



Struts 2

MVC, Struts 2.0/1.x, Actions, Interceptors, OGNL,
Generic and UI Tags, Results, Validators, Internationalization,
Struts 2 Plugins, AJAX with Struts 2,
Freemarker, Velocity, Tiles 2, Type Conversion

Black Book™

Second Edition

Kogent Solutions Inc.

Published by:



©Copyright by Dreamtech Press, 19-A, Ansari Road, Daryaganj, New Delhi-110002

Black Book is a trademark of Paraglyph Press Inc., 2246 E. Myrtle Avenue, Phoenix Arizona 85202, USA exclusively licensed in Indian, Asian and African continent to Dreamtech Press, India.

This book may not be duplicated in any way without the express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review. The information contained herein is for the personal use of the reader and may not be incorporated in any commercial programs, other books, databases, or any kind of software without the written consent of the publisher. Making copies of this book or any portion for any purpose, other than your own, is a violation of copyright laws.

Limits of Liability/disclaimer of Warranty: The author and publisher have used their best efforts in preparing this book. The author makes no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

ISBN: 10: 81-7722-870-6
13: 978-81-7722-870-0

ISBN: 978-93-5004-792-7 (ebk)

Edition: 2008

Printed at: Himal Impressions, New Delhi



Contents at a Glance

Introduction	xvii
Chapter 1: Struts 2 Advancement in Web Technology.....	1
Chapter 2 : Creating Application in Struts 2.....	23
Chapter 3 : Creating Action in Struts 2	41
Chapter 4 : Implementing Interceptors in Struts 2.....	91
Chapter 5 : Manipulating Data with OGNL in Struts 2	163
Chapter 6 : Controlling Execution Flow with Tags in Struts 2.....	193
Chapter 7 : Designing User Interface in Struts 2.....	231
Chapter 8 : Controlling Results in Struts 2.....	277
Chapter 9 : Performing Validations in Struts 2.....	315
Chapter 10 : Performing Interationlization in Struts 2	377
Chapter 11 : Using Plugins in Struts 2.....	405

Table of Glance

Chapter 12 : Securing the Struts 2 Application.....	449
Chapter 13 : Testing the Struts 2 Application	483
Appendix A	503
Appendix B	521
Appendix C	549
Appendix D	575
Appendix E	613
Appendix F	623
Index	653



Table of Contents

Introduction	xvii
Chapter 1	
Struts 2 Advancement in Web Technology	1
Overview of Web Applications.....	3
Hypertext Transfer Protocol (HTTP).....	5
Java Servlet Specification	6
Web Application Framework.....	8
WebWork2	10
Struts 2.....	12
Comparing Struts 1 and Struts 2.....	20
Similarity between Struts 1 and Struts 2	20
Difference between Struts 1 and Struts 2	20
Chapter 2	
Creating Application in Struts 2	23
Setting Struts 2 Environment	25
Platform Requirements.....	25
Getting Struts 2 APIs	26
Directory Structure.....	26
Developing First Application in Struts 2	28
Creating JSP pages	30

Table of Contents

Creating Action Class	32
Configuring Actions in struts.xml	33
Configuring Struts 2 in web.xml	34
Deploying Struts 2 Applicaiton	35
Running Struts 2 Application.....	37

Chapter 3

Creating Actions in Struts 2..... **41**

In Depth **43**

Handling Struts 2 Actions	43
Action Interface	45
ActionSupport Class.....	45
Action Mapping	49
Default Action.....	51
Action Methods	51
ActionContext Class.....	53
Using POJO as Action	57
Using Actions as ActionForms.....	58
Using ModelDriven Actions	59
Understanding Dependency Injection and Inversion of Control.....	61
Struts 2 Aware Interfaces	62
ApplicationAware Interface	62
SessionAware Interface	63
ParameterAware Interface	64
ServletRequestAware Interface	64
ServletResponseAware Interface.....	65

Immediate Solutions **67**

Developing a Hello Action Example.....	67
Creating HelloAction Class.....	68
Configuring HelloAction	69
Using Action as ActionForm.....	72
Creating UserAction	72
Creating GetUserAction	78
Configuring Action Class Method.....	82

Table of Contents

Creating LoginAction with ApplicaitonAware and SessionAware Interface	84
Creating LogoutAction – a POJO Action.....	88

Chapter 4

Implementing Interceptors in Struts 291

<i>In Depth</i>	93
Understanding Interceptors	93
Interceptors as RequestProcessor.....	94
Configuring Interceptors	95
Interceptor Stacks.....	95
Default Interceptor Configuration-struts-default.xml	96
Implementing Interceptors in Struts 2	100
Alias Interceptor.....	101
Chaining Interceptor.....	103
Checkbox Interceptor.....	104
Conversion Error Interceptor	104
Create Session Interceptor.....	105
Debugging Interceptor	106
Execute and Wait Interceptor	106
Exception Interceptor.....	107
File Upload Interceptor	109
I18n Interceptor	110
Logger Interceptor	111
Model-Driven Interceptor	112
Parameters Interceptor	114
Prepare Interceptor	115
Roles Interceptor	116
Scope Interceptor.....	117
Scoped-Model-Driven Interceptor	118
Servlet-Config Interceptor.....	119
Static Parameters Interceptor.....	120
Message Store Interceptor.....	121
Timer Interceptor	122
Token Interceptor	123
Token Session Interceptor	124
Validation Interceptor	125

Table of Contents

Workflow Interceptor	126
Writing Custom Interceptors	128
Immediate Solutions	131
Implementing Interceptors.....	131
Interceptor Example 1.....	133
Interceptor Example 2.....	138
Interceptor Example 3.....	148
Interceptor Example 4.....	154
Interceptor Example 5.....	157
Chapter 5	
Manipulating Data with OGNL in Struts 2	163
In Depth	165
Understanding Object Graph Navigation Language (OGNL)	166
Syntax of OGNL	167
Literals and Operators	168
Indexing.....	169
Method Call	169
Variable References.....	170
Chaining OGNL Sub-Expressions	170
Understanding Basic Expression Language Features.....	171
Accessing Bean Properties	171
Accessing the OGNL Context and ActionContext.....	172
Working with Collections	173
Support for Value Stack.....	180
Accessing Static Property and Methods.....	181
Using OGNL in Struts 2	181
Immediate Solutions	184
Creating an Struts 2 application with OGNL.....	184
JavaBean – The Product Class	184
AddAction Class	185
Creating index.jsp Page	187
Configuring the Application.....	189

Table of Contents

Running the Application.....	190
Chapter 6	
Controlling Execution Flow with Tags in Struts 2.....	193
<i>In Depth</i>	195
Implementing Generic Tags	195
Control Tags.....	195
Data Tags.....	202
<i>Immediate Solutions</i>	215
Using Control Tags with Data Tags	215
Using if, elseif and else Tags.....	217
Using Iteration Tags.....	221
Using Other Data Tags	226
Chapter 7	
Designing User Interface in Struts 2	231
<i>In Depth</i>	233
Implementing UI Tags	233
Form UI Tags	235
Non-Form UI Tags	256
Discussing Common Attributes in UI Tags.....	261
General Attributes.....	261
JavaScript-Related Attributes	262
Template-Related Attributes	263
Tooltip-Related Attributes	263
<i>Immediate Solutions</i>	265
Using UI Tags.....	265
Developing index.jsp	265
Developing personal_info.jsp	266
Developing PersonalAction.java	268

Table of Contents

Developing general_info.jsp	272
Developing GeneralInfoAction.java	274
 Chapter 8	
Controlling Results in Struts 2	277
<i>In Depth</i>	279
Understanding Results.....	279
Result Interface	279
ResultConfig Class	280
ResultTypeConfig Class	281
Configuring Results.....	282
Configuring Result Types	282
Result Elements	283
Global Results	284
Implementing Different Result Types in Struts 2.....	285
Chain Result.....	287
Dispatcher Result	288
FreeMarker Result.....	292
HttpHeader Result.....	293
Redirect Result.....	294
Redirect Action Result.....	295
Stream Result.....	296
Velocity Result.....	297
XSLT Result.....	298
PlainText Result.....	300
<i>Immediate Solutions</i>	301
Developing project_results Applicaiton.....	301
Configuring different Results in Application.....	304
Building Your Own Result Type	311

Chapter 9

Performing Validations in Struts 2 315

In Depth 317

Understanding XWork Validation support in Struts 2.....	317
Understanding Validators in Struts 2	317
RequiredFieldValidator Class.....	319
RequiredStringValidator Class.....	320
IntRangeFieldValidator Class.....	320
DoubleRangeFieldValidator Class.....	321
DateRangeFieldValidator Class.....	322
ExpressionValidator Class	323
FieldExpressionValidator Class	323
EmailValidator Class	324
URLValidator Class	325
VisitorFieldValidator Class.....	325
ConversionErrorFieldValidator Class	326
StringLengthFieldValidator Class.....	327
RegexFieldValidator Class	328
Defining Validators Scopes	329
Field Validator.....	329
Non-Field Validator.....	330
Registering Validators.....	331
Creating Custom Validators.....	332
Defining Validation Rules	333
Per Action Class	333
Per Action Alias.....	334
Short-Circuiting Validations	334
Implementing Validation Annotations	335
ConversionErrorFieldValidator Annotation.....	335
DateRangeFieldValidator Annotation.....	335
DoubleRangeFieldValidator Annotation	336
EmailValidator Annotation.....	336
ExpressionValidator Annotation.....	337
FieldExpressionValidator Annotation.....	337
IntRangeFieldValidator Annotation	337

Table of Contents

RegexFieldValidator Annotation	338
RequiredFieldValidator Annotation.....	338
RequiredStringValidator Annotation.....	338
StringLengthFieldValidator Annotation.....	339
StringRegexValidator Annotation.....	339
UrlValidator Annotation	340
Validation Annotation.....	340
Validations Annotation	341
VisitorFieldValidator Annotation	342
CustomValidator Annotation.....	343
 Immediate Solutions	 344
Creating Basic Validation Example.....	346
Creating JSP Pages	347
Creating AddClientAction Class.....	348
Configuring AddClientAction Action.....	349
Creating AddClientAction-validation.xml File.....	349
Using Field Validators	351
Creating JSP Pages	351
Creating AddEmployeeAction Class.....	353
Configuring AddEmployeeAction Action.....	354
Creating AddEmployeeAction-validation.xml File.....	355
Using Non-Field Validators	359
Creating JSP Pages	359
Creating ChangePasswordAction Class.....	360
Configuring ChangePasswordAction Action.....	361
Creating ChangePasswordAction-validation.xml File.....	362
Performing Validation Using Annotation	364
Creating JSP Pages	364
Creating Action Classes.....	367
Configuring Actions in struts.xml	371
 Chapter 10	
 Performing Interationlization in Struts 2.....	 377
 In Depth	 379

