the IncludeAction and the ForwardAction? The difference is that you need to use the IncludeAction only if the action is going to be included by another action or JSP.

Therefore, if you have code in your JSP that looks like this:

```
<jsp:include page="/someWebApp/someModule/someAction.do"/>
```

The action could not use a ForwardAction because it would forward control to the new action rather than including its output within the output of the JSP—or throw a nasty IllegalStateException if the output buffer was already committed.

DispatchAction

Oftentimes actions seem to be too plentiful and too small. It would be nice to group related actions into one class. What many developers do not realize is that having too many small classes is as much of a cohesion problem as having classes that are too large. Therefore, it is often the case that you would like to group related actions into one class (assuming that the actions are involved in a similar set of roles and goals).

For example, let's say that a group of related actions all work on the same set of objects in the user session (HttpServletSession)—a shopping cart, for example. Another example is a group of actions that are all involved in the same use case. Yet another example is a group of actions that all communicate with the same session façade. You could also consider grouping all actions involved in CRUD operations on domain objects. (CRUD stands for create, read, update, and delete. Think of an add/edit/delete/listing of products for an online e-commerce store.)

If you can group related actions into one class, you can create helper methods that they all use, thus improving reuse (or at least facilitating it). Additionally, if these helper methods are only used by these related actions and the actions are in the same class, then the helper methods can be encapsulated (hidden) inside this one class.

The DispatchAction class is used to group related actions into one class. DispatchAction is an abstract class, so you must override it to use it. It extends the Action class.

It should be noted that you don't have to use the DispatchAction to group multiple actions into one Action class. You could just use a hidden field that you inspect to delegate to member() methods inside of your action. On the other hand, you could use the action element’s parameter attribute and decide which method to invoke based on the value of the parameter attribute. In fact, this is the approach of you took when creating the user registration wizard in Chapter 3 (*Working with ActionForms* and *DynaActionForms*).

The DispatchAction uses a hidden request parameter to determine which method to invoke. Thus, subclasses of the DispatchAction have many methods with the same signature of the execute() method. The name of the hidden request parameter specifies the name of the method to invoke. You specify the name of the hidden parameter by using the parameter attribute of the action element.

Rather than having a single execute() method, you have a method for each logical action. The DispatchAction dispatches to one of the logical actions represented by the methods. It picks a method to invoke based on an incoming request parameter. The value of the incoming request parameter is the name of the method that the DispatchAction will invoke.

Let’s cover an example that groups multiple actions into one action. You will change the previous example to use the DispatchAction. The example you will be using is the user registration wizard from the ActionForm chapter (Chapter 3). To use the DispatchAction, follow these steps:

1. Create an action handler class that subclasses DispatchAction:

```
public class UserRegistrationMultiAction
                            extends DispatchAction {

    ...

}
```

2. Create a method to represent each logical related action:

```
      public ActionForward processPage1(
            ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {

...
      }

      public ActionForward processPage2(
            ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {
...
      }
```

Notice these methods have the same signature (other than the method name) of the standard Action method.

3. Create an action mapping for this action handler using the parameter attribute to specify the request parameter that carries the name of the method you want to invoke:

```
<action path="/userRegistrationMultiPage1"
        type="strutsTutorial.UserRegistrationMultiAction"
        name="userRegistrationForm"
        attribute="user"
        parameter="action"
        input="/userRegistrationPage1.jsp">

        ...

</action>

<action path="/userRegistrationMultiPage2"
        type="strutsTutorial.UserRegistrationMultiAction"
        name="userRegistrationForm"
        attribute="user"
        parameter="action"
        input="/userRegistrationPage2.jsp">

        ...

</action>
```

   Based on this code, the DispatchAction that you created uses the value of the request parameter named method to pick the appropriate method to invoke. The parameter attribute specifies the name of the request parameter that is inspected by the DispatchAction.

4. Pass the action a request parameter that refers to the method you want to invoke.

   The userRegistrationPage1.jsp contains this hidden parameter:

```
<html:hidden property="action" value="processPage1"/>
```

   while the userRegistrationPage2.jsp contains this hidden parameter:

```
<html:hidden property="action" value="processPage2"/>
```

The above implementation sends a hidden field parameter instead that specifies which method to invoke.

If you were going to implement a CRUD operation, you might have methods called create, read, list, update, and delete. Essentially, the action that was responsible for the read operation (the R in CRUD) would set a string in request scope (set to update) that could be written back out as the hidden parameter by the JSP. The action that was responsible for creating a new user would set that same string to create. The idea is that the action before the form display always sets up the hidden parameter for the form to submit back to the next action. If there was a listing of users, the listing would have two links that point to this action with parameters set to delete users and to read/update (edit) users with the following query strings:

```
?method=delete&userid=10
?method=read&userid=10
```

LookupDispatchAction

The LookupDispatchAction is a lot like the DispatchAction (which it subclasses). Except it does a reverse lookup against the resource bundle with the label of a button, which is the value of the request. The idea behind a LookupDispatchAction is to provide a convenient way to select a dispatch() method when the user clicks a button. Essentially, the LookupDispatchAction associates the dispatch() method with a button on the form. You have to implement a special method that maps the message resource keys to the methods you want to invoke.

Note: I personally avoid LookupDispatchActions like the plague. They are difficult to configure, difficult to debug, and seem to be prone to error. An alternative is to use something similar to LookupDispatchAction such as creating your own action called RequestNameDispatchAction. It can perform a similar function as LookupDispatchAction, but it's much easier to configure. You can make RequestNameDispatchAction look for names, for example, that start with action, as in action.save or action.remove. It then removes the "action." part of the name and uses the rest to invoke the method. This is easier since you don't have to create the mappings between resources keys and method names.

For this example, you will modify your user registration form to use the LookupDispatchAction. You will remove the Submit button from before and add two buttons. One button will be labeled Save; this button will do the same action as before, namely, save the user to the system. The second button will be called Remove, and for this exercise, you will just print out that the button has been clicked. To use the LookupDispatchAction, perform the following steps:

1. Create an action handler class that subclasses LookupDispatchAction:

```
public class UserRegistrationAction extends LookupDispatchAction {
   ...
}
```

2. Next, you create a method to represent each logical related action: save and remove:

```
public class UserRegistrationAction
                 extends LookupDispatchAction {

     private static Log log =
          LogFactory.getLog(UserRegistrationAction.class);



     public ActionForward remove(
          ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
          throws Exception {
          log.debug("IN REMOVE METHOD");
          return mapping.findForward("success");
     }

     public ActionForward save(
                ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
                throws Exception {

                /* Create a User DTO and
                  copy the properties from the userForm */
                User user = new User();
                BeanUtils.copyProperties(user, form);

                DataSource dataSource =
                            getDataSource(request, "userDB");
                Connection conn = dataSource.getConnection();

                try {

                      /* Create UserDAO */
                      UserDAO dao = DAOFactory.createUserDAO(conn);

                      /* Use the UserDAO to insert
                         the new user into the system */
                      dao.createUser(user);

                } finally {
                      conn.close();
                }

                return mapping.findForward("success");
          }

    }
```

The two action methods, save and remove, have identical signatures to the standard Action.execute() method (except for the method name).

3. Implement the getKeyMethodMap() method to map the resource keys to method names:

```
protected Map getKeyMethodMap() {
      Map map = new HashMap();
      map.put("userRegistration.removeButton", "remove");
      map.put("userRegistration.saveButton", "save");
      return map;
}
```

4. Create an action mapping for this action handler using the parameter attribute to specify the request parameter that carries the name of the method you want to invoke:

   ```
   <action path="/userRegistration"
           type="strutsTutorial.UserRegistrationAction"
           name="userRegistrationForm"
           attribute="user"
           input="/userRegistration.jsp"
           parameter="action">

               ...
             <forward name="success"
                      path="/regSuccess.jsp" />
             <forward name="failure"
                      path="/regFailure.jsp" />
   </action>
   ```

   Notice that you specify the parameter to the action. This has a similar meaning to the DispatchAction. Essentially, this means the LookupDispatchAction inspects the label of the button called action. The label will be looked up from the resource bundle, the corresponding key will be found, and the key will be used against the method map to find the name of the method to invoke.

5. Set up the messages in the resource bundle for the labels and values of the buttons. Inside your resource bundle (e.g., application.properties), add the following two entries:

   ```
   userRegistration.removeButton=Remove
   userRegistration.saveButton=Save
   ```

   Notice that the keys are the same keys you used in the getKeyMethodMap.

6. Use the bean:message tag to display the labels of the button and associate the buttons with the name action:

   ```
   <%@ taglib uri="/tags/struts-html" prefix="html"%>
   <%@ taglib uri="/tags/struts-bean" prefix="bean"%>
   ...
       <html:submit property="action">
         <bean:message key="userRegistration.removeButton"/>
       </html:submit>

   ...
       <html:submit property="action">
           <bean:message key="userRegistration.saveButton"/>
        </html:submit>
   ...
   ```

The final result is that when the user clicks the Remove button in the HTML form, the remove() method is called in the action. When the user clicks the Save button, the save() method is called in the action.

SwitchAction

The SwitchAction class is used to support switching from module to module. Let's say you have an action that wants to forward to an action in another module. Say you have a web application that uses Struts. Struts has two modules: the default module and a module called admin, both shown below:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
       org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml
        </param-value>
    </init-param>
    <init-param>
        <param-name>config/admin</param-name>
        <param-value>/WEB-INF/struts-config-admin.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

The init parameter config/admin defines the admin module. The config init parameter defines the default module.

Now, let's say you have an action in the default module that edits users and you want to delegate the display of those users to an action in the admin module.

Perform the following steps:

1. First, map a SwitchAction into the default module as shown here:

```
<action
    path="/switch"
    type="org.apache.struts.actions.SwitchAction"
    >
</action>
```

2. Now, you can set up a forward in the action that edits the users as follows:

```
<action
      path="/userSubmit"
      attribute="userForm"
      input="/form/userForm.jsp"
      name="userForm"
      scope="request"
      type="action.UserAction">
  <forward
   name="success"
  path="/switch.do?page=/listUsers.do&amp;prefix=/admin"
      />
</action>
```

Notice that this forward passes two request parameters. The page parameter specifies the module relative action. The second parameter specifies the prefix of the module—in this case, admin. You don't have to use forwards to use the SwitchAction; any JSP can link to the SwitchAction to move to any module. The listUser.do action is not defined in the default module; it's defined in the admin. The forward in this example forwards to the action at the path /admin/listUsers.do. The listUser.do is defined in /WEB-INF/struts-config-admin.xml, and the userSubmit.do action is defined in /WEB-INF/struts-config.xml.

Action Helper Methods

The Action class contains many helper methods, which enable you to add advanced functionality to your Struts applications. Some of the functionality you'll cover in this section includes:

- Make sure that a form is not submitted twice (see the section "Action saveToken and isTokenValid").
- Display dynamic messages that are i18n-enabled (see the section "Action saveMessages and getResources").
- Allow users to cancel an operation (see the section "Action isCancelled").
- Allow users to change their locale (see the section "Action getLocale and setLocale").

Action saveToken and isTokenValid

Have you ever wanted to make sure that a user does not submit a form twice? Perhaps you have even implemented your own routine that does this. The idea behind saveToken and isTokenValid is to make sure that a form has not been submitted twice. It is nice to know that this functionality is built into Struts, and it's handled by many of its internal operations.

Struts provides transaction tokens to ensure that a form is not submitted twice. A transaction token has nothing to do with Extended Attribute (XA) transactions, Enterprise JavaBeans (EJB), or Java Transaction Services (JTS). A transaction token is a unique string that is generated. The token is submitted with the html:form tag. Several classes and one custom tag are involved in this bit of choreography.

To use a transaction token, follow these steps:

1. Before you load the Java Server Pages (JSP) page that has the html:form tag on it, call saveToken inside an action. To do this, you have to make sure an action is called before the JSP page loads.

   Continue with the last example. Let's say you had an action mapping that was associated with the user registration page as follows:

   ```
   <action path="/userRegForm" forward="/userRegistration.jsp" />
   ```

   The code above just associated a JSP page with an action. Then, any JSP page that links to the input form would link to it like this:

   ```
   <html:link action="/userRegForm">User Registration</html:link>
   ```

   Therefore, no JSP links directly to /userRegistration.jsp. If this is the case, it's easy to start using transaction tokens.

2. When the user submits the form, call isTokenValid and handle the form only if the token is valid.

   Now, let's say that you want to make sure that the user cannot hit the Back button in the browser and submit the form twice. To do this, you must change the action mapping associated with the input form to map to an action that will call the saveToken() method of Action:

```
<action path="/userRegForm"
        type="strutsTutorial.UserRegistrationAction"
        parameter="load">

          <forward name="success"
                   path="/userRegistration.jsp" />
</action>
```

   Action's saveToken() method generates and saves a transaction token and puts it in session scope under the key Globals.TRANSACTION_TOKEN_KEY. Think of a transaction token as a unique string.

   Notice that the action mapping for userRegForm sets the parameter to load. The action will use the parameter to load the form.