Table of Contents

Understanding Internationalization.....	379
Localization.....	381
Resource Bundles	385
Implementing Tags for Localization	387
<s:text>Tag.....	387
<s:i18n> Tag.....	388
<s:property> Tag.....	389
<s:textfield> Tag.....	389
<s:submit> Tag	390
<s:reset> Tag.....	390
Implementing I18n Interceptor	391
I18n Interceptor Methods.....	392
Immediate Solutions	395
Developing Strutsi18nApp Application	395
Using Internationalization Tags	395
Preparing Resource Bundles.....	399
Configuring Strutsi18nApp Application.....	401
Chapter 11	
Using Plugins in Struts 2	405
In Depth	407
Understanding Plugin.....	407
Implementing Plugins in Struts 2	408
The Codebehind Plugin.....	409
The Config Browser Plugin.....	411
The JasperReports Plugin.....	412
The JFreeChart Plugin	415
The JSF Plugin.....	415
The Pell Multipart Plugin.....	418
The SiteGraph Plugin.....	420
SiteMesh Plugin.....	422
The Spring Plugin	425
The Struts 1 Plugin.....	428
The Tiles Plugin.....	430

Table of Contents

Implementing Third Party Plugins	431
Groovy Standalone Plugin.....	432
The Spring MVC Plugin.....	433
 <i>Immediate Solutions</i>	435
Implementing Codebehind Plugin.....	437
Implementing Struts 1 Plugin.....	442
Implementing JFreeChart Plugin.....	444
Implementing Config Browser Plugin.....	447
Chapter 12	
 <i>Securing the Struts 2 Application</i>	449
<i>In Depth</i>	451
Understanding Levels of Security	451
Transport Level Security	451
Authentication.....	452
Authorization.....	452
Role-based Access Control.....	452
Container-Managed Security (CMS).....	452
Application-Managed Security (AMS).....	453
<i>Immediate Solutions</i>	454
Implementing Transport Level Security.....	454
Integrating Struts with SSL.....	454
Implementing Role-Based Access Control	455
Using Authentication and Authorization.....	457
User Authentication Schemes.....	458
Authorization.....	461
Using Container-Managed Security	462
Applying Login Configuration.....	464
Basic Login	465
Form-Based Login.....	466
Digest Authentication.....	468
CLIENT-CERT Authentication.....	468

Table of Contents

Application-Managed Security	469
Creating a Security Service	469
Use of Servlet Filters for Security	479
Chapter 13	
Testing the Struts 2 Application	483
<i>In Depth</i>	485
Understanding Unit Testing.....	485
Unit Testing Frameworks.....	486
Immediate Solutions	497
Testing a Struts 2 Action Class.....	497
Creating an Struts 2 Application.....	497
Creating Test Case.....	500
Appendix A	503
Appendix B	521
Appendix C	549
Appendix D	575
Appendix E	613
Appendix F	623
Index	653



Introduction

Thanks for buying the *Struts 2 Black Book*. Designed with comprehensive content on the various concepts of Web application development using Struts 2 Framework, this book includes different new concepts introduced in Struts 2, like Interceptors and Results. You are already aware that Struts 1 has been the popular framework that has created a revolution in the industry. Now Struts 2 is the new Web application Framework that has major architectural changes in terms of APIs, components creation, and configuration when compared with Struts 1. Even then Struts 2 is closer to Struts 1 in terms of the different implementation issues. Struts 2 gets its power features not only from Struts 1, but also WebWork, and XWork Frameworks as it is a combination of all these. Struts 2 implements the MVC-2 architecture, which is quite popular among Web application programmers.

In such a scenario, getting into Struts 2 will be easier for those who have expert hands on Struts 1. However, we have taken care that this book is equally beneficial and comprehensive for those who do not have any exposure to Struts 1. After, going through this book, your preferred framework for designing and developing a Web application will be Struts 2 as you would come to know about the amazing things that can be implemented in your application using the tools, APIs, and the flexibility introduced by this framework. We hope that what Struts 2 has to offer will prove as irresistible to you as it has been for many other developers.

About this Book

In Struts 2 Black Book, you'll find as much Struts 2 as can fit between the covers. There are hundreds of topics covered in different chapters of this book, and each of them is discussed for their implementation through some running Web application in the same chapter.

This book is divided into separate, easily accessible topics with each addressing different programming issues in Struts 2 Framework. In this book, you can find full discussion over following topics:

- ❑ Web application architecture
- ❑ Model-View-Controller (MVC) Model
- ❑ Evolutions of Struts 2 Framework
- ❑ Actions in Struts 2
- ❑ Interceptors
- ❑ Results
- ❑ Struts 2 Plugin Support

- ❑ Validators in Struts 2
- ❑ Struts 2 and OGNL Support
- ❑ Genetic and UI Tags
- ❑ Internationalization
- ❑ Implementing Security in Struts 2 base Web application
- ❑ Testing Struts 2 application
- ❑ Velocity and Framework
- ❑ Struts 2 Tiles Plugin
- ❑ Migration of Struts 1 application to Struts 2
- ❑ Configuration of Struts 2 application

That's just a partial list—there's a great deal more. This book has special coverage for an especially hot topic—AJAX. Together, Struts 2 and AJAX make a combination that is used to create more interactive Web applications. We'll get the full story in this book, with more coverage than some books dedicated to the subject.

How to Use This Book

In this book, we are using Java SE 6 (JDK 1.6) to compile all .java files created in the different applications developed. So, install JDK 1.6 on your system, to build applications developed in this book. All applications discussed in this book are web based applications and the Web server used throughout the book is Apache Tomcat.

Just about everything you need to use in this book, you can get from its companion CD. You can also download the needful from the different locations which have been provided in the related chapters. You will find all the code files created and discussed in different chapters on the CD.

Finding a topic in a particular chapter is also simple. All you have to do is open the first page of the chapter and move through the list of topics displayed, along with their page numbers.

Conventions

There are a few conventions in this book that you should take notice of. For example, all codes explained of this book have been shown through code listings. The code with listing number and caption will appear like this:

Listing 8.18: LoginAction.java for Struts 2

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    private String username;
    private String password;

    public String execute() throws Exception {
        if(username.equals(password))
            return SUCCESS;
```

```

        else{
            this.addActionError(getText("app.invalid"));
            return ERROR;
        }
    }
    public void validate() {
        . . .
        . . .
    }
}

```

You'll also see many important facts throughout this book, which are meant to emphasize on something to be remembered by you:

REMEMBER

8080 is the default port number for the Tomcat Web Server and it can be different for different installation machines. So, change it with the port number of your Web server.

You'll also see notes, which are designed to give you some additional information:

NOTE

This book is about Web application development using Struts Framework, which supports Java-based technologies like Servlet, JSP, etc. Hence, our discussion in this chapter will be centered on the Java-based Web applications.

Each of the figures has a caption to maintain clarity:

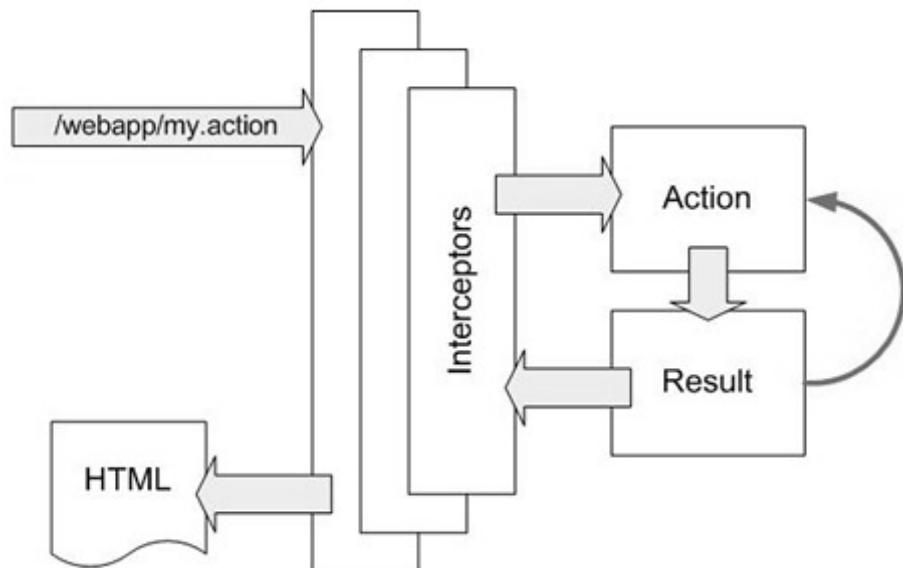


Figure 11.1: Request life-cycle in a Struts 2 application.

Introduction

The tables are placed just below its reference in the chapters like this:

Table 4.2: alias Interceptor	
Interceptor name	Interceptor class
alias	com.opensymphony.xwork2.interceptor.AliasInterceptor
Parameters	
aliasesKey	It is an optional parameter which takes name of the action parameter that will be looked for alias map; the default value for this parameter is aliases

Other Resources

Finally, Struts 2 comes with an immense amount of documentation. This documentation is stored in linked HTML pages. Therefore, you should have a Web browser to work with and view this documentation. Here are some other useful resources where you can find text related to Struts 2, WebWork, and XWork with some helpful tutorials:

- <http://struts.apache.org/>
- <http://struts.apache.org/2.x/>
- <http://www.roseindia.net.struts/struts2/index.shtml>
- <http://www.opensymphony.com/webwork/>
- <http://www.opensymphony.com/xwork/>

There is many help available on Struts 2, but we are hoping you will not have to turn to any of it.

The Black Book Philosophy

Written by experienced professionals, *Black Books* provide immediate solutions to global programming and administrative challenges, helping you to complete specific tasks, especially critical ones that are not well documented in other books. The Black Book's unique two-section chapter format—thorough technical overviews (In Depth) followed by practical examples (Immediate Solutions)—is structured to help you use your knowledge, solve problems, and quickly master complex technical issues to become an expert. By breaking down complex topics into easily manageable components, you can quickly find what you are looking for; diagrams and code help you in appreciating concepts better. Written and edited by the Content and Editorial teams at *Kogent Solutions Inc.* and *Dreamtech Press*, this book is conceptualized to give you everything you need on Struts 2.



1

Struts 2—Advancement in Web Technology

If you need information on:

See page:

Overview of Web Applications	3
Web Application Framework	8
Comparing Struts 1 and Struts 2	20

In the world of Internet, we require various web-based services for all types of business needs. The software industry is sharply looking at new emerging technologies, design patterns and frameworks, which can bring about an evolution in the development of enterprise-level Web applications. We now have numerous technologies (like Servlet, JSP), different design patterns (like MVC) and various new frameworks to develop a Web application. The name of this new application Framework introduced here, in this book, is Struts 2. As you must be aware that a framework is always designed to support the development of an application with its set of APIs, which implement a specific architecture and some reusable components to handle common activities associated with the application. And Struts 2 is one such application framework that is about to revolutionize the Web application development.

NOTE

An in depth discussion over various Struts 2 concepts, here in this book, will be helpful for all types of readers irrespective of whether they have worked with Struts 1 Framework earlier.

Struts 2 is not just a revision of Struts 1, it is more than that. Though the working environment, various component designing, configuration, and few other things seem familiar to the Struts 1 developer, the Struts 2 has some architectural differences when compared with Struts 1. In general, Struts 2 Framework implements MVC 2 architecture with centralizing the control using a Front Controller strategy similar to Struts 1, but the basic code of components and their configuration is quite different here.

In addition to introducing Struts 2 as a new Web application Framework, we'll get into a detailed discussion of this framework which contains every thing that can make it a preferred choice over other frameworks, like Struts 1. Though Struts 2 is not here to replace Struts 1, we now have one more option to choose from while deciding which framework to use for a new Web application development and Struts 2 has all the potential to replace other Web application Frameworks in the industry.

The key features of Struts 2 starting from Interceptors, Results, integration with other popular frameworks and languages through plugins, XWork Validation Framework support, integration with OGNL and implementation of Inversion of Control (IoC) make this framework stand apart from the other frameworks. We have discussed all these features through the chapters of this book.

Struts 2 is a Web application Framework and will be used for the development of a Web application. So, before we start describing Struts 2 Framework and its features, we must have a brief discussion over the topics, like Web application and the technology context which can help you understand architectural decisions taken for the frameworks, like Struts 2. So let's begin our discussion with the term Web application first.

Overview of Web Applications

A Web application is an application that is invoked by a Web browser and executed on the Web server. Whereas, a Web browser is an application that displays the contents of a Web application, a Web server can be defined as a central computer that is accessible to all the clients and delivers requested resources to them. The interaction between a Web browser and a Web application is shown through a flow diagram in Figure 1.1.

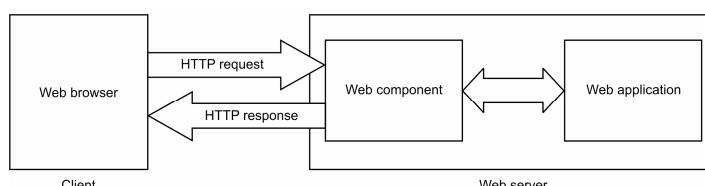


Figure 1.1: Interaction between Web browser and Web server.

The browser sends a request to the Web server by using a communication protocol HTTP. This request is transferred to a Web component, which can communicate with other components, like a database. Once the application gets executed, the Web component sends the HTTP response back to the browser.

There are two types of Web applications. These are as follows:

- ❑ **Presentation-oriented Web applications** – This type of application contains static and dynamic web pages and provides user interaction. It serves only a collection of web pages that display some information and provide no service to the client. In other words, presentation-oriented applications have only presentation logic and do not implement any Business logic. For example, an online tutorial displays what is written already and interacts with the user to get the desired page displayed.
- ❑ **Service-oriented Web applications** – Service-oriented Web applications are generally implemented as an end point of a Web service and provide services to the presentation-oriented applications. Service-oriented applications implement the presentation as well as the Business logic. For example, an online share-trading portal not only displays the status of the share market but also provides services such as buying and selling shares.

In the earlier types of client-server model of computing, there was no standard interfacing mechanism between the client and the server. Any upgrade to the server would require an upgrade to the client, accordingly. Web applications have overcome this drawback of client-server computing. Web applications generate web documents which are supported by almost all the Web browsers. This sort of arrangement allows upgrading of Web application without disturbing thousands of clients.

Generally, the web pages delivered by a Web application to the Web browser are static in nature. To provide a dynamic nature to the web page, special types of programs, called *Scripts* are used. Client-side scripts are executed on the client-side by the Web browser, whereas the server-side scripts are executed by the Web server. Client-side scripts are written in scripting languages, like JavaScript. These scripts utilize the functions provided by the client's computer. In contrast, server-side scripts utilize the functions provided by the server. When the client sends a request to the server, the server responds to the request by sending some type of content to the client. The server sends this content in the form of HTML (HyperText Markup Language). This HTML is understandable to the browser and tells the browser how to render the data to the viewer.

Further, a Web application can be a combination of both active and passive resources. A passive resource, also known as static resource, is the one which does not have any processing of its own. For example, an HTML file is a passive resource because its contents do not change. On the other hand, active resources are those which have their own processing. For example, Servlets and JSPs are active resources. They generate the result dynamically. This dynamic behavior of Servlets and JSPs creates interactivity in Web applications. Today, almost all the Web applications include dynamic content or active resources to increase interactivity between the user and the Web application.

Web applications have to face a number of technical challenges, when they are developed using traditional architectures. We have different Web application Frameworks, like Struts 2, that help to build Web applications with reduced number of errors and efforts. A new framework is always developed over some existing technologies, e.g. Struts 2 has been developed on HTTP and Java Servlet Specification. Though using a framework alleviates a developer from interacting with different technologies on which a framework is based, a working knowledge of these technologies is required. So, before understanding Web application Framework and Struts 2 Framework, here we are providing a brief discussion on the technologies on which Struts 2 Framework is built:

- HyperText Transfer Protocol (HTTP)
- Java Servlet Specification

Now, let's discuss all these technologies in detail.

Hypertext Transfer Protocol (HTTP)

In most of the Web applications, the requested resource is some HTML content and hence most of the Web applications run over HTTP. HTTP is a network protocol used for communicating between the browser running on the user's computer and the Web application running on the Web server. This protocol defines the rules for the ways some data can be transferred and decoded. When the user accesses the Internet, the browser sends an HTTP request to the remote server. This HTTP request is very simple that looks like an ordinary text document. The first line in the HTTP request is the HTTP header that provides the necessary information to the server. It includes the name of the method to be invoked, location of the page, the version of the HTTP protocol used, etc. Having the request serviced, the server sends an HTTP response back to the browser. An HTTP response can contain HTML. The HTTP response has a header in the beginning followed by the HTML text, as shown in Figure 1.2.

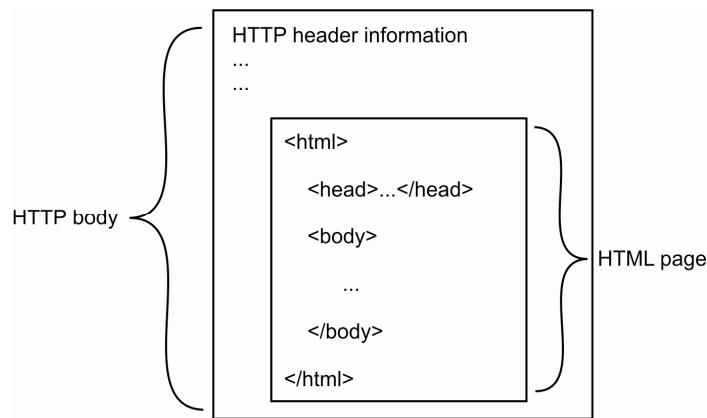


Figure 1.2: HTTP response body.

The HTTP request has no HTML body. It only contains the header information required to perform an action on the server. An HTTP request can handle eight actions and each action is achieved by different methods, like GET, POST, HEAD, PUT, DELETE, TRACE, OPTIONS and CONNECT. Out of these eight methods, only two, namely GET and POST, are frequently used which are defined here as follows:

- ❑ GET – It requests the server to provide the information specified by the Request-URI. The Request-URI refers to a resource on the server. The amount of characters sent to the server is appended to the URL and is limited to URL space available.
- ❑ POST – It submits the server the data to be processed by the resource identified by the URI-Request. It is different from GET because the total amount of characters is not limited and the data sent to the server is not appended to the URL. It is useful for making a request and at the same time sending form data.