You already have this action defined. Thus, you need to modify the `UserRegistrationAction` so that it can handle loading the form by calling saveToken:

```
public class UserRegistrationAction extends
                                 LookupDispatchAction {

     private static Log log =
            LogFactory.getLog(UserRegistrationAction.class);


     public ActionForward execute(
          ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
          throws Exception {
          log.trace("UserRegistrationAction.execute");

          if ("load".equals(mapping.getParameter())){
               return load(mapping, form, request, response);
          }else{
           return super.execute(mapping, form,
                                 request, response);
          }
     }

     private ActionForward load(
              ActionMapping mapping,
              ActionForm form,
              HttpServletRequest request,
              HttpServletResponse response)
                          throws Exception{
          log.debug("In LOAD Method");
          saveToken(request);
          return mapping.findForward("success");
     }
```

If you have been following along, you realize that you want to pass the parameter back to the action when the form is submitted. Moreover, you may think you need to add your own hidden field to the form. Actually, the html:form tag does this bit of magic for you; in fact, here is a code snippet from the html:form custom tag handler:

```
String token =(String) session.getAttribute
      (Globals.TRANSACTION_TOKEN_KEY);

if (token != null) {
     results.append(
     "<input type=\"hidden\" name=\"");
     results.append(Constants.TOKEN_KEY);
     results.append("\" value=\"");
     results.append(token);
     if (this.isXhtml()) {
        results.append("\" />");
     } else {
        results.append("\">");
     }
}
```

Now, as you'll recall from step 2, when the user submits the form call isTokenValid, the code should handle the form only if the token is valid. Now that you know the form will have the transaction token, you can check to see whether it exists by using isTokenValid. When the user submits the valid form, the save() method is called as follows:

```
public class UserRegistrationAction extends
                                    LookupDispatchAction {

     private static Log log =
            LogFactory.getLog(UserRegistrationAction.class);


     public ActionForward execute(
          ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
```

```
        throws Exception {
        log.trace("UserRegistrationAction.execute");

        if ("load".equals(mapping.getParameter())){
             return load(mapping, form, request, response);
        }else{
         return super.execute(mapping, form,
                              request, response);
        }
}

public ActionForward save(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        if ( !isTokenValid(request, true)){
             return mapping.findForward("failure");
        }

        /* Create a User DTO and copy
            the properties from the userForm */
        User user = new User();
        BeanUtils.copyProperties(user, form);

        DataSource dataSource = getDataSource(request, "userDB");
        Connection conn = dataSource.getConnection();

        try {

             /* Create UserDAO */
             UserDAO dao = DAOFactory.createUserDAO(conn);

             /* Use the UserDAO to insert
                  the new user into the system */
             dao.createUser(user);

        } finally {
             conn.close();
        }

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

Because of the way the add() method is implemented, it will not allow a user to submit a form, then hit the Back button in their browser, and submit the same form again. If the token is valid, the add() method will do something useful with the form and then forward to success. If the token is not valid, the user is forwarded to the "resubmit" (failure) forward.

Then, the saveToken and isTokenValid() methods are implemented in the Action class. They have been refactored, and now all they do is delegate to methods of the same name in TokenProcessor (in the Struts util package). Thus, you can use these methods in your own web components (e.g., the Tile controller, a custom RequestProcessor, JSP custom tags).

The RequestUtils computeParameters() method optionally adds a transaction token to its list of parameters. The computeParameters() method is used by the html:link, bean:include, html:rewrite, and logic:redirect tags. Many tags are transaction token aware.

Take a look at the JavaDocs for TokenProcessor to learn more and to see some variations of these methods. You can check the source code for TokenProcessor to gain a good understanding of how things really work.

Action isCancelled

It is often nice to give your user the ability to cancel an operation. Struts provides a special input field called html:cancel, which you can use inside your JSP as follows:

```
<html:cancel/>
```

You can use this input field to cancel a form submission. Then, you just use the isCancelled() method in the action handler to see whether the action was canceled:

```
public class UserRegistrationAction extends
                                    LookupDispatchAction {

     private static Log log =
            LogFactory.getLog(UserRegistrationAction.class);


     public ActionForward execute(
          ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
          throws Exception {
          log.trace("UserRegistrationAction.execute");

          if (isCancelled(request)) {
               log.debug("Cancel Button was pushed!");
               return mapping.findForward("welcome");
          }

          if ("load".equals(mapping.getParameter())){
               return load(mapping, form, request, response);
          }else{
           return super.execute(mapping, form,
                                    request, response);
          }
     }
```

This action handler checks to see whether the operation was canceled. If it was, the action forwards the user to the welcome page.

Tip: With both the LookupDispatchAction and the DispatchAction, you can create a cancel() method that handles the case when the user hits the Cancel button.

Action getLocale and setLocale

You can dynamically set the locale for the user’s session. Essentially, you let users decide what locale they want, using a form or link. Think of an automated teller machine (ATM). The first thing it often asks you is what language you speak, and you press the appropriate button. Well, you could do the same thing for your site.

To do perform this magic, you have an action tied to a form or link that reads the locale information from the request, creates a java.util.Locale, and stores the locale in session scope under the key Globals.LOCALE_KEY using the setLocale() method:

```
public class SetLocaleAction extends Action {

  public ActionForward execute(
             ActionMapping mapping,
             ActionForm form,
             HttpServletRequest request,
             HttpServletResponse response) throws Exception {

    String language = request.getParameter("language");
    String country = request.getParameter("country");

    Locale locale = new Locale(language, country);
    setLocale(request,locale);


    if (getLocale(request).getLanguage().equals("en")){

        System.out.println("the language is set to English");
        // Do something special for those who are logged in
        // who speak English. Perhaps some features
        // are different or disabled for English.
        ...
    }
    ...

    return actionMapping.findForward("success");

  }
}
```

Notice that you can also get the locale and operate on it accordingly if needed. All of the tags (such as bean:message) respect this locale for the entire user session.

See Chapter 14 (I18n) for more details on how to support the internationalization (i18n) features with resource bundles and bean:message.

Action saveMessages and getResources

Back in the good old days of Struts(Struts pre-1.1), people liked ActionErrors. They started using them for tasks for which they were never intended. People started using ActionErrors as a general-purpose way to display i18n-enabled dynamic messages. Then, Struts 1.1 added the concept of ActionMessages. Now, you can use ActionMessages to display dynamic i18n-enabled messages without feeling like a hack.

Working with ActionMessages is nearly identical to working with ActionErrors. Here is an example of adding ActionMessages that you want to be displayed inside an Action's method:

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {

    ActionMessages messages = new ActionMessages();



    ActionMessage message = new
               ActionMessage("welcome.greet");
    messages.add(ActionMessages.GLOBAL_MESSAGE, message);

    ...
    saveMessages(request,messages);
    ...
```

Note: The code above would apply to saving errors as well. Just replace ActionMessages, ActionMessage, and saveMessages with ActionErrors, ActionError, and saveErrors.

Then, to display the messages in the JSP, you use the html:messages tag as follows:

```
<ul>
<font color='green' >
<html:messages id="message" message="true">
  <li><%= message %></li>
</html:messages>
</font>
</ul>
```

The html:messages tag will iterate over all of the messages. Notice that the message attribute of html:messages is set to true. This forces html:messages to get the messages from Globals.MESSAGE_KEY in request scope. If you do not set the message attribute, the html:messages tag will display the errors instead (Globals.ERROR_KEY).

What if you want to display an arbitrary number of messages that could vary by locale? In that case, you use getResources to get the number of messages for that locale:

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {

    ActionMessages messages = new ActionMessages();



    ActionMessage message = new
           ActionMessage("welcome.greet");
    messages.add(ActionMessages.GLOBAL_MESSAGE, message);

    String num =
        this.getResources(request)
            .getMessage("welcome.messageNum");

    int messageCount = Integer.parseInt(num);
    for(int index=0; index<messageCount; index++){
        String messageKey="inputForm.message" + index;
        message = new ActionMessage(messageKey);
        messages.add(ActionMessages.GLOBAL_MESSAGE,
                       message);
    }
    saveMessages(request,messages);

    System.out.println(Globals.MESSAGE_KEY);
    if (messages ==
         request.getAttribute(Globals.MESSAGE_KEY)){
       System.out.println("its there can't you see it");
    }
  ...
```

The previous code corresponds to the following resource bundle entries:

```
welcome.greet=Hello Welcome to ArcMind Corporation
welcome.messageNum=2
welcome.message0=Please be sure to fill out your phone ...
welcome.message1=Pick a user name and password you can remember
```

The welcome.messageNum() method specifies the number of messages for a page. The for loop iterates up to the count of messages, grabbing each message for each welcome page—that is, welcome.message0, welcome.message1, and so on.

These examples cause the messages in the resource bundle to display in an unordered list at the top of the input JSP page.

The RequestUtils class has a method called getActionMessages() that is used by logic:messagePresent and html:messages. The getActionMessages() method allows you to retrieve the messages saved by Action.saveMessages and Action.saveErrors. You would use saveErrors, when the messages you are saving are really error messages.

Populating ActionForms with Domain Objects

To populate an ActionForm with a domain object so that it can be displayed on an input JSP with html:form, you first have to create an action that is called before the form:

```
<action path="/readUser"
        type="rickhightower.ReadUserAction"
        name="UserForm" scope="request"
        validate="false">
    <forward name="success" path="/userForm.jsp"/>
</action>
```

Notice that you mapped in the form that the userForm will display. This form will be created and passed to the execute() method of ReadUserAction before the userForm.jsp service() method is called. This gives the code an opportunity to populate the form and put it into scope so that the userForm.jsp's html:form tag can display it.

Here is an example of the ReadUserAction's execute() method:

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
                       throws Exception {

        //Get the user id of
    String id = request.getParameter("id");

        //Obtain the default DAOFactory
    DAOFactory factory = DAOFactory.getDefaultfactory();

        //Obtain the DAO for users
    UserDAO userDAO = factory.getUserDAO();

        //Retrieve the user from the database
    UserDTO user = userDAO.getUser(id);

        //Cast the form to a UserForm
    UserForm userForm = (UserForm)form;

        //Copy over the properties from the DTO to the form.
    BeanUtils.copyProperties(userForm,user);
}
    ...
```

Notice that the action does not create a new UserForm. It did not need to because the action mapping caused a form to be created. The code looks up the user based on an ID that was passed as a request parameter. Then, it uses BeanUtils.copyProperties (org.apache.commons.beanutils.BeanUtils) to copy the domain object's properties to the userForm. The userForm properties should all be of type string (strings are best for validation). The domain objects properties can be any primitive type or wrapper objects. (BeanUtils will perform the type conversion automatically.)

However, BeanUtils does not convert dates well. This means that you cannot have properties that are dates with the same property name as the form. If you do, you will have to use the BeanUtils.describe() method in combination with the BeanUtils populate() method. The describe() method converts a bean into a Map, where the name/value pairs in the Map correspond to the properties in the bean. The populate() method populates bean properties with name/value pairs from a Map. Therefore, you would describe the domain object, remove the offending property from the Map, and then use the populate() method. For simple form-to-domain object mapping, it is easy to use form.setXXX(domainObject.getXXX()). For more complex forms, BeanUtils saves the day.

It is probably best just to avoid property name type mismatches by having different names, as follows:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
String dateString = sdf.format(user.getHireDate());
userForm.setDateHire(dateString);
```

Note that userForm's property is called dateHire and that the domain object user date property is called hireDate. A third option is to create and register a custom Converter; see the JavaDocs for org.apache.commons.beanutils.converters to learn more. (Read Chapter 12 to learn more about BeanUtils.) Using converters to do this is a best practice.

Uploading Files with Actions

Let’s say that you want users to be able to upload their personal pictures. To do this, you must enable file upload in your Struts application. Of course, you don’t want them to upload a 1GB image of themselves, so you have to put some limits on the pictures that users upload.

First, set up two areas for enabling the uploading of files: enable the user to supply the file with an HTML form, and enable the Struts application to process the file upload.

To enable the user to upload a file, set the encoding type of html:form to multipart/form-data and use the html:file tag as follows:

```
<html:form action="/UserUpdate"
           method="post"
           enctype="multipart/form-data">

   <html:file property="userImage"/> <br />
  ...
```

The property userImage in the userForm is of type FormFile (org.apache.struts.upload.FormFile). FormFile represents the file sent by the client. FormFile has a method called getInputStream(), which returns an InputStream. The action handler for this form uses FormFile to access the file:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
                             throws Exception {
    UserForm userForm = (UserForm) form;

    InputStream inputStream =
         userForm.getUserImage().getInputStream();
    // Do something with the inputStream, it is the file data.
    ...
  }
```

To set up the Struts application to restrict file size and to specify the directory location, you could set up a controller element:

```
<controller maxFileSize="200K"
            tempDir="/temp/struts/user/uploads/"/>
```

The code snippet above states that the temporary files will be put into a directory provided by your servlet container in the directory /temp/struts/user/uploads/, and it specifies that users are allowed to upload files up to 200 KB only. You can also specify the size of the input buffer and other parameters. When you compare this code to the way it used to be done, you can easily see that the Struts approach is much easier than the past alternatives.

Summary

In this chapter, you learned all of the standard actions:

- ForwardAction
- IncludeAction
- DispatchAction
- LookupDispatchAction
- Switch Action

You also learned some of the helper methods that the Action class ships with. In addition, you now know how to use the saveToken, isTokenValid, and getResource() methods. Finally, you learned how to display messages and save errors with the saveMessages and saveErrors() methods as well as how to populate ActionForms with Domain Objects. You can now handle some of the most common tasks when developing Struts applications.