A serious drawback associated with HTTP is that it does not preserve the state information between two requests and it is text based. HTTP is a stateless protocol and we cannot track multiple requests from the same client. The large Web application needs a fine approach for user session handling and HTTP only will not work here. HTTP is text-based and Web application needs data binding with requests, which is to be used for further processing. Since, HTTP request and HTTP response can have data for String type only, we need to convert String to a different Java data type and vice-versa, which is time-consuming and error prone.

Java Servlet Specification

Java Servlet specification is introduced to describe the real Java-based Web application and different web components. According to this specification, a Web application can be developed with the components, like Servlets, JSP pages, HTML pages, and some other Java classes. The API developed with specification provides all interfaces and classes for the development of these web components. The Java Servlet specification has been a base for different Web application Framework, like Struts 1 and now for Struts 2.

In the previous section, we discussed two problems with HTTP. To overcome this problem, Java Servlet API has been used over HTTP in the development of Struts 2 Framework. Using Java Servlet API, Java developers can now write the code to be executed on HTTP server to generate HTML content dynamically. We can now create Servlets as a Web component, which can handle HTTP requests and process over different request parameters to generate HTTP response for the client. Servlets are solutions for hurdles faced with HTTP. We can handle client sessions with Servlets and text can easily be converted to other Java data types using type casting mechanism in Java. Let's discuss the different Java technologies, like Servlet and JSP, in brief.

Java Servlets

A Servlet can be viewed as a way to format HTML inside Java. A Servlet is a Java class. It is contained into a Container, which is attached to the Web server. The Container provided with the server has the ability to access the Servlet. Container loads the Servlet during start-up and calls the `init()` method. This method initiates the Servlet for the first time. Once initialized, the Servlet can respond to any number of requests. With each request, a new thread is created as shown in Figure 1.3. To destroy the Servlet, the Container calls the `destroy()` method. In this way, the Servlet is taken out of the service.

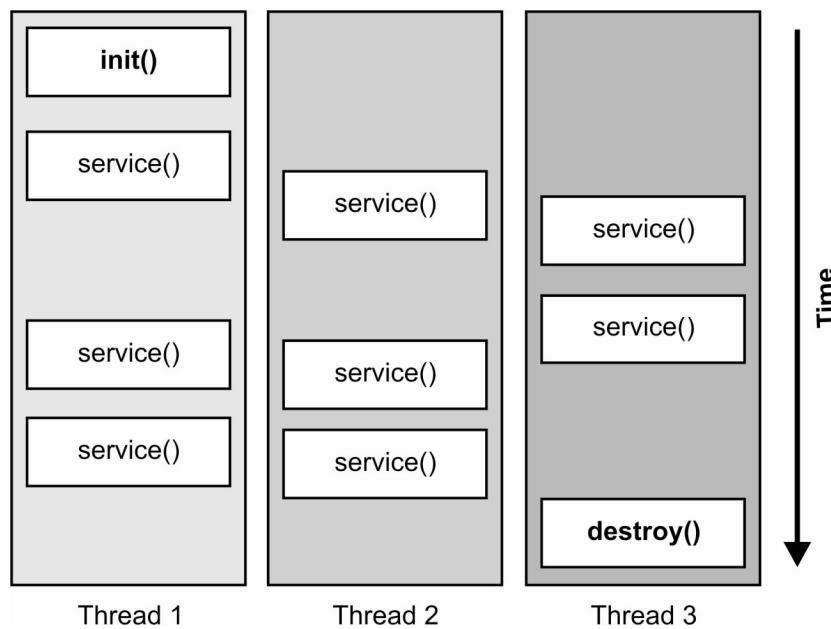


Figure 1.3: A Servlet life-cycle.

To respond to each request, the Container calls the `service()` method of the Servlet. This method is inherited from the parent class used (`GenericServlet` or `HttpServlet`) and need not be implemented in the Servlet class created. This `service()` method further calls the methods, like `doGet()`, `doPost()`, and `doTrace()`, etc. and these methods, too, are provided by the parent class. But, unlike `service()` method, these methods must be implemented in the Servlet, otherwise an exception will be thrown.

Figure 1.3 shows the life-cycle of a Servlet. Here three threads are created, which call the `service()` method concurrently. The first, Thread 1, is created by the `init()` method. Thread 1 further creates Thread 2 and Thread 3. Thread 3 calls the `destroy()` method, which terminates the life-cycle of the Servlet.

When the client sends a request for the Servlet, it is not handled by the server directly. The server hands over the request to the Servlet Container in which the intended Servlet is deployed. The Container provides the HTTP request from the server to the Servlet and the HTTP response from the Servlet to the server.

NOTE

A Web server uses a separate module for performing tasks related to the Servlets, i.e. loading and running of a Servlet. This separate module, which is purely dedicated to the tasks of Servlet management, is called Servlet container or Servlet engine.

Despite the great features of Servlet explained here, there are two big issues concerned to the use of Servlets:

- ❑ **An HTML page designer must be a Java developer**—Servlets use HTML and Java together to design an application. Therefore, a Servlet programmer must be an HTML page designer as well as a Java developer.
- ❑ **Formatting HTML into Java Strings is tiresome and error-prone**—Writing HTML into Servlet requires the HTML content to be written as the argument to `println()` method. This requires a lot of care because HTML content should be written in quotes, which are quite exhaustive.

Java has made it easy to develop and deploy Web applications. Servlets allow the fastest way to use JavaServer technology in your Web application. The power and flexibility provided by Java allows a simple application designed by using Servlets to be gradually converted into a multi-tier enterprise application.

JavaServer Pages (JSP)

JavaServer Pages (JSP) provides solutions to the two issues concerned to the use of Servlet. A developer, while using Servlet, has to work keeping in mind the look of the page as viewed by the viewer. This means that a Servlet designer should be an HTML page designer also. JSP emerged as a solution of this problem by dividing the development procedure in two segments. A Java programmer is responsible for designing the Business logic, while an HTML designer is responsible for the presentation. We can now write Java code and HTML code in a single JSP page separated by delimiters which makes life of a Java programmer and a HTML page designer easy. JSP was designed to provide a static HTML page with more power and flexibility. Let's see how.

When a user clicks a link on a website, which is made by using JSP, the browser makes a request to the server. The Web server recognizes it as a JSP file and passes it to the JSP Servlet Engine. If the JSP file is called for the first time, it is loaded into the RAM. Next, a Servlet is generated from the JSP file by converting HTML into `println` statements. The Servlet is then compiled into a class. To initialize the Servlet, `init()` and `service()` methods are called. The HTML page formed by the Servlet is sent back

to the browser. This HTML page is displayed to the viewer by the Web browser. Figure 1.4 shows a Web server implementing JSPs.

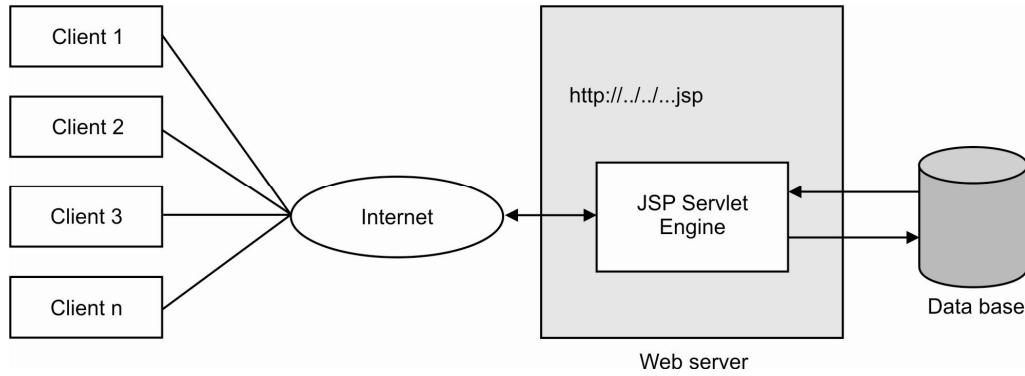


Figure 1.4: Web server supporting JSP files.

The Web server recognizes the request for JavaServer Pages by the .jsp file extension in the URL request and passes the request to the JSP engine. The JSP engine translates the JSP page into a Java class, which is further compiled into a Servlet. This Servlet remains in the memory, until it is not destroyed.

After this discussion over Web application and different technological contexts, we can now move to Web application Framework which is followed by the introduction of Struts 2, a powerful Web application Framework.

Web Application Framework

All the Web applications have some common features. They execute on a common machine, accept input from common input devices, output data to common output devices, and stores information in a common memory. An application framework is based on these common aspects and provides a foundation for developing the applications to the developers and alleviates from putting additional efforts for it.

A framework refers to a set of libraries or classes, which are used in the creation of an application. This involves bundling sections of code, which perform a different task, into a framework. With this sort of arrangement, it becomes very easy to design an application by following the framework and reusing those code-sections. The framework which is used for the development of a Web application is known as Web application Framework, such as Struts 1 and Struts 2.

Struts 2 is a Web application Framework which involves tools and libraries for the creation of Web applications. Struts 2 Framework follows Model-View-Controller (MVC) architecture and implements a front controller approach like Struts 1 Framework.

NOTE

Model-View-Controller (MVC) is a pattern used for the creation of Web applications. It solves the problems of decoupling the Business logic and data access code from the Presentation logic. The two different MVC models are MVC 1 and MVC 2. In MVC 1, all requests are handled by JSP, while in MVC 2 it is handled by a Controller servlet. Struts 2 follows the MVC 2 architecture pattern.

Developing a Web application by using Web application Framework, like Struts 2, first requires a layout of the application structure, followed by an incremental addition of framework objects to that structure.

This is accomplished by creating the framework tools module. This involves generating framework components by using the wizards and customizing them according to the requirement.

A framework provides some common techniques (Figure 1.5) to design an application, which makes the application easier to design, write, and maintain.

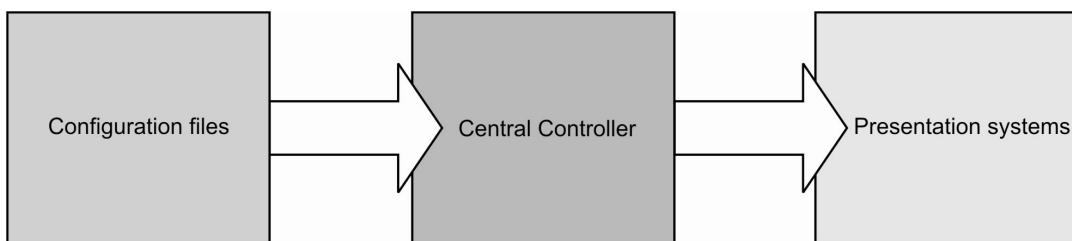


Figure 1.5: A common framework strategy.

The most common technique that frameworks use includes the following strategies:

- **Configuration files** – These files provide the implementation details. These details contain settings specific to an application, such as assembly binding policy, remote objects, etc. Configuration files are read at runtime. These files are external to the application and are not included in the source code.
- **Central Controller** – The Controller provides a way to manage the incoming requests. It receives the requests, processes them, according to a pre-defined logic, and forwards the result to the Presentation systems.
- **Presentation system** – This system allows different people to access different parts of the application at the same time. Java developers can work on the Central Controller, while page designers can work on JSPs.

Having understood the concept of a framework, we can now brief that the Web application Framework eases the process of entire Web application development cycle and allows designing complex Web applications in a way that is easily understandable and manageable. The advantages of using a Web application Framework are described as follows:

- **Simplified design and development** – Web application Frameworks greatly simplify the process of development and designing of Web based projects. It provides a foundation, upon which the developers can stand their structures to implement the application very easily. Since the responsibility of developing various application components is given to different members of the team, depending upon their skills, the development process proceeds smoothly.
- **Reduced production time** – The entire Web application is broken down into small components that are developed and tested individually. This sort of arrangement reduces the production time because testing of smaller components requires less time to debug, as compared to testing the entire project at once.
- **Reduced code and effort** – The Web application components that have been tested and debugged, provide a specific functionality can be set aside to be used further when a similar functionality is required in any application. It offers a reduced amount of code and effort involved in the entire development cycle.
- **Improved performance** – Since each component is developed by a specialized person having specialized knowledge and experience in developing similar applications, the best efforts can be provided to application development. The performance of the code improves further by using a powerful set of APIs, provided by the framework, which offers its own features.

- ❑ **Easy customization and extension**—In order to customize the application or to extend the capabilities of the application, it is required to make changes in a particular portion of the application, rather than developing an entire new application.
- ❑ **Code reuse**—The Web application Framework provides a set of components that are known to work well in other applications. These components can also be used with the next project and by the other members in the organization. New components can also be developed that can be used as reusable code components.

These features of Web application Framework and the way they help in the development of a Web application has made them popular among Web developers. We have a large number of Web application Frameworks available, which are being used in the industry for Web development, like Spring, JSF, Struts 1, WebWork, Struts 2 and many more. Each framework has its own architecture and the way the components are designed and configured are also different. So, instead of having a discussion over the different Web application Frameworks, we can now start our journey with Struts 2.

However, Struts 2 is derived from the WebWork2 Framework which has been in use for the development of Java Web applications. After working for several years, the WebWork2 and Struts communities joined together and created Struts 2. Let's first have a brief introduction of WebWork2 framework.