Chapter 6

MVC for Struts Smarties

Mastering MVC and Model 2

Struts promotes a Model 2 / MVC architecture. Essentially, Struts architecture is based on the J2EE blue print for Java, which heavily promotes using MVC. Using MVC can help the maintainability and testability of your code base. It has been said that Struts is the instantiation of the J2EE Blueprint's View of the web tier. Understanding the value of Model 2 and MVC is probably the most important thing you can do to create a successful Struts project.

Note: Read the J2EE Blueprint at http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/index.html.

Many times applications can have more than one type of client (e.g., web clients rich GUI clients, etc.). For client independence and testability, it is good to develop your applications where the application data and business rules are separated from a particular client implementation. In order to adapt the application to multiple clients and to organize the code into separate areas of concern, you typically implement the systems using a variation of Model-View-Controller Architecture (MVC).

MVC is an open architecture with multiple tiers. The Struts framework uses a variation of MVC that was adapted for web development called Model 2. The following session considers MVC/Model 2 and its ramification on Struts usage.

Warning! Java promotes object-oriented programming, but does not enforce it. (You can create non-OOP code in Java. Similarly, Struts promotes MVC / Model 2 architecture, but does not enforce it. Making Struts foolproof with regards to MVC is difficult as foolish developers are ingenious. To get the best out of Struts, you need to know MVC / Model 2.

Model Tier

The Model is the application data and the business rules of your application. The business rules are the rules for changing your application data. Many types of applications can share your application's Model. The Model code should not be dependent on Struts or the JSP/Servlet API.

The Struts framework provides no Model classes per se. All applications should have different Models.

Note: The Struts framework does not provide any Model objects. Your Model is specific to your application. Often, developers develop their Model with EJB, JDO, or Hibernate. The Model consists of the persistence layer and the business rules for modifying the persistence layer.

Many references refer to the ActionForm as being part of the Model. This is a matter of perspective. Even if you View the ActionForm as part of the Model, you should not use it directly in your Model for most relatively complex applications because the ActionForm is already coupled to the Struts framework

View Tier

The View tier is what the end-user sees. The View displays parts of the Model to the end user. Different types of clients (web clients, rich GUI clients, etc.) have different Views. The View should never modify Model data directly, nor should it interact with business rules of the Model.

The Struts framework does not specify a particular View technology. However, the Struts framework does provide many JSP custom tags to aid in developing with JSP. You can also develop Views with other technologies like Velocity (which has Struts tools), JSF and XSLT. Struts is fairly “View agnostic.”

Note: Many of the JSP custom tags that ship with Struts are essentially deprecated in favor of their JSTL equivalent (c:forEach vs. logic:iterate). Other Struts custom tags are key to Struts development, such as html:form.

Controller Tier

A controller acts like the glue between the Model and the View. The controller defines the flow of the application. It interprets events from the user (user gestures) and acts like the mediator between the View and the Model. It parses the user gestures into actions it performs on the Model. Different types of clients require different types of controllers. In Struts, the actions (Action), ActionServlet, and config objects (created from struts-config.xml) act as the controller.

The controller accepts user events from the View. In a web interface, the user events are page loads (HTTP GET) and form submissions (HTTP POST) requests.

To help separate the Model from the View, the controller should convert the request data, scoped beans, and request into business actions. The Struts framework does part of this for you. You do the rest of the conversions as your write actions (Action) that act as intermediaries from the web request to the Model.

It is the controller's job to select the next View to display; this is part of managing the flow of the application. The actions that you write return an ActionForward to the next View. Thus, the ActionForward represents the next View in the system. Moreover, it is the job of the actions you write to return the next View. In the actions you write, you typically interact with the Model's business rules, look up Model data, map the Model data into scope (request, session, etc.), and then select the next View so the View can display the Model data.

Rules for Writing Proper MVC Application with Struts

Rules for the View Tier

The View should never select the next View. This means that the JSP files should never use jsp:forward, logic:forward, logic:redirect, or core:redirect (JSTL) to change the View to a new JSP. Your actions should select the next View by returning the appropriate ActionForward. In fact, you should not even from JSPs to other JSPs; more on this later.

Try to only use stupid JSPs. A stupid JSP is a JSP that does not have a lot of logic. Try to limit the logic to looping through collection of data and displaying objects if they are in scope (logic:present and core:if value="not empty…). Often times I see JSP errantly sorting data, or checking with a Model to see if a part of the page should display. Instead, let the action sort the data before it gives it to the JSP, and let the action decide if the Model object(s) needs to be displayed by interacting with the business rules of the Model. JSP should not know why it is displaying an object; if it is there, it simply displays it. That way, even if the business rules for displaying the tag change, the JSP does not have to change.

Tip: A certain amount of pragmatism should be used here. For example, if you do not have a library that does sorting and pagination, you might use existing Custom Tag libraries that perform these tasks in the View tier. It is almost always better to use a working solution than to write one from scratch. Use these rules as a guide, not as the gospel.

A funny thing happened. Developers added custom tags to simplify JSP development, and they removed Java code (Java scriptlets and expressions) from the JSP. Then, people started writing logic custom tags like they write Java. This helped, but maintainability problems remained. The key is to not only limit the Java code in a JSP, but to limit the logic in the JSP. Keep your JSPs simple and your web application will be a lot more maintainable.

If your JSPs start to have too much logic (Struts logic:* tags, and JSTL core:* tags), move that logic into the action. If you are worried about too much code in your action, use action chaining. For example, you could have an action that sorts the data that another action put into scope and then forward that to the JSP.

Tip: Again, use this as a guide. At times, you may find yourself putting some logic tags in your JSP page. For example, some developers use the logic:present tag with the role attribute to turn off parts of the page if the user is not in a certain role. Sometimes it's just easier to do it that way. (Once you master tiles, you will find yourself doing this less and less.) Rather than being dogmatic about removing logic from you JSP pages, note when you do it and identify that area as a good place to do some restructuring in a future iteration of the project.

Rules for the Controller Tier

Actions should be small. Actions should never do JDBC directly. In fact, actions should never talk to EJBs directly, or Hibernate, or etc. Actions should instead only talk to façade facets of the Model. If you are using Session Facades (EJB), use delegate objects (wrapper objects) so that the action does not talk directly to the EJB. Limiting the action to only talk to a façade of the Model helps keep the action testable with unit tests. Conversely, tying the action to some underlying technology couples the action to that technology and makes the action harder to test.

Actions should never implement business rules or perform complex calculations. Actions should delegate those tasks to the Model.

Tip: It is a good idea to use a Business Delegate, or Manager, and not to even talk to your persistent layer (in our example DAO) directly from the action. To keep the examples simple, we choose to have the action talk directly to the persistent layer (DAO). In future chapters, you will revisit and correct it.

Rules for the Model

The Model should be completely divorced from the controller. The Model should not be dependent on any class in the Struts hierarchy. The individual Model objects should be hidden behind a façade.

Tip: A great OOP guru once said that the great thing about objects is that they can be replaced. As it turns out, the controller is more testable if it is divorced from the Model implementation. This allows you to substitute stubbed out objects that return hard-coded results needed to test the controller logic in your actions. There are several ways to do this. The controller code (actions), could look up Model facades in a registry, and instantiate them. You could also use an IOC (Inversion of Control) container like Spring, PicoContainer, HiveMind, or XWork. The Spring et al framework is beyond the scope of this discussion, but look for a chapter in the future on using the Spring framework plug-in with Struts. A great discussion of IOC can be found at http://www.picocontainer.org, which is a type 3 ICO container. Another good reference is Martin Fowler's paper on dependency injection at http://www.martinfowler.com/articles/injection.html.

Don't Link to JSPs from JSPs

Generally speaking, it is a bad idea to place any direct links to other JSP pages within your JSP pages (e.g., <html:link page="/page.jsp">). For one reason, it's not a good design decision; the struts-config.xml file, which is part of the controller, should contain the entire flow of your application.

In the MVC architecture, it is the role of the controller to select the next View. The controller consists of the Struts configuration, the ActionServlet, and the actions. If you add a link directly to another JSP, you are violating the architectural boundaries of the Model 2 architecture. (Model 2 is the instantiation of the MVC architecture for web applications.)

At times, however, all you really need is just a plain link; you don't want (or need) an action to execute first. Perhaps there are no objects from the domain that need to be mapped into scope in order for the View to display. Perhaps the page is very simple. In this case, a better approach is to use the forward attribute of the Action element in the struts-config file, and link using the Action or forward attribute of the html:link tag (e.g., <html:link action=”someAction”… or <html:link forward=”someForward”).

The forward attribute of the Action element acts as a bridge from the current View (JSP) and the pages it links to. It uses the RequestDispatcher to forward to a specified web resource. It’s the glue that allows you to link to an action instead of linking directly to a JSP.

Tip: Another approach developers use is the ForwardAction, which has the same effect as the forward attribute.

Later, if you need to, you can change the action mapping in the Struts configuration file so that every page that links to that action will link to the new action. In addition, you can change the action to a custom one that you write instead of using the forward attribute that Struts provides.

If you find that you have a lot of links to JSP pages from other JSP pages, you may not understand MVC very well. Using ActionForward may mask the fact that your design is inherently flawed. JSPs linking to other JSPs, whether they use forward attributes or not, work only for the simplest of web applications in MVC. If it works in your application, chances are you’re probably doing something wrong. Either your JSPs or custom tags are too “fat.”

Complex MVC web applications typically need an action to execute first; it is the job of the action to map Model items into scope so that the View (typically JSP) can display them. If this is not the case for your web application, you are probably putting too much logic in the JSP pages (or in custom tags that the JSP pages use)—which can be a source of huge maintenance issues later on.

Summary

The JSP pages and custom tags should contain only display logic, and this logic should be kept to a bare minimum. Thus, the JSP pages and custom tags should not talk directly to the Model—they only display different pieces of the Model. In essence, they should speak only to the Value object and Data Transfer objects from the Model. Again, the action talks to the Model (typically via Model façade objects) and delegates the display of the Model objects to the View. The Model, in turn, implements the persistence and business rules for the application.

Adopting a strict MVC architecture is generally a good idea; it keeps your JSP smaller and more focused. JSPs are harder to test than actions, so adopting MVC increases the liquidity and flexibility of your code base.

In addition, as you saw in the testing chapter, dividing your application into layers makes the application easier to test.

Working with Struts Custom Tags

Mapping the Model to the View with Struts Custom Tags

As stated in past chapters, Struts is somewhat "View agnostic." However, the Struts framework provides a set of custom tags to map Model objects of your application to the View. Struts custom tags include HTML tags, bean tags, and logic tags.

JSP Limitations

Mixing the presentation (HTML) with the implementation (Java) in a JSP can create a number of issues. For one thing, maintenance of JSPs can become quite complex. Both HTML page designers and Java coders have to work on the same pages. This usually results in at least one party, if not both, being unhappy with the final results. The HTML page designer's changes to the presentation can affect the Java coder's implementation and vice-versa; the final result is that they step on each other's toes with each change.

Despite these complications, JSP aimed to separate roles (Java developers and HTML page designers) from the beginning. Therefore, JSP provides a number of built-in actions like: jsp:useBean, jsp:getProperty, and jsp:setProperty.

However, even with these built-in actions, it's difficult to only use built-in actions to create a JSP page that does not rely on any Java code scriptlets. Even simple layout logic, like iteration, requires Java code scriptlets without custom tags.

Custom Tags

Sun has a strategy for keeping Java code out of JSP: JSP custom tags. Custom tags are like custom actions, and they function much the same way as built-in actions. This allows you to create JSP pages that have no Java code. The layout logic, like iterating over a list, is done with custom tags. The Struts framework provides a number of custom tags to perform this type of logic. The custom tag framework provides support for both simple and complex tags.

Tip: Note that JSTL, which is the JSP Standard Tag Library, provides many of the same features as the Struts logic and bean tags. The Struts tags predated the JSTL tags. It's recommended that you use the JSTL equivalents whenever possible. The Struts framework provides an additional tag library in the contrib area of Struts called Struts-EL. Chapter 8 covers JSTL and Struts-EL.

Custom tags are really a mechanism for adding new action tags to the JSP grammar. When you write a custom tag library, you are writing an extension to the JSP language. Many custom tags are used to dynamically generate HTML. By allowing JSP pages to be Java-less, page designers have sole control of the presentation of the pages. This allows page designers to work in a format that is familiar to them. Developers, on the other hand, write custom tags, which is very Java-centric. This allows developers and designers to live in perfect harmony, like keys on a keyboard piano.

Typical custom tag capabilities include: inserting text into a page, flow of control, iteration, and cooperating nested tags. Inserting text into a page can be, for example, text from a database query. The flow of control can optionally display part of the page or stop the rendering of the page. Iteration typically implies iterating over a collection or array of objects and displaying their properties. Nested tags can rely on the context of their parent tags. Thus, nested tags can cooperate with their parent tags.