WebWork2

WebWork2 is an Open Source Framework that is used for creating Java-based Web applications. It is fully based on MVC 2 architecture and is built on a set of Interceptors, Results, and Dispatchers that is integrated on XWork Framework. The View components of WebWork2 include JSP, Velocity, JasperReports, XSLT and FreeMarker. The WebWork2 Framework provides you the ability of creating your own set of templates by using JSP tags and Velocity macros. It also provides features, such as redirect, request dispatcher results, and multipart file uploading.

WebWork2 provides three features—dispatcher, library of tags to be used in view, and Web-specific results. A dispatcher is responsible for handling client requests by using appropriate actions. By using this technique, the most common way to invoke an action is either via a Servlet or a Filter. The Model is composed of POJO (Plain Old Java Objects) classes. WebWork2 also comes with some built-in JSP tags for creating HTML pages.

History of WebWork 2.0

WebWork 2.0, developed by OpenSymphony, was released in February 2004. WebWork 2.0 is the first release of the complete rewrite of WebWork with implementation on top of XWork. WebWork 2.0 Framework is the result of a combination of earlier versions of WebWork Framework and Xwork Framework. Hence, all the versions of WebWork2 (2.1.x and 2.2.x) support the features of WebWork's previous releases and XWork Framework. XWork is a generic command pattern MVC Framework, which is used to power WebWork as well as other applications. It provides a built-in rich framework, which contains Inversion of Control container, a powerful expression language, validation, data type conversion, and pluggable configuration.

Flow of Execution in WebWork2 Architecture

WebWork2 is based on the Model-View-Controller architecture, which means that the logic is separated from view and the Web application can run on the server by the Controller component. The basic flow of execution in WebWork architecture is shown in Figure 1.6.

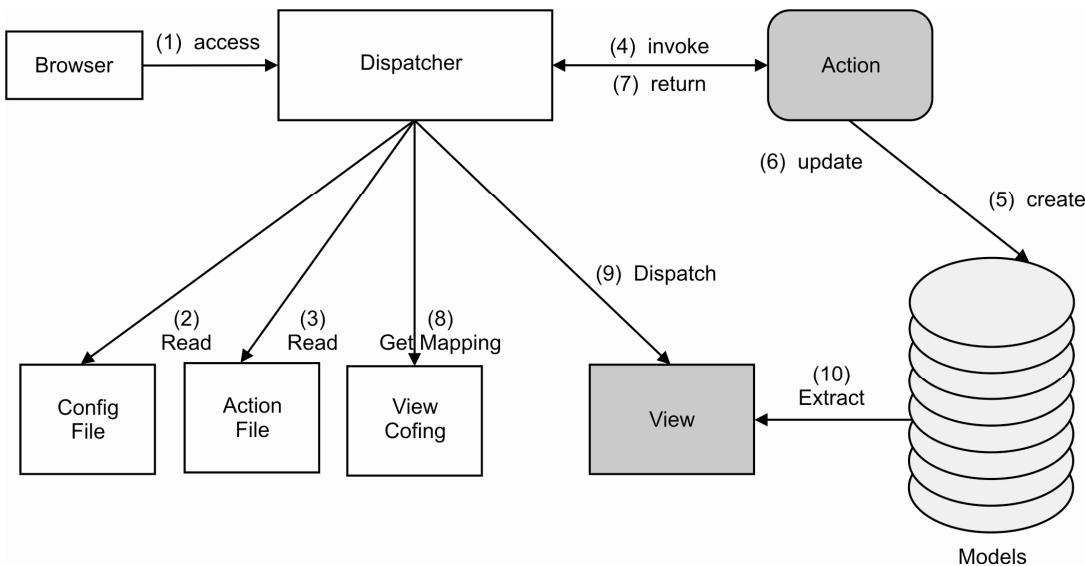


Figure 1.6: Flow in WebWork's architecture.

Figure 1.6 shows the following the sequence of steps:

1. Browser access dispatcher through a request
2. Dispatcher uses configuration file to get the configuration details
3. Dispatcher reads action configurations to know the appropriate action to be invoked
4. Dispatcher creates an instance of Action class and executes its method
5. Action class creates instance of Model. If the Model is already created, step 6 is taken
6. After creating Model instances, the Action class updates Model for the changes, if any
7. Action returns the result code
8. The dispatcher reads result/view configuration to know appropriate view according to the result code returned by the Action class
9. Dispatcher dispatches to the selected view
10. View further extracts data from the Model and displays it to the user in the appropriate manner

The work flow described through these steps can be compared with what you will study for the Struts 2 based application as the basic concept of dispatcher, action and interceptors have been inherited to Struts 2 from WebWork2.

Features of WebWork2

There are several features of WebWork2, and some of the important features are described here:

- ❑ It is easy to learn. You can start programming using MVC 2 design pattern very easily.
- ❑ The interfaces used in WebWork2 are simple.
- ❑ WebWork2 provides configuring the Servlet container through `web.xml`, and configuring WebWork through `xwork.xml`.
- ❑ Interceptors are one of the most important features of WebWork2. Interceptors allow you to do some processing before and/or after the action and results are executed.

- ❑ It is easier to upload a file. WebWork provides an Interceptor, `FileUploadInterceptor`, which provides the retrieval and cleanup of uploaded files. Using this Interceptor, your Action doesn't need to worry about request objects, request wrappers, or even cleaning up the `File` objects.
- ❑ WebWork2 creates dynamic web pages without using any Java code in your JSPs, by using simple expression language called the Object Graph Navigation Language (OGNL).
- ❑ Velocity is used in place of JSP Scriptlets. However, it is not best practice to write Java code in JSP. WebWork provides Velocity that has a simpler format, which can be used in HTML editors very easily.
- ❑ WebWork provides FreeMarker, a template engine which is used to generate text output based on templates. It generates output by using HTML template file and Java objects. It also provides features, like using any JSP tag library.
- ❑ WebWork provides an Expression Language (EL) that is a scripting language which allows simple and concise access to JavaBeans, collections, and method calls. It is also used to eliminate the repetitive Java code.

Struts 2

Struts 2 is a free Open Source Framework for creating Java Web applications. It is based on WebWork2 technology. The features of Struts 2 are user interface tag, type conversion, Interceptor, result and validation. Struts 2 is a highly extensible and flexible framework for creating Java-based Web application. It has been developed on the concepts of MVC 2 architecture. The Struts 2 project provides an open-source framework for creating Web applications that easily separate the Presentation layer and Transaction and Data model on different layers. Struts 2 Framework includes a library of mark-up tags, which is used to make dynamic data. To ensure that the output is correct with the set of input data, the tags interact with the framework's validation and internationalization. The tag library can be used with JSP, Velocity, FreeMarker, and other tag libraries, like JSTL and AJAX technology.

Before exploring the features of Struts 2 and understanding how it is different from its previous versions, let's take a look at the history behind it.

Struts 1 was introduced as an idea of using JSPs and Servlets in Web applications to separate the Business logic from the Presentation logic. As time passed, a need for new enhancements and changes was felt in the original design. This forced developers to evolve a next generation framework that could keep up with these changes. There were two candidates at that time that could meet the requirements of the next generation framework—Shale and Struts Ti. Shale is a component-based framework, which has now become a top-level Apache project, whereas Struts Ti continues to follow the Model-View-Controller architecture. Then in March 2002, the WebWork Framework was released. WebWork includes new ideas, concepts and functionality with the original Struts code. In December 2005, WebWorks and the Struts Ti joined hands to develop Struts 2. The request flow in Struts 2 Web application Framework is shown in Figure 1.7.

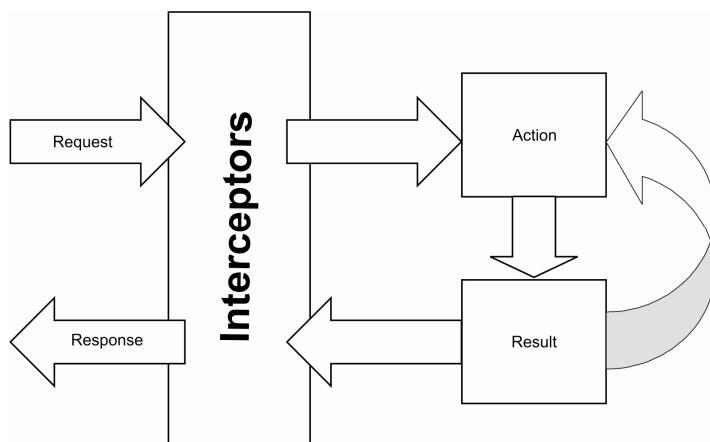


Figure 1.7: Request processing in Struts 2.

The processing of a request can be divided into the following steps:

- ❑ **Request received**—As a request is received by the framework, it matches it to a configuration so that the required interceptors, Action class, and result class can be invoked.
- ❑ **Pre-processing by Interceptors**—Once the configuration is found, the request passes through a series of interceptors. These interceptors provide a pre-processing for the request.
- ❑ **Invoke Action class method**—After the pre-processing by the interceptors, a new instance of the Action class is created and the method providing the logic for handling the request is invoked. It is to be noted that in Struts 2, the method to be invoked by the Action class can be specified in the configuration file.
- ❑ **Invoke Result class**—Once the action is performed and the result is obtained, a Result class matching the return from processing the actions method is determined. A new instance of this return class is created and invoked.
- ❑ **Processing by Interceptors**—The response passes through the interceptors in reverse order to perform any clean-up or additional processing.
- ❑ **Responding user**—The control is returned back to the Servlet engine and the result is rendered to the user.

This was how different components interact internally, but the high level design pattern followed by Struts 2 Framework is MVC 2, similar to Struts 1, and the different components represent the different concerns of MVC 2 design pattern. The Model, View and Controller are represented by different Struts 2 components Action, Result and FilterDispatcher, respectively. The MVC pattern implementation through Struts 2 components is shown in Figure 1.8.

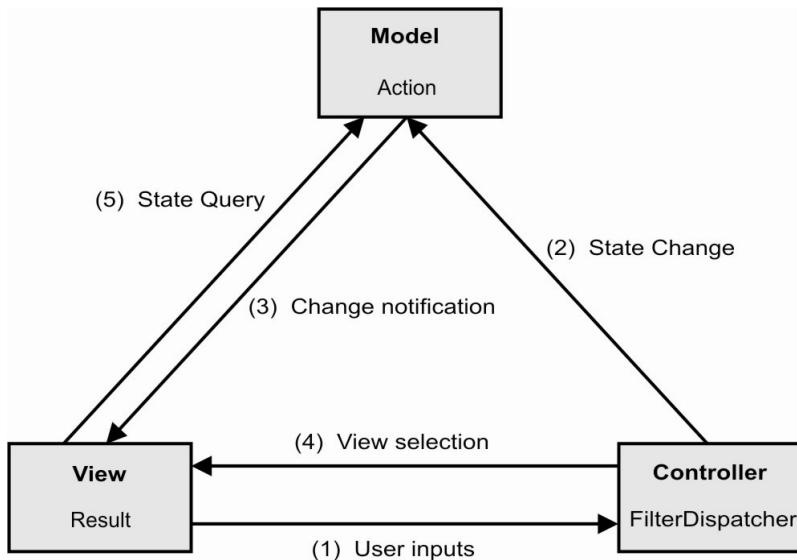


Figure 1.8: Struts 2 components and MVC pattern implementation.

The work flow is broken into the following steps:

1. The user sends request through a user interface provided by the View, which further passes this request to the Controller, i.e. FilterDispatcher which is a Servlet filter class.
2. The Controller Servlet filter receives the input request coming from the user via the interface provided by the View and instantiates an object of suitable Action class, and executes different methods over this object.
3. If the state of model is changed, then all the associated Views are notified for the changes. Otherwise, this step is skipped.
4. Then the Controller selects the new View to be displayed according to the result code returned by action and executes suitable Result to render the View for the client.
5. The View presents a user interface according to the Model. The data is also contained within the Model i.e. actions. View queries about the state of Model to show the current data which is pulled from the action itself.

Similar to Struts 1, Struts 2 implements front controller approach with MVC 2 pattern, which means there is a single controller component here. All requests are mapped to this front controller, i.e. org.apache.struts2.dispatcher.FilterDispatcher class. The main work of this controller is to map user requests to proper actions. Unlike Struts 1, where actions are used to hold the Business logic and the model part represented by form beans, in Struts 2 the Business logic and model both are implemented as action components. In addition to JSP pages, in Struts 2 View can be implemented using other Presentation layer technologies, like Velocity template or some other presentation layer technology. We will study the working of Struts 2 Framework in detail later in this chapter with the involvement of other Struts 2 components, like Interceptors.

Let's now explore the features of Struts 2, which make it so popular.

Features of Struts 2

Struts 2 was created with the intention of streamlining the entire development cycle, from building, to deploying, to maintaining applications over time. These goals are achieved by Struts by providing the following features supporting building, deploying and maintaining applications:

Build Supporting Features

The Build supporting features of Struts 2 are described as follows:

- ❑ New projects can be started easily with the online tutorials and applications provided by various Struts forums.
- ❑ Stylesheet-driven form tags provide decreased coding effort and reduce the requirement for input validation.
- ❑ Smart checkboxes are provided with the capability to determine if the toggling took place.
- ❑ Cancel button can be made to do a different action. Generally, Cancel button is used to stop the current action.
- ❑ It supports AJAX tags that provide an appearance similar to standard Struts tag and help to design interactive Web applications.
- ❑ It provides integration with the Spring application framework, which is also an open source application framework for the Java platform
- ❑ Results obtained after the processing can be processed further for JasperReports, JFreeChart, Action chaining, file downloading, etc.
- ❑ It uses JavaBeans for form inputs and putting binary and String properties directly on an Action class.
- ❑ There is no need of interfaces because the interfacing among the components is handled by the Controller and any class can be used as an Action class.

Deployment Supporting Features

The deployment supporting features of Struts 2 are as follows:

- ❑ It allows framework extension, automatic configuration, and plugins to enhance the capabilities and add new features in future.
- ❑ Error is reported precisely by indicating the location and line of an error, which helps in debugging.

Maintenance Supporting Features

- ❑ Struts Actions can be tested directly, without using the conventional mock HTTP objects to debug the application.
- ❑ Most of the configuration elements have an intelligent default value that can be set only once.
- ❑ Controller can be easily customized to handle the request per action. A new class is invoked for handling a new request.