A custom tag may look like the following:

```
<html:link action="/welcome"
        paramId="greetingType" paramName="manager">
```

The above works like the HTML anchor tag (<a>) except that it does URL rewriting and JSP session ID encoding. Custom tags are implemented using Java objects. The Java object is called the tag handler. You can easily write your own tag handlers. In fact, you will learn how to create your own tag handlers in Chapter 11. The container (really the generated servlet), invokes the tag handler transparently when it sees the name of the tag. In the example above, `html` refers to the tag library prefix, while `link` refers to the name of the corresponding tag handler.

A tag library is used to hold and deploy a collection of related custom tags together. Tag libraries extend the functionality of JSPs on a page-by-page basis. Tag libraries typically consist of a JAR file that holds implementing code and a TLD (tag library descriptor) file. TLD files are to tag handlers as web.xml is to servlets, or as ejbjar.xml is to EJB beans. A TLD file is the deployment descriptor for the set of custom tag handlers in the tag library. The TLD file provides mappings between tag names and tag handlers. The TLD file is provided by the developer of the tag library. When you use a TLD file, you typically store the TLD file in the WEB-INF directory of the web application.

You can use multiple tag libraries on the same JSP as long as each tag library has its own prefix. The prefix of the TLD file is specified when you use the tag lib directive to import the tag library into the JSP page.

Using a Custom Tag Library

In order to start using a tag library, you must first acquire this tag library or write a tag library. Then, you have to make the class files available to the web application's classloader. You can do this by adding the JAR file containing the tag library to the WEB-INF/lib directory of your web application. Or, you can put its classes under the WEB-INF/classes directory.

Once the tag library is available to the web application classloader, using a tag library is a two-step process. The first step is declaring that you are going to use the tag library in your web.xml file, as shown below:

```
<taglib>
  <taglib-uri>struts-html</taglib-uri>
  <taglib-location>
      /WEB-INF/struts-html.tld
  </taglib-location>
</taglib>
```

The code above would show up after the welcome file list in your web.xml file. The taglib-uri element specifies the short name that you have given this tag library. The taglib-location element specifies the physical location of the TLD file relative to your web application directory or WAR file. Thus, you must put the TLD files into your web application or WAR file.

The second step is declaring that you're going to use the tag library in your JSP page, as follows:

```
<%@ taglib uri="struts-html" prefix="html" %>
```

The taglib directive above refers to the taglib that you added to your web.xml file. Notice how the `uri` attribute of the taglib directive keys on the value of the taglib-uri element in the web.xml file.

Once you import the tag library into the JSP, you can start using it as follows:

```
<html:link action="/welcome"
        paramId="greetingType" paramName="manager">
```

There is another way to include a tag library into the JSP page. You do this by specifying the URI (which is embedded inside of the TLD file) directly as the URI of the tag-lib directive. For example:

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html"
           prefix="html" %>
```

Tip: The code above allows you to skip the step of referencing the TLD file in the deployment descriptor. I find the code above a little hard to read, and I find it a little hard to remember the URI. I prefer the extra step of adding the tag library to the web application deployment descriptor. This allows me to give the tag library a short name that is easier to remember.

Keep in mind, you are going to learn more about custom tags when you cover creating custom tags in Chapter 11.

Now that you know how to use a custom tag, you need to learn about the custom tags that ship with Struts. But before you do that, let's discuss the trinity of tag attributes.

Trinity of Tag Attributes

Many Struts tags use the same attributes, namely `name`, `property`, and `scope`. The `name` attribute specifies the name of a bean in a given scope whose property is going to be retrieved. The `property` attribute identifies a property from the bean that the tag is going to use (the bean was specified by the `name` attribute). If the `property` attribute is missing, the tag will use the toString value of the `named` JavaBean. The `scope` attribute specifies the location of the bean (page, request, session, or application scope). If you do not specify the `scope` attribute, the tag will search all the scopes from page scope (the innermost) to application scope (the outermost).

The `property` attribute can specify a nested property as well by using dot notation. It can also use square bracket notation to specify a nested indexed property.

For example:

```
<somePrefix:someTag name="department"
                    property="employee[2].lastName"
                    scope="session"… />
```

The code above would grab the department object out of session scope, and call the following sequence of getter methods: department.getEmployee(2).getLastName(), also known as the third employee's last name from the department object. This works for form field tags as well; on form submission, the code above would generate the equivalent to departments.getEmployee(2).setLastName(request parameter value).

HTML Tag Library

The HTML tag library is mainly used to render the fields on input forms. However, there are other uses for the tag library, including: displaying error messages, displaying messages, rendering links with URL rewriting and session id encoding, generating JavaScript validation code, and supporting i18n.

Using the HTML Tags

To use the HTML tag library, do the following:

1. Copy the TLD file, struts-html.tld, to the WEB-INF directory of your web application.
2. The compiled code for the struts HTML tag library is in the struts.jar file, so make sure that the JAR file is in your WEB-INF/lib directory.
3. Add the following taglib element to your web.xml file (WEB-INF/web.xml).

```
<taglib>
  <taglib-uri>struts-html</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
```

Add the following taglib directive to the JSP that wants to use the HTML tag library:

```
<%@ taglib uri="struts-html" prefix="html" %>
```

Once you have properly installed a tag library, you can start using the tags.

html:base

The html:base tag inserts an HTML <base> tag, which allows the page to use relative URL references. If you do not use this tag, the JSP page can only use URLs that are relative to the last requested resource. Note, if using Model 2 type architecture, all of the requests will be to actions.

You would use the html:base tag as follows:

```
<html:base />
```

which would render the following HTML code:

```
<base
```

```
         href="http://localhost:8080/strutsTut/welcome.jsp">
```

The html:base tag would go inside of the html:head tag.

You can specify a different target and server name with the `server` attribute and the `target` attribute as follows:

```
        <html:base target="/foo/welcome.jsp" server="arc-mind" />
```

The above would generate the following HTML:

```
        <base
        href="http://arc-mind:8080/struts-exercise-taglib/html-link.jsp"
        target="/foo/welcome.jsp">
```

The main focus of the HTML tag libraries is the form control elements (fields and buttons).

Common Attributes

Many of the HTML custom tags have a similar set of attributes.

Table 7.1: Common Attributes

Attribute	Description	Optional? (Yes/No)
`property`	Associates this field with the property from the corresponding ActionForm.	No
`accessKey`	Similar in concept to mnemonic in Swing.	Yes
`alt`	Specifies alternative text, which is used when the browser cannot render this tag.	Yes
`altKey`	The `altKey` attribute is the same as the `alt` attribute except that it specifies the name of the key in the resource bundle. The messages in the resource bundle will display. Resource bundles are organized by country and language.	Yes
`disabled`	If set to true, disables the input element, which usually causes it to be grayed out by the browser. Defaults to false.	Yes
`indexed`	Renders indexed property names. The `indexed` attribute only works in the context of the logic:iterate tag. Defaults to false.	Yes
`onblur`	Allows you to write an event handler for when the element loses its focus. You write event handlers with client-side JavaScript.	Yes
`onchange`	Specifies an event handler for when the element loses input focus and its value changes.	Yes
`onclick`	Specifies an event handler when the element is clicked with a mouse.	Yes
`ondblclick`	Specifies an event handler for when the element is double-clicked with a mouse.	Yes
`onfocus`	Specifies an event handler for when the element receives focus.	Yes
`onkeydown`	Specifies an event handler for when the element has focus and the user presses a key.	Yes
`onkeypress`	Specifies an event handler for when the element has focus, and the user presses a key and then releases the key.	Yes
`onkeyup`	Specifies an event handler for when the element has focus and the user presses a key.	Yes
`onmousedown`	Specifies an event handler for when the element receives a mouse click down.	Yes
`onmousemove`	Specifies an event handler for when the element is under the mouse pointer and the mouse pointer moves.	Yes
`onmouseout`	The mouse leaves.	Yes

Table 7.1: Common Attributes

`onmouseover`	The mouse moves over.	Yes
`onmouseup`	The mouse button goes up while over the element.	Yes
`style`	Specifies an in-line style (CSS).	Yes
`styleClass`	Specifies a style class; gets rendered as a `class` attribute.	Yes
`styleId`	Specifies a style id; gets rendered as an `id` attribute.	Yes
`tabindex`	Specifies tab order.	Yes
`title`	Specifies the advisory title text for the input control; this gets used as they tool tip by the browser.	Yes
`titleKey`	The `titleKey` attribute is the same as the `title` attribute, except it gets the title out of the resource bundle using the key specified by `titleKey`. This is used to support internationalization.	Yes
`value`	Specifies a label for a button. If you use the `value` attribute with any other type of input, it will render a default value.	Yes

html:button

The html:button tag is used to render HTML button tags.

Thus, the following tag, which would be inside of an html:form tag:

```
<html:button property="action">BUTTON</html:button>
```

would generate the following HTML:

```
<input type="button" name="action" value="BUTTON">
```

html:cancel

The html:cancel tag is used to render a Cancel button. The cancel tag has special support throughout the Struts framework. There is a helper method in the action class called isCanceled() that checks to see if this button was pressed. In addition, the RequestProcessor checks to see if the Cancel button was pushed before it does form population and validation. If you click the Cancel button, the form will not be validated or populated.

You could use the html:cancel tag inside of a form as follows:

```
        <html:cancel>Cancel</html:cancel>
```

The code above would render the following HTML:

```
        <input type="submit"
               name="org.apache.struts.taglib.html.CANCEL"
               value="Cancel" onclick="bCancel=true;">
```

Notice that the `onclick` attribute sets a flag called bCancel to true (in JavaScript). This is for when you are using the Validator framework to generate client-side JavaScript validation code. See Chapter 4 for more details.

If you wanted to include internationalization support, you could also use the tag as follows:

```
        <html:cancel><bean:write key="form.cancel" /></html:cancel>
```

See Chapter 14 for detailed information on internationalization for more details.

html:checkbox

The html:checkbox tag is used to render a check box.

Here is an example of using the checkbox tag:

```
        <html:checkbox property="booleanProperty" />
```

The ActionForm that corresponds to this each html:form would have to have a property named booleanProperty.

The tricky thing about working with check boxes is that HTTP only sends boxes that are checked. Thus, in your ActionForm’s reset() method, you must set all check boxes to false to allow the request processor to populate the check boxes from the incoming request—see Chapter 3, which covers ActionForms in detail.

html:errors

The html:errors tag is used to display error messages to the end users. These error messages are typically ActionErrors for a given form. This tag is falling out of favor, and you should use the html:messages attribute instead.

Typically, the error messages are stored in the default resource bundle; you can use the `bundle` attribute of the html:errors tag to specify a different resource bundle.

By default, this tag will display error messages in the user's locale, which is stored in session scope; however, you could override this by specifying a locale with the `locale` attribute.

By default, this tag will get the errors (ActionErrors) from request scope under the attribute name Globals.ERROR_KEY. (Global is a helper class, and ERROR_KEY is a constant.) This tag also provides an attribute called `name` that can be used to get the errors from another attribute name in request scope.

When you place errors into an ActionErrors object, you can associate those errors with a form field. The html:errors tag allows you to specify the name of the property associated with a particular set of error messages (i.e., for a form field). The `property` attribute of this tag is used to grab errors for a particular form field. If you do not specify the `property` attribute, you will get the error messages from all of the attributes.

When you use html:errors, you have to define the following HTML markup in your resource bundle:

```
errors.header=<ul>
errors.footer</ul>
errors.prefix=<li class="error">
errors.suffix=</li>
```

Generally speaking, it’s a bad idea to put HTML in your resource bundle. You can use the html:messages tag to get around this limitation.

Once you have the above entries in your resource bundle, use the following code to display all of the errors on the page:

```
<html:errors />
```

To display an error message for a particular field, use this code:

```
<html:errors property="someFieldName" />
```

html:file

The html:file tag is used to allow end users to upload files. Consider that you might want users to be able to upload a picture of themselves. To do this, you must enable file upload in your Struts application. Of course, you don’t want them to upload a 1GB image of themselves, so you have to put some limits on the pictures that users upload.

First, you need to set up two areas for enabling the uploading of files: enable the user to supply the file with an HTML form and enable the Struts application to process the file upload.

To enable the user to upload a file, set the encoding type of html:form to multipart/form-data, and use the html:file tag as follows:

```
<html:form action="/UserUpdate"
           method="post"
           enctype="multipart/form-data">

   <html:file property="userImage"/> <br />
  ...
```

The property userImage in the userForm is of type FormFile (org.apache.struts.upload.FormFile). FormFile represents the file sent by the client. FormFile has a method called getInputStream(), which returns an inputStream. The action handler for this form uses FormFile to access the file:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                          HttpServletResponse response) throws Exception {
    UserForm userForm = (UserForm) form;

    InputStream inputStream =
         userForm.getUserImage().getInputStream();
    // Do something with the inputStream, it is the file data.
    ...
  }
```

To set up the Struts application to restrict file size and to specify the directory location, you could set up a controller element in the Struts config file as follows:

```
<controller maxFileSize="200K"
            tempDir="/temp/struts/user/uploads/"/>
```

This code snippet states that the temporary files will be put into a directory provided by your servlet container in the directory /temp/struts/user/uploads/ and that users are allowed to upload files up to 200 KB only. You can also specify the size of the input buffer and other parameters.

At eBlox, we allowed users to upload product data using this Struts feature, a vast improvement over the way we use to do it. When you compare this Struts feature to the way developers used to have to do it, you can easily see that the Struts approach is much easier than the alternatives.

html:form

The html:form tag renders an HTML form. The html:formtag is associated with an action mapping by the `action` attribute. The `action` attribute specifies the path of the action mapping. Each field in the html:form should correspond to a property of the associated ActionForm.

Therefore, when the user submits the form, the action associated with the action mapping will be invoked (if the form is valid). It is interesting that this tag inspects the action mapping and finds the ActionForm associated with the action mapping. If the ActionForm is in scope, the property values of the ActionForm will be rendered as the values in the HTML form field of the html:form tag. In fact, if the action mapping has an error (e.g., points to an invalid ActionForm), the page with the html:form tag will never display until you fix the action mapping.

There are many more things you can do with html:form, but these are not advisable. You could override the ActionForm associated with this HTML form by using the `scope` and `type` attributes. The `scope` attribute specifies where to look for the ActionForm, and the `type` attribute specifies what type of ActionForm it is (i.e., the fully qualified Java classname). This technique is not used in practice very often, but it is good to know that it exists. This feature will be removed in Struts 1.2, so don't use it.

The `focus` attribute is used to set the initial focus to a particular input field in the form. Most of the other attributes are the same as the attributes for an HTML form; for example, the onreset and onsumbit attributes are the same.

The `enctype` attribute is used when you want to enable the end user to upload a file. To enable the user to upload a file, set the encoding type of html:form to multipart/form-data and use the html:file tag as follows:

```
<html:form action="/UserUpdate"
           method="post"
           enctype="multipart/form-data">

   <html:file property="userImage"/> <br />
  ...
```

See the Chapter 1 tutorial and Chapter 3 (ActionForms) to see more examples of using the html:form tag.

<html:hidden />

The html:hidden custom tag renders a hidden field in a form. Its usage is similar to the other input field wrapper tags. For example:

```
<html:hidden property="foo"/>
```

The code above would associate the hidden field with the foo property of the ActionForm. If the ActionForm is in scope when the html:form renders, then the value of this field will be the value of the foo property of the ActionForm subclass. When the HTML form gets submitted, the foo property of the ActionForm subclass will be populated.

Tip: TIP: The problem I've seen developers have with the html:hidden tag is that they try to use it to pass all hidden fields, and then it errors out because the backing ActionForm subclass does not have that property. If you want to pass an arbitrary hidden field, just use the plain old <input type="HIDDEN" value="whatever"/> tag. I use this with the DispatchAction hidden field method discriminator, as I do not want the method discriminator as part of the ActionForm. This is really a matter of choice.

html:html

The html:html tag renders the <html> element. You can use the `locale` attribute (locale="true") to set the Locale object named by the HTTP Accept-Language. The xhtml attribute allows you to output to the browser using xhtml by producing the xml:lang attribute.

The `locale` attribute has been deprecated in Struts 1.2. Use the `lang` attribute in its place when you migrate to Struts 1.2. The `lang` attribute renders the locale stored in the user's session if found; otherwise, it uses the `lang` attribute from the Accept-Language HTTP header. If still not found, it uses the default locale of the server.

html:image

The html:image tag is used to render an image button. You can specify the image using the `page` or `src` attributes. The `page` attribute specifies the application-context relative path of the image source. The `src` attribute specifies a full URL location of the image source. It's interesting to note that this tag supports internationalization too. The `page` attribute and the `src` attribute have a corresponding `pageKey` attribute and `srcKey` attribute, which look up the image location per locale (language and/or country combination) in the resource bundle.

Internationalized images may not be an intuitive concept. However, different locales can have different iconography. Simply put, images mean different things in different places Another reason you'll want to internationalize your images is if the images contain text. The html:image tag supports a lot of the common attributes specified earlier, as well as the corresponding HTML attributes that you would expect. See the Struts online documentation for more details.

html:img

The html:img tag renders an <img> element. It is very similar to the html:image tag with the exception that this tag is not an input field. The html:img tag supports the same concepts of internationalizing as the html:image tag. It also supports all of the attributes that you would expect. See the Struts online documentation for more details. The html:img tag uses the same mechanism to specify request parameters as the html:link tag (covered below).

<html:javascript />

The html:javascript renders JavaScript validation methods. This tag is tightly coupled with the Validator framework. (See Chapter 3 for a usage example.) Using the html:javascript tag, you can, for example, turn on and turn off the HTML comments. You can also use dynamically generated JavaScript code or you can use static JavaScript code. Again, see the online Struts documentation for more details.

html:link

The html:link tag renders an HTML anchor tag (i.e., a hyperlink). This tag uses a lot of the same common attributes as described earlier.You have multiple options for rendering the URL of a hyperlink. You can use the `href`, `action`, `forward`, or `page` attributes to specify the URL. The `href` attribute is used to specify a full URL without any knowledge of the web context of this web application. The `page` attribute is used to specify a web context relative link. The `action` attribute is used to specify a link to an action mapping, as described in the Struts config file. The `forward` attribute is used to specify a link to a global forward, as described in the Struts config file.

Tip: TIP: If you are following a Model 2/MVC architecture, then you should use the `page` attribute and the `href` attribute sparingly. In fact, you should almost never use the `page` attribute. The `href` attribute should only be used to link to resources that are not in the current web application. This helps you separate the controller from the View by not letting the View select the next View directly. Only the controller should select the next View. Using the `action` and `forward` attributes instead forces you to delegate selection of the next View to the controller.

Here is an example of linking to an action (/html-link.do) with the `page` attribute:

```
<html:link page="/html-link.do">
     Linking with the page attribute.
</html:link>
```

Notice that you do not have to specify the web context of the web application. Conversely, if you used the `href` attribute, you would have to specify the web context as follows (where the context is struts-exercise):

```
<html:link href="/struts-exercise-taglib/html-link.do">
    Using Href
</html:link>
```

Obviously, it is better to use the `page` attribute when you are linking to things in the same web application (thus, the same context). You can also use the `href` attribute to create links that are not on the same server as follows:

```
<html:link
   href="http://otherserver/strutsTut/html-link.do">
        Using Href
</html:link>
```

Another way to link to the html-link.do action is to use the `action` attribute as follows:

```
<html:link action="/html-link">
  Using Action attribute
</html:link>
```

You can specify parameters by hard coding them as follows:

```
<html:link page="/htmllink.do?doubleProp=3.3&amp;longProp=32">
        Double and long via hard coded changes
</html:link>
```

Or, you can use the `paramId`, `paramName`, and `paramProperty` attributes as follows:

```
<html:link page="/html-link.do"
           paramId="booleanProperty"
           paramName="testbean"
           paramProperty="nested.booleanProperty">
  Boolean via paramId, paramName, and paramValue
</html:link>
```

The code above generates the query string as follows:

```
<a href="/struts-exercise-taglib/html-
   link.do?booleanProperty=false">
         Boolean via paramId, paramName, and paramValue</a>
```

The `paramId` attribute specifies the name of the parameter on the query string. The `paramName` attribute specifies the name of the JavaBean that is used to generate the value of the query string parameter. If you do not specify a paramProperty, then the toString value of the bean will be used. If you specify paramProperty, the value that the paramProperty will be used as the value of the query string parameter. You can also specify a `paramScope` where the link tag will look for the bean. If the scope tag is missing, the link tag looks for the bean and all the scopes starting with page scope (the innermost scope) and going to application scope (the outermost scope). The dot notation in the `property` attribute value means you are using a nested property (i.e., testbean.getNested().getBooleanProperty()).

In addition, you can pass multiple query string parameter properties by specifying a map with the `name` attribute demonstrated as follows:

```
<%
  java.util.HashMap newValues = new java.util.HashMap();
  newValues.put("floatProperty", new Float(444.0));
  newValues.put("intProperty", new Integer(555));
  newValues.put("stringArray", new String[]
   { "Value 1", "Value 2", "Value 3" });
  pageContext.setAttribute("newValues", newValues);
%>
...
```

```
            <html:link action="/html-link"
                       name="newValues">
              Float, int, and stringArray via name (Map)
            </html:link>
```

Notice that the Java scriptlet creates a HashMap and sticks the HashMap into page scope under the attribute `newValues`. The link tag links to an action and specifies the HashMap as its list of query string parameters with the `name` attribute. The code above would generate the following link:

```
<a href="/struts-exercise-taglib/html-
link.do?stringArray=Value+1&amp;stringArray=Value+2&amp;stringArray=Value
+3&amp;floatProperty=444.0&amp;intProperty=555">
Float, int, and stringArray via name (Map)</a>
```

Note: Many of the examples for this chapter are based on the WAR files that ship with the Struts package to demonstrate and test the use of the Struts tags. You can find these WAR files under the webapps directory of the Struts distribution.

You can on use the `transaction` attribute to generate a transaction token query string parameter based on the saved transaction token. To learn more about transaction tokens, read Chapter 5: Working with Actions.

html:messages

The html:messages tag displays a collection of messages. The messages can be represented as ActionErrors, ActionMessages, String, or String array objects.

To display the messages in the JSP, you use the html:messages tag as follows:

```
<ul>
<font color='green' >
<html:messages id="message" message="true">
  <li><%= message %></li>
</html:messages>
</font>
</ul>
```

The html:messages tag will iterate over all of the messages. Notice that the `message` attribute of html:messages is set to true. This forces html:messages to get the messages from Globals.MESSAGE_KEY in request scope. If you do not set the `message` attribute, the html:messages tag will display the errors instead (Globals.ERROR_KEY). The default is set to display error messages because that is what the tag is mostly used to do.

Another thing you may want to do is put the error message right next to the field that has the error. You can do that with html:messages as follows:

```
<html:text property='userName'/>
 <html:messages id="message" property='userName'>
 <font color="red">
      <%=message%>
 </font>
 </html:messages>
```

In the example above, the html:messages tag iterates over all of the error messages for the userName field.

If you like how html:errors specifies footer and headers in the resource bundle, you can do the same type of thing with html:messages as follows:

```
<html:messages property="property2" message="true" id="msg"
  header="messages.header" footer="messages.footer">
  <tr><td><%= pageContext.getAttribute("msg") %></td></tr>
</html:messages>
```

The code above uses the header and footer as keys to get messages out of the resource bundle. To get a list of all of the attributes that html:messages supports, see the online Struts documentation.

Note: As a best practice, you should use CSS, not the font tag. The font tag was used for simplicity.

html:multibox

The html:multibox custom tag is a little weird, and it's bit hard to understand at first. Basically, this tag is used to manage an array of check boxes.

The html:multibox tag works with an array of strings. Each element in the array represents a check box. The multibox tag works similar to the html:select tag. When a string is added to the array, the corresponding check box is checked. Conversely, if the string is missing from the array, the corresponding check box is unchecked.

Once the form is submitted, the array property of the action form that is associated with the html:multibox tag is populated with all the strings that correspond to the checked check boxes. Here is an example usage of the tag as follows:

```
<logic:iterate id="newsLetter"
               name="user"
               property="newsLetterChoices">

   <html:multibox property="selectedNewsLetters">
       <bean:write name="newsLetter"/>
   </html:multibox>
       <bean:write name="newsLetter"/>
</logic:iterate>
```

See the modified user registration form example at the end of this chapter for more details on usage.

html:select, html:option, html:options, and html:optionsCollection

The html:select tag renders a drop-down selection or a list box to the end user. You can specify the options for the html:select tag with the html:select, html:option, html:options, and html:optionsCollection tags.

The html:option tag generates an option for the html:select tag (<option>). The html:options tag is used to generate a group of HTML <option> elements inside of the html:select tag. The html:optionsCollection tag is also used to generate a list of HTML <option> elements from a collection of JavaBeans.

There is an example of using the html:select tag in the example application at the end of this chapter.

Here is an example of creating a single selection box:

```
<html:form action="html-select.do">
...
      <html:select property="singleSelect" size="10">
        <html:option value="Single 0">Single 0</html:option>
        <html:option value="Single 1">Single 1</html:option>
        <html:option value="Single 2">Single 2</html:option>
        <html:option value="Single 3">Single 3</html:option>
        <html:option value="Single 4">Single 4</html:option>
      </html:select>
```

The above would render a selection box that was 10 items long (size=“10”). Notice that values of the label are the bodies of the html:option tag. The value that gets submitted to the action is the `value` attribute. The corresponding ActionForm needs to have a property called singleSelect, which would be a string.

One potential problem with the code above is that it does not support internationalization. You can support internationalization as follows:

```
<html:select property="resourcesSelect" size="3">
<html:option value="Resources 0" key="resources0"/>
<html:option value="Resources 1" key="resources1"/>
     <html:option value="Resources 2" key="resources2"/>
</html:select>
```

Here is an example of creating a multiple that gets its values from one collection and its labels from another:

```
<html:select property="multipleSelect"
                  size="10" multiple="true">

  <html:options name="multipleValues"
                labelName="multipleLabels"/>
```

```
        </html:select>
```

Notice that it sets the `multiple` attribute to true, this tells the custom tag to render a select input that allows multiple items in the selection list to be selected. When you set multiple="true", you have to make sure that the corresponding property of the ActionForm subclass is an array type. The html:options tag uses two collections. The first collection is called multipleValues and is used to specify the values that are submitted to the action if the item is selected. The second collection is called multipleLabels, and it is used to populate the labels for the corresponding values in the first collection. There is more than one way to skin a cat:

```
        <html:select property="collectionSelect"
                     size="10" multiple="true">
          <html:options collection="options"
                        property="value" labelProperty="label"/>
        </html:select>
```

A variation of the html:options tag specifies a single collection with a `collection` attribute. Then, it specifies an attribute that will contain the value property, and an attribute that specifies the label property. The tag above will iterate over each item in the options collection and call item.getValue() for the value and item.getLabel() for the label. The property names can be any property names that the beans in the collection support.