- ❑ It provides built-in debugging tools to report problems; hence not much time is wasted in the manual testing.
- ❑ It supports JSP, FreeMarker, and Velocity tags that encapsulates the Business logic and requires no need to learn taglib APIs.

Relation between WebWork2 and Struts 2

Struts 2 Framework is based on the WebWork2 Framework. One may think that Struts 2 is an extension of WebWork2, but it is not so. Struts 2 Framework includes features of both WebWork2 and Struts 1 Framework.

Since the arrival of the Apache Struts in year 2000, it has enjoyed a great run among the developer's community. Struts 1 provided a solid framework to organize a mess of JSP and Servlets to develop Web applications, which mostly used server generated HTML with JavaScript for client-side validation. These Web applications were easier to develop and maintain. As time passed, even though the customer's demand for Web applications increased, Struts 1 remained the same. Struts 1 was unable to cope with the growing demands of the customers.

In year 2005, the developer's sat down to discuss the future of Struts project. They came up with Struts Ti proposal. The Struts Ti was also a Struts Framework with some advanced features. But the problem was that Struts 1 was unable to cope with the advancement in technology. So the developers decided to join hands with WebWork developer's team in order to provide a new and efficient framework to the users. The WebWork Framework was part of OpenSymphony and at that time the latest version of this framework was WebWork2. This new framework was meant to provide support for almost all the new technologies which were present at the time for developing a Web application. The Struts and WebWork was then merged together to provide a new framework called Struts 2.

The Struts 2 Framework is dependent on the WebWork2 Framework. The code of WebWork2 was included in Struts 2. Though Struts 2 has incorporated Struts 1 features, drastic changes have been made in Struts 2 in order to make this framework simpler and easy to use for the developers. It has included features, like Interceptors, Results, etc. from the WebWork2. Several files and packages were included from WebWork2 in Struts 2 Framework. Some elements were included as it is, while others were included with some changes, e.g. names of some packages, files, etc. are changed, but their structure remains same. Some of the features from WebWork were removed. New features were added in Struts 2 Framework, but the core part of this new Struts 2 Framework was based on the WebWork2 Framework.

The architecture of Struts 2 is fully taken from the architecture of WebWork2 Framework. The processing of the request in Struts 2 is almost the same as that in WebWork2. There are a number of extra plug-ins that are included in the Struts 2 as compared to WebWork2 Framework.

Thus, it can be said that Struts 2 Framework is very much dependent on the WebWork Framework.

Changes from WebWork2

Both Struts 2 and WebWork2 are frameworks, which are used to create Web applications based on the MVC architecture. There are a number of changes in Struts 2 Framework as compared to WebWork2 Framework. Some of the features are added or changed, some even removed from the Struts 2 Framework. Also some of the members are renamed in the newly created Struts 2 Framework. But, the basic implementation at most of places is same. To make the picture clear, let's take a look at the changes between WebWork2 and Struts 2 Framework.

There are several features which are changed from WebWork2 while developing Struts 2. Some of them are as follows:

- ConfigurationManager is no longer a static factory; an instance is created through Dispatcher. DispatcherListener is used for custom configuration.
- The tooltip library used by xhtml theme was replaced by Dojo's tooltip component.
- Tiles integration is available as a plug-in.
- Wildcards can be specified in action mappings.

- ❑ To allow field errors/action errors to be stored and retrieved through session, MessageStoreInterceptor is introduced. Messages are also stored and retrieved through session.

There are several features which are removed from WebWork2 to develop Struts 2. Some of them are as follows:

- ❑ **AroundInterceptor** – The AroundInterceptor is removed in WebWork2. If you are extending the AroundInterceptor in your application, import the class into your source code from WebWork2 and modify it to serve as your base class, or rewrite your Interceptor.
- ❑ **oldSyntax** – The oldSyntax is removed from WebWork2.
- ❑ **Rich text editor tag** – Rich text editor tag is removed and is replaced by the Dojo's rich text editor.
- ❑ **Default method** – doDefault is not supported for calling the default method.
- ❑ **IoC Framework** – The internal IoC framework has been deprecated in WebWork 2.2 and removed in Struts 2. Struts 2 Framework has a Spring plugin for implementing the IoC. This is implemented by using the factory, ObjectFactory.

Table 1.1 shows the members that are renamed in Struts 2.

Table 1.1: Renamed members in Struts 2	
WebWork2	Struts 2
com.opensymphony.xwork.*	com.opensymphony.xwork2.*
com.opensymphony.webwork.*	org.apache.struts2.*
xwork.xml	struts.xml
webwork.properties	struts.properties
DispatcherUtil	Dispatcher
com.opensymphony.webwork.c onfig.Configuration	org.apache.struts2.config.Settings

In the following example, the tag prefix conventions are changed in Struts 2:

JSP	s:	<s:form ...>
Freemarker	s.	<@s.form ...>
Velocity	s	#sform (...)

Architecture of Struts 2

The architecture of Struts 2 Framework is fully based on MVC architecture. The Struts 2 architecture has a flexible control layer based on standard technologies, like JavaBeans, ResourceBundles, XML, Locales, Results, Interceptors, etc. The architecture of the Struts 2 is shown in Figure 1.9.

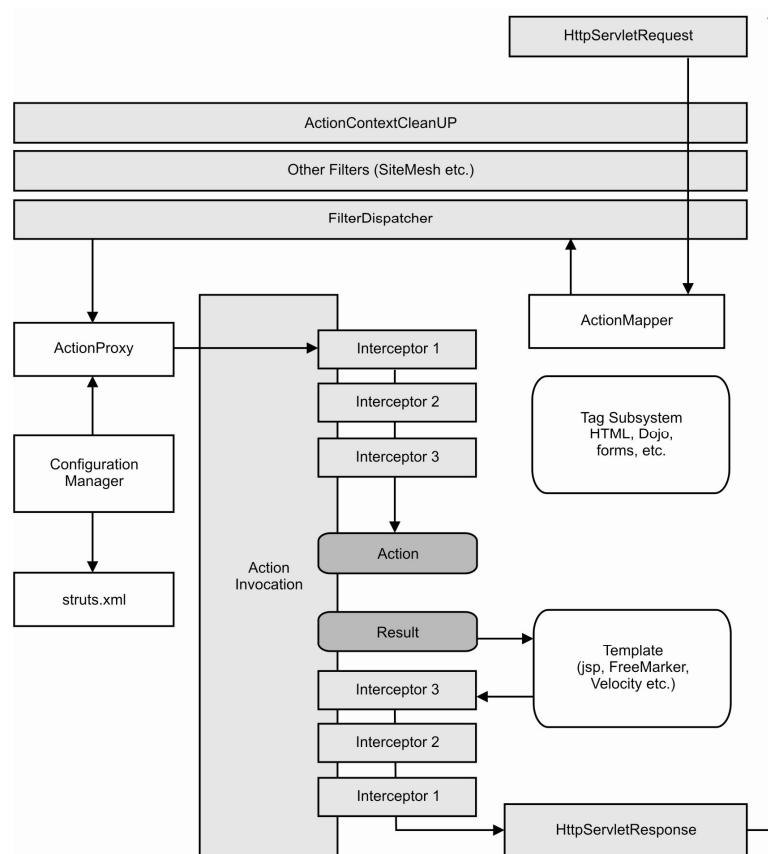


Figure 1.9: Architecture of Struts 2 Framework.

The various components of Struts 2 can be categorized into the following groups, according to the Struts 2 architecture:

- ❑ **Servlet Filter**—The **ActionContextCleanUP**, **FilterDispatcher**, and **SiteMesh** (and other filter) components are treated as Servlet filter component.
- ❑ **Struts 2 Core**—**ActionProxy**, **ActionMapper**, **Tagsystem**, **ActionInvocation**, **ConfigurationManager** and **Result** are treated as Struts 2 Core component.
- ❑ **Interceptors**—Interceptors are presented both above the Action component and below the Result components.
- ❑ **User created components**—The **struts.xml** file, Action classes, and Templates are treated as user created components. Here the user is allowed to change these configurations.

In the Figure 1.9, first the client request passes through the Servlet container, such as Tomcat. Then the client request passes through a chaining system, which involves the three components of Servlet filters.

In the chaining system, first the **HttpServletRequest** goes through the **ActionContextCleanUP** filter. The **ActionContextCleanUP** works with the **FilterDispatcher** and allows integration with **SiteMesh**. Next the **FilterDispatcher** invokes the **ActionMapper** to determine which request should invoke an Action. It also handles execution actions, clean up **ActionContext**, and serves static

content, like JavaScript files, CSS files, HTML files, etc. whenever they are required by various parts of Struts 2.

The ActionMapper maps HTTP requests and action invocation requests and vice-versa. The ActionMapper may return an action or return null value, if no action invocation request matches.

When the ActionMapper wants that an Action should be invoked, the ActionProxy is called by interacting with the FilterDispatcher. The ActionProxy obtains the Action class and calls the appropriate method. This method consults the framework Configuration Manager, which is initialized from the struts.xml file. After reading the struts.xml file, the ActionProxy creates an ActionInvocation and determines how the Action should be handled. ActionProxy encapsulates how an Action should be obtained, and ActionInvocation encapsulates how the Action is executed when a request is invoked. Then the ActionProxy invokes Interceptors with the help of ActionInvocation. The main task of Interceptors is to invoke the Action by using ActionInvocation.

Once the Action is determined and executed, the result is generated by using Result component. The Result component lookup data from the Template with the help of Action result code mapped in struts.xml. The Template may contain JSP, FreeMarker, or Velocity code which is used to generate the response. The Template uses the Struts Tags, which are provided by the Struts 2 Framework. After collecting data/result from Template, the Action Invocation produces the result with the help of Interceptors.

Before, we conclude our discussion over Struts 2 architecture and how the different events take place one after another when a request is processed in Struts 2 based Web application, let's go through a few more important Struts 2 concepts in brief – Interceptors, ValueStack, and OGNL.

Interceptors

The architecture of Struts 2, shown in Figure 1.9, shows a number of interceptors being executed before action and after result is executed. Interceptors are key concepts being utilized in Struts 2 Framework. Interceptors allow you to develop code that can be run before and/or after the execution of an action. Interceptors may also provide for Security checking, Trace Logging, Bottleneck Checking by using the interceptor package. We have a stack of interceptors being executed before action and they can be considered as a kind of request processor in Struts 1. Different sets of interceptors can be used in the stack according to the common functionalities being provided by them. Each interceptor simply defines some common workflow and crosscutting tasks which make them separate from other concerns and makes them reusable.

NOTE

Interceptors have been covered for all its functioning, and configuration details in chapter 4.

ValueStack and OGNL

Some important concepts implemented in Struts 2 are ValueStack and the expression language to interact with this ValueStack, i.e. Object Graph Navigation Language (OGNL). A ValueStack can be defined as a storage structure where the data associated with the current request is stored. As the name suggests, ValueStack is a collection of data stored in a Stack. OGNL expression language is used to retrieve and manipulate data from ValueStack. The data can be set into ValueStack during the pre-processing of the request, manipulated when the Business logic in action is being executed, and read from ValueStack when the Result renders the response for the client.

NOTE

We have covered OGNL with all its syntax and uses in Chapter 5.

All new concepts introduced in Struts 2 Framework has been thoroughly discussed in the featured chapters of this book with various examples and applications.

Comparing Struts 1 with Struts 2

Struts 2 Framework is a new framework, but it can easily be grasped for its new concepts and implementations by comparing it with the most popular Web application Framework, i.e. Struts 1. This comparison is important as both, Struts 1 and Struts 2, share a common implementation in terms of MVC 2 pattern and action based approach in the development of components. There are also some advanced features which are added in the new Struts 2 version, such as Interceptors, XWork Validation Framework support, OGNL support, integration with other frameworks using various plugins, POJO Action, POJO Forms, etc. There are lots of changes which we can observe in Struts 2 Framework. But the basic implementation at most of the places is same as in Struts 1. To make the picture clear, let's take a look at the similarities and differences between Struts 1 and Struts 2 Framework.

Similarity between Struts 1 and Struts 2

Both Web applications framework, i.e. Struts 1 and Struts 2, provide the following key components:

- Both versions follow the MVC 2 architecture.
- Both versions use a *request* handler that maps Java classes to Web application URIs.
- A *response* handler that maps logical names to server pages, or to other Web resources in both Struts 1 and 2.
- A tag library helps us to create rich, responsive, form-based applications, and generate dynamic content of web pages in both versions of Struts.

In Struts 2, all the preceding concepts are redesigned and enhanced, but the same architectural hallmarks remained.

We have discussed the similarities between Struts 1 and Struts 2. But the Struts 2 Framework introduces a number of new concepts and type of implementations. Let's now discuss the key changes in Struts 2, as compared to the Struts 1.

Difference between Struts 1 and Struts 2

In addition to various similarities between Struts 1 and Struts 2, there exists various differences as well that gives Struts 2 an edge over the its counterpart. The description of these differences, shown in Table 1.2, is helpful while migrating from Struts 1 to Struts 2. These points help in getting aware of the differences between Struts 1 and Struts 2 components. Further, the comparison makes the benefits of Struts 2 over Struts 1 clear to the reader.

Table 1.2: Difference between Struts 1 and Struts 2

Feature	Struts 1	Struts 2
Action classes	For developing Struts controller component of MVC model, the Action classes are needed, which extends the Abstract base class. This is a common problem in Struts 1 to program abstract classes rather than interfaces	In Struts 2, an Action may implement <code>com.opensymphony.xwork2.Action</code> interface along with other interfaces to enable optional and custom services. Struts 2 provide a base

Table 1.2: Difference between Struts 1 and Struts 2		
Feature	Struts 1	Struts 2
		com.opensymphony.xwork2.ActionSupport class to implement the commonly used interfaces. Thus, Action interface is not required. Any POJO object with a execute signature can be used as a Struts 2 Action object
Binding values into views	To access different objects, Struts 1 uses the standard JSP mechanism for binding objects into page context	The technology <i>ValueStack</i> is used in Struts 2, so that taglibs can access values without coupling your JSP view to the object type it is rendering. Reuse of views to a range of types that may have same property name but different property types are allowed by using ValueStack strategy
Servlet Dependency	When an Action is invoked the HttpServletRequest and HttpServletResponse are passed to the execute() method. That's why Struts 1 Actions have Servlet dependencies	Struts 2 Actions are not coupled to container. Servlet contexts are represented as simple Maps that allow Actions to be tested in isolation. If required, the original request and response can still be accessed in Struts 2
Thread Modelling	There is only one instance of a class for handling the entire requests specific to a particular action. Thus Struts 1 Actions are singleton and must be thread-safe. The singleton strategy places restrictions on what can be done by using Struts 1 Actions and thus requires extra care to develop	There are no thread-safety issues because Action objects are instantiated for each request in Struts 2
Testability	The execute method exposes the Servlet API; this is the major hurdle for testing Struts 1 Action	The testing is done by instantiating the Action, setting properties, and invoking methods. Also the dependency injection support also makes the testing simpler
Control of Action Execution	Struts 1 supports separate Request Processors (life-cycles) for each module, but the same life-cycle is shared to all the Actions in the module	Struts 2 supports creation of different life-cycles on a per Action basis via Interceptor Stacks. Whenever needed custom stacks can be created and used with different Actions

Table 1.2: Difference between Struts 1 and Struts 2

Feature	Struts 1	Struts 2
Harvesting Input	To capture input, Struts 1 uses an object of <code>ActionForm</code> . All <code>ActionForms</code> must extend a base class. Since other JavaBeans cannot be used as <code>ActionForms</code> , developers often create redundant classes to capture input. <code>DynaBeans</code> can be used for creating conventional <code>ActionForm</code> classes, but here too developers may be redescribing the existing JavaBeans	Struts 2 eliminates the need of second input object by using action properties as input properties. This Action property is accessed from the web page via the taglibs. Struts 2 also support POJO form objects and POJO Action. Input properties may be rich object types with their own properties. Rich object types, which include business or domain objects, can be used as input/output objects
Expression Language (EL)	Struts 1 supports JSTL, so it uses the JSTL EL. The EL has relatively weak collection and indexed property support	The Struts 2 Framework supports Expression Language (EL) that is known as Object Graph Notation Language (OGNL). The EL is more powerful and very flexible. Struts 2 also support JSTL
Validation	Validation in Struts 1 is supported by a validate method on the <code>ActionForm</code> , or by using a Validation Framework thorough a plugin	Validation in Struts 2 is performed by the validate method and the XWork Validation framework. The XWork Validation framework supports the chaining validations into the sub-properties by using the validations defined for the properties class type and the validation context
Type Conversion	The type-conversion is performed by Commons-Beanutils. The properties of Struts 1 <code>ActionForms</code> are usually of String type	The type-conversion is performed by Object Graph Notation Language (OGL)

This comparison between Struts 1 and Struts 2 is helpful for the developers who have interacted with Struts 1 Framework for Web application development. This explains where Struts 2 is different from Struts 1 and what makes it preferred over Struts 1.

This chapter embarks with Struts 2 Framework and Web application Frameworks, followed by the introduction of Struts 2 Framework—a powerful and flexible Web application Framework—and moves towards a discussion on WebWork2 and Struts 2 that brings various architectural views, which further creates a new terminology to be used throughout this book, such as Interceptors, Results, ValueStack, OGNL, and more. The chapter concludes with the comparison of Struts 1 and Struts 2.

The next chapter covers the essentials for the development of Struts 2 based Web applications—downloading required APIs, standard directory structure of Struts 2 applications, creation of different components, and their configuration details, etc.



2

Creating Application in Struts 2

If you need information on:

See page:

Setting Struts 2 Environment	25
Developing First Application in Struts 2	28
Deploying Struts 2 Application	35
Running Struts 2 Application	37

After discussing Struts 2 Web application Framework, its evolution and its architecture in the previous chapter, we can now move on to building our first Web application based on this elegant, flexible, and extensible framework. The way different components are designed and configured in Struts 2 Framework makes Web application development faster, simpler and in addition more productive in comparison to Struts 1. As we know that Struts 2 is an action based MVC 2 Web application Framework, the basic structure of the application developed using it is similar to what has been used in Struts 1. But the flexibilities introduced and support for the concepts of Interceptor, Inversion of Control, ValueStack, and OGNL have removed various overheads. For example, we do not need an ActionForm type of class to handle data. The action class itself can behave as ActionForm and data can be obtained from the ValueStack using OGNL as expression language, which is further supported by new Struts 2 tags introduced.

Any framework comes with its own set of APIs which can help you to create different components to be used within the framework. We need to understand these APIs which has been developed for the implementation of all framework features. Different interfaces, classes, and tag libraries should be used after understanding the core of their design and uses. You might find the interfaces, classes, different components (Interceptors, Results) and other configuration techniques to be used in Struts 2 quite new and difficult when used for the first time. But as the work flow is quite simple in Struts 2 through the different components development and their execution in the application, a simple Struts 2 application first can be very helpful here. This Struts 2 application creation here will support our further discussion on the new techniques and traditions introduced in the framework.

Let's first start the development of our first Struts 2 application by setting the development environment for it.

Setting Struts 2 Environment

Before getting started with your Struts development, you need to prepare your environment first. This includes getting Struts 2 APIs from some source and making required JARs available for the component development. The standard directory structure of Struts 2 Framework should also be explained earlier to any component development. Further, a framework is developed on some existing technology as Struts 2 is based on Java Servlet specification. Hence, we must have support of some Web container, which can manage a Web application with components of framework compatible version. So, let's first look at the platform requirement for Struts 2 Framework.

Platform Requirements

To construct a Web application using Struts 2, you need the following version of JDK, JSP, and Servlet supported on your system:

- Servlet API 2.4
- JSP API 2.0
- Java 5 (JDK 1.5) (you can use Java 6 also)
- A J2EE compatible Web server

Make sure that the Web server used has a J2EE compatible container, which supports Servlet 2.4 and JSP 2.0 specifications.

NOTE

Though, you can use any Web server with J2EE compatible container, we have used Tomcat 5.5.23 as Web server to deploy our Web applications.

Getting Struts 2 APIs

To develop a Struts 2 based Web application, you need to install Struts 2 APIs. You can download the latest version of Struts 2 from the Apache website (<http://struts.apache.org/2.x/>). At the time of writing this book, the latest available release of Struts 2 is Struts 2.0.6. Download the binary distribution of Struts 2.0.6 on your computer system.

Once you have the Struts 2.0.6 distribution installed, you can develop Web application using Struts 2 APIs, which are available in the form of JARs. Follow these steps for getting prepared before you start coding for Struts 2-based Web application:

- ❑ Uncompress the Struts 2 archive (struts-2.0.6-all.zip) and save it on your local disk, say D:\Struts2. You will get a directory struts-2.0.6 created in D:\Struts2. Other important subdirectories which can be found here are as follows:
 - D:\Struts2\struts-2.0.6\apps – It contains .war files for sample Struts 2 applications.
 - D:\Struts2\struts-2.0.6\lib – This folder contains all Struts 2 JAR files which are required for Struts 2 based Web applications. These JAR files contain Strut 2 API in the form of interfaces, classes, and some default configuration files.

Directory Structure

All the Web applications are packaged into a standard directory structure. This directory structure is the container that holds the components of a Web application. You can start with the following steps to start with a standard Web application directory structure:

- ❑ Create a new directory D:\Code\Chapter 2 to contain Struts 2 application created in this chapter.
- ❑ Now create a new directory structure, as shown in Figure 2.1, in the D:\Code\Chapter 2 folder for your first Struts 2 application (Struts2Application) to be created here.

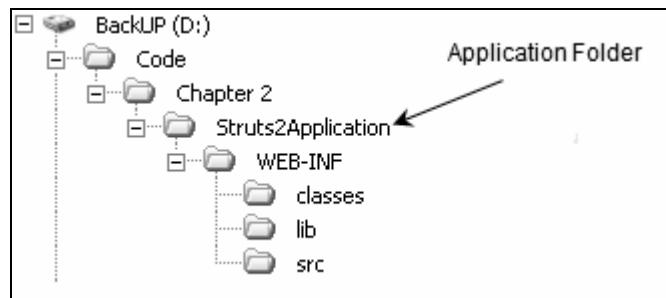


Figure 2.1: Directory structure of First Struts 2 Web application.

Table 2.1 describes a directory structure of a simple Web application, named Struts2Application, and a list of files that should exist in this directory.

Table 2.1: Directory structure of Struts2Application

Directory	Contains
/Struts2Application	This is the root directory of the Web application. All the JSP and HTML files are stored in this directory. You can also put JSP and HTML pages in separate folders here

Table 2.1: Directory structure of Struts2Application

Directory	Contains
/Struts2Application/WEB-INF	This sub-directory contains all resources which are used in the application. web.xml must exist in this directory which contains the configuration/ deployment of the Web application
/Struts2Application/WEB-INF/classes	In this sub-directory Java class file, Struts 2 action class files and other utility classes are located. For developing this application, the struts.xml and ClientAction class file must exist in this directory
/Struts2Application/WEB-INF/lib	This sub-directory contains Java Archive (JAR) files on which the components of this Web application is dependent
/Struts2Application/WEB-INF/src	This sub-directory contains the source code, which is used to develop a Web application

- ❑ Copy some required Struts 2 JARs from D:\Struts2\struts-2.0.6\lib for this application into your D:\Code\Struts2Application\WEB-INF\lib directory. The JARS added for this application are as follows:
 - commons-logging-1.1.jar
 - freemarker-2.3.8.jar
 - ognl-2.6.11.jar
 - struts2-core-2.0.6.jar
 - xwork-2.0.1.jar
- ❑ Create a web.xml file into your D:\Code\Struts2Application\WEB-INF\ directory. The following code snippet shows the basic structure of web.xml file without any Servlet or Filter mapping:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

</web-app>