Typically when you're accessing a bean in a certain scope in Struts, then you always use three attributes, namely: `name`, `property`, and `scope` (more on this when you cover the bean taglib). The html:options tag breaks all the rules when it comes to dealing with collection. However, the html:optionsCollection uses the standard trinity of attributes as follows:

```
        <html:optionsCollection name="testbean"
                                property="beanCollection"
                                scope="session"
                                label="label"
                                value="value"

                                />
```

The code above uses the trinity of attributes to grab the test bean out of session scope and uses its collection property called beanCollection. The collection has a list of JavaBeans of a certain type. The JavaBean has a label property and a value property, which are used by the `label` and `value` attributes to extract the label and values. (The property of the JavaBean does not have to be called label and value.)

html:password

The html:password tag renders a password field. The user registration from Chapter 1 has an example of using the html:password tag.

html:radio

The html:radio tag renders a radio check box.

html:reset

The html:reset tag renders a Reset button.

html:rewrite

The html:rewrite tag generates a URL that is encoded similar to what html:link does, except that this tag does not generate an anchor tag—just a URL. This tag is useful for client-side JavaScript function string constants. You can also specify transaction tokens with this tag. For more information, look at the html:link tag discussed earlier in this chapter, as they share a lot of the same attributes.

html:submit

The html:submit tag renders a Submit button. There was an example of this in Chapter 1, and just about every example in this book uses the html:submit tag.

html:text

The html:text tag renders a text input field. This tag has a read-only attribute that makes it a display only tag (i.e., not editable). You can also use the `size` attribute to limit how much of the field displays. You can use the `maxlength` attribute to limit how many characters the end user can enter into the field.

html:textarea

The html:textarea tag renders a text area field. You can specify the width of the text area by using the `cols` attribute. Similarly, you can specify the number of rows with the `rows` attribute.

html:xhtml

The html:xhtml tag tells the rest of tags in the page to render xhtml elements instead of html elements.

Bean Custom Tag Lib

The Bean tag library provides a group of tags to work with JavaBeans and define JavaBeans.

Before you can start this section, it needs to be clear that if you are working on green field applications, skip this section. (Green field applications are a new type of application without baggage.) Don't use this custom tag library on new projects. You should use JSTL on new projects. See Chapter 8 for more details.

Remember, once the tag library is available to the web application classloader, using a tag library is a two-step process. The first step is declaring that you are going to use the tag library in your web.xml file as follows:

```
<taglib>
  <taglib-uri>struts-bean</taglib-uri>
  <taglib-location>
      /WEB-INF/struts-bean.tld
  </taglib-location>
</taglib>
```

The second step is to declare that you're using the tag library in your JSP page as follows:

```
<%@ taglib uri="struts-bean" prefix="bean" %>
```

bean:cookie, bean:header, and bean:parameter

The bean:cookie, bean:header, and bean:parameter tags are used to retrieve the cookies, request headers, and request parameters. The bean:header and bean:parameter tags have an `id` attribute, so you can define a string attribute in page scope. The bean:cookie tag defines a Cookie object (from the Servlet API). You can specify a default value with the `value` attribute. If you're expecting multiple values to be returned (i.e., an array), then you set the `multiple` attribute to true. When you set the `multiple` attribute to true with bean:header and bean:parameter, a String[]will be defined instead of just a String. When you set the `multiple` attribute to true with bean:cookie, a Cookie[] will be defined instead of just a single Cookie object.

Tip: JSTL has an expression language that can access cookies, headers, and parameters; therefore, you do not need these tags if you are using JSTL. All new projects should use JSTL. Existing projects should migrate to JSTL.

```
<bean:cookie id="sessionID" name="JSESSIONID" value="JSESSIONID-IS-
UNDEFINED"/>
```

The code above defines a scriptlet variable called sessionID. The value of the sessionID scriptlet variable will be set to “JSESSIONID-IS-UNDEFINED” if the request does not have a cookie called JSESSIONID.

The following code would print out the properties of the Cookie object as follows:

```
<jsp:getProperty name="sessionID " property="comment"/> …
<jsp:getProperty name="sessionID" property="domain"/> …
<jsp:getProperty name="sessionID" property="maxAge"/> …
<jsp:getProperty name="sessionID" property="path"/> …
<jsp:getProperty name="sessionID" property="value"/> …
<jsp:getProperty name="sessionID" property="version"/> …
```

Here is an example that prints out all of the headers in the request:

```
<%
   java.util.Enumeration names =
     ((HttpServletRequest) request).getHeaderNames();
%>
…
<%
  while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
%>
    <bean:header id="head" name="<%= name %>"/>
    … <%= name %>
    … <%= head %>
    …
<%
  }
%>
```

Here are some bean:parameter examples:

```
<bean:parameter id="param1" name="param1"/>
<bean:parameter id="param2" name="param2" multiple="true"/>
<bean:parameter id="param3"
                name="param3" value="UNKNOWN VALUE"/>
```

The first example grabs a parameter called param1 and defines a scriptlet variable called param1 of type string. The second example grabs a parameter called param2 and defines a scriptlet variable called param2 of type string array. The third example grabs a parameter called param3, and if the parameter isn't present, it uses the default that is specified by the value entered.

bean:define

The bean:define tag allows you to copy a bean’s property from any scope into a bean in any scope. The bean:define tag uses the trinity of attributes defined in the section earlier in this chapter. It copies the object specified by the trinity of attributes to the scope specified by the `toScope` attribute. The name of the new attribute is specified by the `id` attribute.

Tip: Don't use bean define if you are your using JSTL. There's no point. It is better to use core:set. It's the suggestion of the Struts developers that where a tag overlaps with the JSTL, the tag is consider a future deprecated tag (Struts 1.2 or greater).

bean:include

The bean:include tag is similar to the jsp:include tag with the exception that it defines a scriptlet variable that you can reuse. The name of the new scriptlet variable is specified by the `id` attribute. The bean:include tag supports four other attributes: `forward`, `href`, `page`, and `transaction`. The attributes have the same meaning as they do in the html:link. (See the html:link section from earlier in this section for more details.)

Here's an example of how you can use the bean:include tag:

```
<bean:include id="footerSpacer"
             page="/long/path/footerSpacer.jsp"/>
```

Then you could use the footerSpacer in several places with the bean:write tag as follows:

```
<bean:write name="footerSpacer" />
```

bean:message

The bean:message tag is used to grab messages from the resource bundle. The `key` attribute specifies the key of the message in the resource bundle. The tag can pass up to four arguments to the message with parameters arg0 through arg3. (See the Java API docs for java.text.MessageFormat for more details about what the argument attributes do). You can also specify a new resource bundle using the `bundle` attribute. You can override the locale that is in session scope by using the `locale` attribute. See Chapter 14 for more details on how to use this tag for internationalization.

bean:page

The bean:page tag allows you define JSP implicit objects as scriptlet variables.

Here is a simple example that defines the request object as a scriptlet variable:

```
        <bean:page id="requestObj" property="request"/>
```

The example above retrieves the implicit request object and stores its reference in the scriptlet variable called requestObj.

bean:resource

The bean:resource tag grabs the raw value of resources in the web application context directory. This concept is best illustrated with an example:

```
        <bean:resource id="webxml" name="/WEB-INF/web.xml"/>
```

The code above defines a scriptlet variable called webxml based on the value of your web.xml deployment descriptor.

Now, to display the code above, you could use the bean:write tag as follows:

```
        <pre>
        <bean:write name="webxml" filter="true"/>
        </pre>
```

bean:size

The bean:size tag gets the count of items stored in an array, collection, or map, and stores the count in a scriptlet variable defined by the `id` attribute.

```
        <bean:size id="count" name="employees" />
```

The above would define a scriptlet variable called count based on the number of employees in the employees collection.

bean:struts

The bean:struts tag copies Struts objects into a paged, scoped scriptlet variable. This tag can retrieve ActionForms, ActionForward, or an ActionMapping object with the `formbean`, `forward`, or `mapping` attributes respectively.

Here's an example using a bean:struts tag:

```
        <bean:struts id="userRegistrationForm"
          formBean="UserRegistrationForm"/>
```

The code above grabs the UserRegistrationForm from the Struts config file.

bean:write

The bean:write tag gets the value of a named bean property. If the `format` or `formatKey` attribute is encountered, then the value being written will be formatted based upon the `format` attribute value for the format value that is stored in the resource bundle. You can specify a particular resource bundle with the `bundle` attribute. You can also specify a particular locale with the `locale` attribute. The `filter` attribute, if true, turns HTML reserved characters into entities (i.e., "<" becomes &lt;). This tag uses the trinity of attributes, namely, `name`, `property`, and `scope`. To learn more about the trinity of attributes, see the trinity of attributes section earlier in this chapter. The value specified by the trinity of attributes will be output to the JSPWriter. The `ignore` attribute states that if the object is missing, don't throw an exception (the default is false).

```
        <bean:write name="userRegistration"
                property="email"
                    scope="request"/>
```

The code above retrieves the `e-mail` attribute of the user registration object, which is in request scope.

The Logic Tag Lib

Logic Tag Library

The focus of the logic tag is to provide presentation layer focus. The logic tag library is on its way out.

> **Note:** The JSTL core:if tag does almost everything the logic tag library does in one tag. Try to use JSTL first on new projects. See Chapter 8 on JSTL for more details.

Remember, once the tag library is available to the web application classloader, using a tag library is a two-step process. The first step is declaring that you are going to use the tag library in your web.xml file as follows:

```
<taglib>
  <taglib-uri>struts-logic</taglib-uri>
  <taglib-location>
      /WEB-INF/struts-logic.tld
  </taglib-location>
</taglib>
```

The second step is to declare that you are using the tag library in your JSP page as follows:

```
<%@ taglib uri="struts-logic" prefix="logic" %>
```

logic:empty and logic:notEmpty

The logic:empty tag checks to see if a scriptlet variable is null, an empty string, collection, or map. The logic:notEmpty tag does the opposite (i.e., it checks to see if a scriptlet variable is not null, if it is a string, collection, or map). The logic:notEmpty tag also checks to see if it is not empty:

```
<logic:empty name="myBean">
  The bean is missing
</logic:empty>
<logic:notEmpty name="myBean">
  The bean is not missing
</logic:notEmpty>
```

The code above would print out “The bean is missing” if the bean named myBean is not in any scope. Conversely, it would print out “The bean is not missing” if the bean is found in any scope. Both of these tags use the trinity of attributes to access the object (`name`, `property`, and `scope`).

Tip: JSTL EL has an empty operator that functions much like this tag.

logic:equal, logic:notEqual, logic:lessThan, logic:greaterThan, logic:lessEqual, and logic:greaterEqual

The logic:*equal* tags do just what their names state. These tags are a bit cumbersome to use. They all use the trinity of attributes (`name`, `property`, and `scope`).

Here is an example using some of the tags above:

```
<logic:equal name="bean" property="doubleProperty"
       value="<%= doub1 %>">
  equal
</logic:equal>
<logic:greaterEqual name="bean" property="doubleProperty"
       value="<%= doub1 %>">
  greaterEqual
</logic:greaterEqual>
```

logic:forward

The logic:forward tag is similar to a jsp:forward tag except it uses an ActionForward from the global forwards section of the Struts config file. It specifies the name of the forward with the `name` attribute. Example as follows:

```
<logic:forward name="login" />
```

The code above refers to a forward in the global forwards area demonstrate of the Struts config file as follows:

```
<global-forwards>
  <forward name="login" path="/loginForm.jsp"/>
</global-forwards>
```

Tip: If you are following the Model 2/MVC architecture, the important thing to remember about logic:forward is to never use it. If you are following the Model 2/MVC architecture, then you always select the next View from the controller, not from the View. If you are not following the Model 2/MVC architecture, go back to the Model 1 world.

logic:redirect

The logic:redirect tag is similar to the last tag except by default it does a response.sendRedirect() instead of a requestDispatcher.forward(). Another key difference is it supports all the same attributes as the html:link, so you can pass request parameters and such. Here is an example using the logic:redirect tag and passing a request parameter:

```
<logic:redirect name="login"
                paramId="employeeId"
                paramName="employee" property="id" />
```

The code above redirects to the global forward login, which is defined in the global-forwards section of the Struts config file.

Tip: Again, if you are following the Model 2/MVC architecture, do not use the logic:redirect tag. If you are following the Model 2/MVC architecture, then you always select the next View from the controller, not from the View. If you're not following the Model 2/MVC architecture, you're losing some of the benefits of Struts.

logic:iterate

The logic:iterate tag iterates over a collection, enumerator, iterator, map, or arrayIt evaluates its body for each item in the collection. Here is an example:

```
<logic:iterate id="employee"
               name="department"
               property="employees"
               scope= "request">
…
      <bean:write name="employee" property="username" />
…
      <bean:write name="employee" property="name" />
…
      <bean:write name="employee" property="phone" />
…
</logic:iterate>
```

The code above iterates over a collection of employees. The collection of employees is based on the employees property of the department, which is a request scope. The `id` attribute specifies a scriptlet variable that is assigned the current item at the start of each iteration, namely employee.

To print out the fifth through the tenth employee, you would use the `length` and `offset` attributes as follows:

```
<logic:iterate id="employee"
               name="department"
```

```
                       property="employees"
                       scope= "request"
                       length="10"
                       offset="5">
…
      <bean:write name="employee" property="username" />
…
      <bean:write name="employee" property="name" />
…
      <bean:write name="employee" property="phone" />
…
</logic:iterate>
```

You may want to define a variable that holds the current iteration. Here is an example:

```
<ol>
<logic:iterate id="element"
               name="bean"
               property="stringArray"
               indexId="index">

       <li>
           <em>
            <bean:write name="element"/>
           </em>&nbsp;
           [<bean:write name="index"/>]</li>
</logic:iterate>
</ol>
```

The `indexId` attribute specifies a scriptlet variable called index, which holds that current index number of the loop. This is useful if you want to generate row numbers for long listings.

logic:match and logic:notMatch

The logic:match and logic:notMatch tags check to see if two strings have equal parts at the start of the string, at the end the string, or if any parts of the strings are equal. You can specify the string, which can be a cookie, header, request parameter, or bean property by using the attribute's cookie, header, parameter, or name and property. The `location` attribute specifies where the match occurs (start or end). If the `location` attribute is missing, then the match can occur anywhere in the string.

Here is an example that checks browser type:

```
<logic:match header="User-Agent" value="Mozilla">
    Mozilla!
  </logic:match>
  <logic:notMatch header="User-Agent" value="Mozilla">
    Not Mozilla :(
  </logic:notMatch>
</logic:present>
```

Here is another example that checks to see if a bean property matches the string “hello world”:

```
<logic:match name="bean"
             property="stringProperty"
             value="hello world">
  Hello World!
</logic:match>
<logic:notMatch name="bean"
                property="stringProperty"
                value="hello world">
  I'm so sad and lonely.
</logic:notMatch>
```

Tip: There's nothing in the JSTL library that is similar to the match or notMatch tags. Therefore, you may use this tag with reckless abandon. Remember, don't put too much application logic in your JSP page.

logic:present and logic:notPresent

The logic:present and logic:notPresent tags check to see if headers, request parameters, cookies, JavaBeans, or JavaBean properties are present and not equal to null. You can specify the string or object, which can be a cookie, header, request parameter, or bean property by using the attribute's cookie, header, parameter, or name and property. Additionally, you can check to see if the current users are in a certain role using the `roles` attribute. (This integrates with the J2EE security model.)

Tip: Both the logic:present and logic:notPresent tags fit really well in a Model 2/MVC architecture. You can allow the controller (i.e., your actions) to decide which objects to map into scope based on executing business rules of your application Model. The JSP page, which should have very little logic in it, will check to see if an object is in scope, and if it's in scope, then it displays the object. This way, if the underlying business rules of the Model change, it does not affect the JSP code. The JSP never knows why it's displaying an object—just if that object is present. (If the object is present, then it should display the object.) It's up to the controller to decide which objects to map into scope after executing business rules on the Model.

You can use JSTL to do the same thing as the logic:present and logic:notPresent tags. New projects should use the JSTL equivalent.

Struts Tutorial Example: Continued

Using a Multibox

In order to use a multibox tag, you must perform the following steps:

1. Create a list of available choices (a String array).

```
public class UserRegistrationForm extends ValidatorForm {
          ...
        private static final String [] newsLetterChoices = {
                "ArcMind Struts Journal",
                "ArcMind WebWork Journal",
                "ArcMind Spring Journal",
                "ArcMind JSF Journal"
        };

        public String[] getNewsLetterChoices() {
                return newsLetterChoices;
        }
...
```

   The newsletterChoices property states the available choices to the end user. The newsletterChoices property is not an input property; it is, in fact, a read-only property. Note that you are storing the choices property in the ActionForm as a convenience. You can put the available choices anywhere. For example, you might get the choices out of a database in the previous step.

2. Add an array property to your ActionForm subclass to hold the selected items.

```
public class UserRegistrationForm extends ValidatorForm {
...
        private String[] selectedNewsLetters =
                      new String []{"ArcMind Struts Journal"};


        public String[] getSelectedNewsLetters() {
                return selectedNewsLetters;
        }

        public void setSelectedNewsLetters(String[] strings) {
                selectedNewsLetters = strings;
        }
...
```

The property above selectedNewsletters represents the newsletters that the user selected. This is the backing property for the multibox tag. Notice that you set the default value so that the user selects the “ArcMind Struts Journal” by default.

3. Override the reset() method of your ActionForm subclass to reset the selected items property to an empty array.

```
public class UserRegistrationForm extends ValidatorForm {
...
     public void reset(ActionMapping mapping,
                       HttpServletRequest request) {
          log.trace("reset");
          selectedNewsLetters = new String []{};
     }
...
```

Since you are dealing with check boxes and HTTP/HTML only sends checked check boxes, you need to reset all of the check boxes to see which check boxes the end user selected. If you remember from the life cycle of the ActionForm discussion in Chapter 3, the reset() method gets called before the request processor populates the ActionForm. Thus, the reset() method is often used to set the check boxes to unchecked.

4. In your JSP, iterate over the available choices and render a multibox for each iteration.

```
<td>
Pick out some newsletters.
</td>
 <td>
 <logic:iterate id="newsLetter"
                name="user"
                property="newsLetterChoices">
         ... [Step 5 goes here]
        <bean:write name="newsLetter"/>
 </logic:iterate>
 </td>
```

The code above grabs your ActionForm subclass, which is mapped into the `user` attribute under session scope. Then, it takes the available choice’s property (newsLetterChoices) and iterates over it with the logic:iterate tag. For each iteration, it writes out the newsletter as the label of multibox control.

5. Associate the multibox with the selected item's property of the ActionForm subclass.

```
<td>
Pick out some newsletters.
</td>
 <td>
 <logic:iterate id="newsLetter"
                name="user"
                property="newsLetterChoices">

        <html:multibox property="selectedNewsLetters">
             <bean:write name="newsLetter"/>
        </html:multibox>
        <bean:write name="newsLetter"/>
 </logic:iterate>
 </td>
```

For each iteration, the code above uses a multibox that's associated with the property selectedNewsletters that renders a check box. The following HTML is rendered from the code above:

```
<td>
Pick out some newsletters.
</td>
 <td>
    <input type="checkbox"
           name="selectedNewsLetters"
           value="ArcMind Struts Journal"
           checked="checked">
        ArcMind Struts Journal


    <input type="checkbox"
           name="selectedNewsLetters"
           value="ArcMind WebWork Journal">
        ArcMind WebWork Journal


    <input type="checkbox"
           name="selectedNewsLetters"
           value="ArcMind Spring Journal"
           checked="checked">
        ArcMind Spring Journal


    <input type="checkbox"
           name="selectedNewsLetters"
           value="ArcMind JSF Journal"
           checked="checked">
        ArcMind JSF Journal

 </td>
```

In this example, you are going to continue the user registration form. You're going to add the ability for the end user to select newsletters from a list of newsletters with corresponding check boxes.

Using a Check Box

In order to use a checkbox tag, you must perform the following steps:

1. Add a boolean property to your ActionForm subclass to hold the selected item.
2. Override the reset() method of your ActionForm subclass to reset the selected item property to false.
3. Associate the checkbox tag with the selected item property.

Based on the above steps, here is the ActionForm subclass code:

```
public class UserRegistrationForm extends ValidatorForm {

      private boolean yearlySubscription;

      public boolean isYearlySubscription() {
            return yearlySubscription;
      }

      public void setYearlySubscription(boolean b) {
            yearlySubscription = b;
      }


      public void reset(ActionMapping mapping,
                        HttpServletRequest request) {
            log.trace("reset");
            ...
            yearlySubscription = false;
      }
```

The property that backs the check boxes is yearlySubscription. Notice that this property is reset in the reset() method. Just as before, you are dealing with a check box, and HTTP/HTML only sends checked check boxes. Therefore, you need to reset this check box to see if the user selected it.

Then, the JSP page for the code above would be as follows:

```
<tr>
  <td>
    Yearly subscription
  </td>
  <td>
    <html:checkbox property="yearlySubscription" />
  </td>
</tr>
```

Using a Select Tag

In order to use a select tag, you should perform the following steps:

1. Create a collection of available choices (a bean array).

   You need to create some JavaBeans and a collection; your real Model should be rife with these objects, like JobCategory, as follows:

```
public class UserRegistrationForm extends ValidatorForm {

      public static class JobCategory implements Serializable{
            private String name;
            private int id;
            public JobCategory(){}
            public JobCategory(int id, String name){
                  this.id = id;
                  this.name = name;
            }

            public String getName() {return name;}
            public void setName(String string) {name = string;}
            public int getId() {return id;}
            public void setId(int i) {id = i;}
      }

      static Collection categories = new ArrayList();
      static {
            categories.add(new JobCategory(1, "VP"));
            categories.add(new JobCategory(2, "CTO"));
            categories.add(new JobCategory(3, "Marketing"));
            categories.add(new JobCategory(4, "Developer"));
            categories.add(new JobCategory(5, "Architect"));
      }

      public Collection getJobCategories(){return categories;}
```

   The code above defines an inner class called JobCategory and then creates a collection of JobCategories.

   JobCategories is a property that states the available choices to the end user just like the multibox example, except this time the list of choices is not a simple String array. The JobCategories collection property is not an input property; it is, in fact, a read-only property. Note that you are storing the choices' properties in the ActionForm as a convenience for this example. You can put the available choices anywhere in your Model. The choices are unlikely to be in your ActionForm subclass if the choices are dynamic. For example, you might get the choices out of a database in the previous step and populate the JobCategories collection.

2. Add an array property to your ActionForm subclass to hold the selected items.

```
public class UserRegistrationForm extends ValidatorForm {
...
    int [] selectedJobCategories;

      public int[] getSelectedJobCategories() {
            return selectedJobCategories;
      }

      public void setSelectedJobCategories(int[] is) {
            selectedJobCategories = is;
      }
...
```

The property above, selectedJobCategories, represents the JobCategory property that the user selected. This is the backing property for the select tag.

3. Associate a select tag with the ActionForm property that holds the selected items.

```
<html:select  property="selectedJobCategories"
              multiple="true"
              size="5">
           [Step 4 ...]

</html:select>
```

4. Use html:optionsCollection to generate list items.

```
<html:select  property="selectedJobCategories"
              multiple="true"
              size="5">

     <html:optionsCollection name="user"
                             property="jobCategories"
                             value="id"
                             label="name"/>

</html:select>
```

The optionsCollection tag grabs the ActionForm subclass, which is stored under the `user` attribute. The tag then uses the property attribute to grab the collection of JobCategories from the ActionForm subclass. The tag iterates over the JobCategories collection and for each iteration, it grabs the id of the current JobCategory for the value of the option element and grabs the name property of the JobCategory for the label of the option element. The code above would render the following HTML code:

```
<td>
 Select all of the job categories that apply to you
 </td>
  <td>

     <select name="selectedJobCategories"
             multiple="multiple"
             size="5">
       <option value="1" selected="selected">VP</option>
       <option value="2">CTO</option>
       <option value="3">Marketing</option>
       <option value="4"
               selected="selected">Developer</option>
       <option value="5">Architect</option></select>

  </td>
```

In this example, you are going to continue the user registration form. You're going to add the ability for the end user to select job categories from a collection of job categories.

Summary

This completes this chapter on the Struts custom tags. First, you learned the importance of custom tags. You also learned how to install and configure custom tags. Next, you learned how to use common Struts tag attributes. Then you covered a tag-by-tag explanation of all the tags in the Struts tag libraries with tips on how to use them (or not use them). Finally, you took this knowledge and applied it to your user registration application to create check boxes, multiboxes, and select controls.

Chapter 8

JSTL and Struts-EL

Using JSTL in Place of Struts-EL Core Tags

This chapter covers the combination of the JSP Standard Tag Library (JSTL) and Struts. Struts pushed JSP custom tags to the limit. JSTL extends many of the concepts found in Struts logic tags. In many ways, JSTL is the natural successor to the Struts tag libraries. You may wonder why a book on Struts mentions JSTL. Quite simply, JSTL tags take the place of many Struts tags. The Struts development team recommends that you use JSTL tags in place of Struts tags when there is an overlap.

Introduction to JSTL

JSTL is a collection of custom tags libraries. It provides common functionality that many web applications need. JSTL provides support for presentation logic, formatting, XML support, and database access.

JSTL 1.0 is made up of four custom tag libraries: core, format, XML, and SQL. The core tag library is similar to the Struts bean and logic taglib in that it provides custom actions to manage data through scoped variables, as well as iteration and logic based on the page content. The core is also similar to the Struts HTML library in that it provides tags to generate and operate on URLs for URL rewriting and session management.

The format taglib formats data based on the classes in java.text. The format taglib has tags for formatting numbers and dates. It also provides support for i18N similar to the Struts bean:message custom tag in the bean taglib.

The xml taglib has tags to manipulate XML via XPath and XSL-like support. The sql taglib defines actions to manipulate and update relational databases. Neither of these tag libraries will be covered in this chapter as they do not fit well in the MVC model.

Easy Expression Language

One of the limitations of JSP 1.2 and earlier is that it relies heavily on Java scriptlets and Java expressions, which do not fit in the scripting and templating model that JSP espouses.

For example, you have an employee object in request scope, and you would like to print out her first name. You could use the following Java expression:

```
<%=((Employee)request.getAttribute("employee")).getFirstName()%>
```

You could rewrite the code above for clarity as follows:

```
<%
 Employee employee (Employee) = request.getAttribute("employee");
 out.println( employee.getFirstName());
%>
```

If you are working in a mixed group of page designers and Java coders, the expression and Java scriptlet above is bound to make the page designers a bit queasy. The JSTL approach to do the above is as follows:

```
<c:out value="${employee.firstName}"/>
```

The code above is not much different from what you can already do with the Struts bean tag library. You could do the above with bean:write. Here is the equivalent using bean write.

```
<bean:write name="employee" property="firstName"/>
```

When there is overlap between the Struts approach and the JSTL approach, the recommendation is to use the JSTL approach.

Unlike bean:write, you can use many expressions in the same string:

```
<core:out value="Employee Record: ${employee.firstName}
${employee.lastName}"/>
```

Compare the code above to the bean:write version below:

```
Employee Record: <bean:write name="employee"
                                       property="firstName"/>
 <bean:write name="employee" property="lastName"/>
```

Having an expression language is handy. The EL syntax is like a combination of JavaScript and XPath. With the JSTL expression language (JSTL-EL), you can look up beans and their properties, and perform operations on them. Attribute values for JSTL custom tags can be specified using the easy-to-use expression language. The JSTL-EL is a mini-language. JSTL-EL provides identifiers, accessors (bean getter methods), and operators. With EL, you can retrieve and manipulate values in JSP scopes (session, request, page, application), headers, and parameters.

EL expressions are delimited using a dollar sign ($) and curly braces ({}) as follows:

```
<c:out value="${employee.firstName}"/>
```

The code above prints out the firstName property of the employee bean. Since a scope is not specified, it searches all of the scopes starting with page, followed by request, session, and application scope.

The EL allows you to do some things similar to what you can do with JavaScript. Examine the following c:set, which is similar to bean:define:

```
<c:set var="product"
   value="${param['dailySalary'] * param['numberOfDays']} "
scope="request"/>
```

The code above defines a bean called product by multiplying the numeric equivalent of the request parameters dailySalary and numberOfDays, and then places the resulting product into the request scope. The dailySalary and numberOfDays request parameters are both strings, so the code above converts them to numbers and then multiplies them. The code above would be equivalent to:

```
<%
int dailySalary = Integer.parseInt(request.getParameter("dailySalary"));
int numberOfDays = Integer.parseInt(request.getParameter("numberOfDays"));
int product = dailySalary * numberOfDays;
request.setAttribute("product",new Integer(product));
%>
```

The JSTL-EL language is easy to learn, especially if you have a JavaScript background. The language has the following built-in implicit objects.

Table 8.1: JSTL-EL Built-in Implicit Objects

Object	Function
pageScope	Map for page scope
requestScope	Map for request scope attributes
sessionScope	Map for session scope
applicationScope	Map for application scope
param	Map for request parameters
paramValues	Map stores request parameters with multiple values
header	Map for header values
headerValues	Map stores header values with multiple values
cookie	Map for cookies
initParam	Map stores servlet context initialization parameters
pageContext	PageContext for the current page

Most of the parameters are Map parameters. Map parameters hold name value pairs (like a HashMap). The param example you covered earlier used the Map parameter.

A Quick Tour of the JSTL Tags

To get you started with JSTL, here's a quick tour of some of the JSTL tags that you will use often.

Defining Beans with JSTL core:set

Core:set defines beans with the following syntax:

```
<core:set var="name" scope="scope" value="expression"/>
```

Define the bean companyName value, Intel, in session scope:

```
<core:set var="companyName" scope="session" value="Intel"/>
```

The core:set custom tag defines beans similar to the Struts custom tag bean:define. Even though it is somewhat similar to bean:define, it is not the same, since the core:set tag has the power of the expression language.

A More Complex Example of core:set

Define the bean sum by multiplying the numeric equivalent of the request parameters dailySalary and numberOfDays, and then put the product in scope request:

```
<core:set var="product"
   value="${param['dailySalary'] * param['numberOfDays']} "
   scope="request"/>
```

The expression param['dailySalary'] is equivalent to request.getParameter("dailySalary"). The param object is built in the JSTL object that refers to request parameters. It is like a HashMap of request parameters. There are many other built-in objects.

Iterating Over Collections with core:forEach

You should use core:forEach when iterating over collections and arrays, as shown in the following example:

```
<core:forEach var="name"
                 items="expression">
          body content
</core:forEach>
```

Here is an example of iterating over results:

```
<core:forEach items="${results}" var="currentCD">
    <core:out value="${currentCD.ID}"/> <br />
</core:forEach>
```

The core:forEach tag can be used in place of logic:iterate.

Compare this code:

```
<core:forEach items="${results}" var="currentCD">
    <core:out value="${currentCD.ID}"/> <br />
</core:forEach>
```

to this code:

```
<logic:iterate name="results" id="currentCD" scope="request">
     <jsp:getProperty  name="currentCD" property="ID"/> <br />
</logic:iterate>
```

After comparing the two code sets, you may think that you could probably get away with never using logic:iterate again. And you would be right, except for indexed fields. Form control tags (html:text, html:radio, etc.) that are used in the context of the logic iterate tag can specify in the logic iterate tag that indexed is equal to true. This feature creates indexed fields. See Chapter 7 for more details.

The core:if Tag

The core:if tag checks to see if an expression is true. It has the following syntax:

```
<core:if
    test="expression"
    var="name"
    scope="scope">
          body content
</core:if>
```

Tip: Notice the var attribute name. The var attribute is used to create a variable that holds the results of the test expression. This allows you to avoid executing a long expression over and over again. You can store the results the first time and then use the results for future core:if tags in the page.

When you combine the core:if tag with expression syntax (JSTL-EL), the core:if tag takes the place of logic:equal, logic:notEqual, logic:greaterEqual, logic:lessEqual, logic:greaterThan, logic:lessThan, logic:present, and so on.

Here is an example using the core:if tag with expression syntax:

```
<core:if test="${bean.prop=1}">
      equal
</core:if>

<core:if test="${bean.prop>mydouble}">
      greater
</core:if>
```

The first core:if tag outputs "equal" if bean.getProp() is equal to 1. The second core:if tag outputs "greater" if bean.myProp() is greater than mydouble. You can see that having the expression language simplifies the amount tags that you need.

URL Rewriting with core:url

URL rewriting and can be done with core:url. This is similar to html:rewrite, but much more powerful because core:url uses the EL. Like html:rewrite, it performs URL rewriting and session id encoding if needed.

Here is the syntax of the core:url tag.

```
<core:url
          value="expression"
          context="expression"
          var="name"
          scope="scope">

        <core:param
             name="expression"
             value="expression"/> ...

</core:url>
```

Another thing that makes core:url better than html:rewrite is the fact that you can specify parameters using expressions, and the parameters are nested tags of the core:url tag. You can specify as many parameters as you like—each using the JSTL-EL.

Here is a simple example:

```
<a href="<core:url value='/index.jsp'/>">Home Page</a>
```

Formatting Numbers with the Format Taglib

You can use the format tags to provide i18n-based formatting. Format tags rely on I18n Java classes in java.text (MessageFormat, SimpleDateFormat, etc.).

For example, say that you have a product object and you want to format the product object's price property. The following formats the product's price as currency in the current locale:

```
<!-- output List Price with format:formatNumber -->

<format:formatNumber value="${product.price}" type="currency"/>
```

If you are in the US, the code above formats the price using the US dollar sign and the correct decimal points. If you are in merry old England, the code above formats the price using the British Pound sign.

Formatting Dates with the Format Tag

In a similar vein to the last section, you can format dates. For example, to format a product's expirationDate, you could use the pattern yyyy-MM-dd as follows:

```
<format:formatDate value=
    "${cd.releaseDate}" pattern="yyyy-MM-dd"/>
```

The date patterns are based on the SimpleDateFormat class.

Struts-EL

The Struts-EL tag handlers subclass the Struts tag handlers in most cases. The Struts-EL tag handlers enable the base Struts tag handlers to understand EL expressions. In the past, I have used Struts-EL on projects. I was surprised how productive it made the projects. Struts-EL easily solves problems that were considered very difficult to solve or didn't have an easy solution in plain Struts. Here's an example for you to study.

Before we can start this example, we need to install the Struts-EL tag libraries. Go to the Struts distribution directory (e.g., C:\tools\jakarta-struts-1.1\).

Navigate to the contrib\struts-el\lib directory (C:\tools\jakarta-struts-1.1\contrib\struts-el\lib).

Copy the following files to the WEB-INF directory of your project:

- struts-bean-el.tld
- struts-html-el.tld
- struts-logic-el.tld
- c.tld
- fmt.tld

The step above includes the deploy descriptors for the custom tags you are going to use. Recall that the TLD files have the mappings from the tag names to the tag handler classes.

Copy the following files to the WEB-INF/lib directory of your project:

- jstl.jar
- standard.jar
- struts-el.jar

The step above ensures that the classes that the tag handlers need (including tag handler classes) can be found by the classloader of the web application.

Give the tag libraries a short name so it's easy to work with them from JSP files. To do this, open up the web deployment descriptor (web.xml) and add the following entries:

```
<taglib>
  <taglib-uri>/tags/struts-bean-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean-el.tld
  </taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-html-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-html-el.tld
  </taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-logic-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/jstl-format</taglib-uri>
  <taglib-location>/WEB-INF/fmt.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/jstl-core</taglib-uri>
  <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>
```

The code above imports the html-el, bean-el, and logic-el from the Struts-EL package. It also imports the core tag library and the format tag library of JSTL. The next step is to start to use these.

Do you remember the admin user form JSP file? It had some pretty ugly Java code in it. Open up adminUsersForm.jsp, and look for the following code:

```
<logic:iterate id="lineItem" indexId="index"
         name="adminUsersForm" property="usersList">
  <tr>
    <td>

    <%
     String slineItem = "user[" + index + "]";
     String nest= slineItem + ".user.";

     String checked = slineItem + ".checked";
     String firstName = nest + "firstName";
     String lastName = nest + "lastName";
     String email = nest + "email";
     System.out.println(firstName);
    %>
      <html:text property="<%=firstName%>" />

    </td>
    <td>
      <html:text property="<%=lastName%>" />
    </td>
    <td>
      <html:text disabled="true" property="<%=email%>" />
    </td>
    <td>
     <html:checkbox property="<%=checked%>" />
    </td>
  </tr>
 </logic:iterate>
```

The code above is created to get at a nested JavaBean array. Review the Chapter 3 (ActionForms) example for more details. You are now going to rewrite that example with Struts-EL.

The first step is to change the HTML tag library import to a Struts html-el tag library import:

Change the line that reads:

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>
```

to:

```
<%@ taglib uri="/tags/struts-html-el" prefix="html"%>
```

Now you can get rid of that nasty Java scriptlet block by using a regular expression inside of the html:text and html:checkbox as follows:

```
        <logic:iterate id="lineItem" indexId="index"
                       name="adminUsersForm"
                       property="usersList">
          <tr>
            <td>
            <html:text property="user[${index}].user.firstName" />
            </td>
            <td>
            <html:text property="user[${index}].user.lastName" />
            </td>
            <td>
              <html:text disabled="true"
                   property="user[${index}].user.email" />
            </td>
            <td>
             <html:checkbox property="user[${index}].checked" />
            </td>
          </tr>
         </logic:iterate>
```

Not only is this code shorter than what you had before, it's easier to read. And it has no Java code in it. You cannot do the above with core:forEach!

Goodbye bean:*, Goodbye logic:*; It Was Nice Knowing You!

Hopefully, from what you learned in this chapter, you realize that using Struts-EL is going to make your JSP development much easier. Many of the bean:tags have vanished, as there is usually no need to use them once you commit to using JSTL and Struts-EL. Only three bean tags remain: bean:include, bean:resource, and bean:size. The only tags that you really need from logic are logic:match and logic:iterate. Logic present and all of the empty (notEmpty) tags are no longer needed because JSTL has keywords to see if an object is present or empty (i.e., not obj, and empty collection).

Summary

This chapter began by covering the basics of JSTL. Then, you learned how to start using JSTL and Struts-EL on your web project by including the right JAR files and TLD files. Finally, this chapter demonstrated how to use Struts-EL and JSTL to make your development simpler and aid in removing Java code from your web applications.

The key thing to remember about JSTL and Struts-EL is that they are the future of Struts. The Struts 1.1 tags are not deprecated in this release, but may be deprecated in the near future. In addition, the JSTL and Struts-EL tags are much easier to use because of the expression language. The EL makes your JSP coding easier.