```

- ❑ Create a struts.xml file into your D:\Code\Struts2Application\WEB-INF\classes\ directory. The following code snippet shows the basic structure of struts.xml file without any action mapped in it:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
</struts>
```

Now, we are ready with the basic directory structure and require APIs to develop various components for our first Struts 2 based Web application.

Developing First Application in Struts 2

Now that you have a fundamental knowledge of Struts 2 Web application directory structure, it is time to create different Struts 2 Web application components. A Struts 2 application consists of JSPs, action classes, action mapping, Struts 2 configuration files, and web configuration file. Let's have a basic understanding of these parts and how they work together in our application.

Our sample application consists of a simple JSP page providing a form with some input fields where the user enters his/her name, age, and salary. If the user enters the input field correctly and submits the page by clicking the 'Submit' button, then another page showing the correct entries would be displayed. Otherwise, a JSP error page is displayed.

You already know by now that the Struts 2 application is based on MVC design pattern. The process of creating a Web application using Struts 2 begins with the identification of the Model, the View, and the Controller. This application can be described in the following steps:

1. Create all the Views, which will provide user interface for your application.
2. Create an action class and define the different input properties and its getter/setter methods within that action class. In this way, we need not create a separate FormBean class as required in Struts 1. You can also create a separate class for your Model, like ActionForms in Struts 1. It is discussed in the coming chapters.
3. Establish the relationship between the Views and the Controllers (action mapping) in your configuration file, i.e. `struts.xml` file.
4. Modify your `web.xml` to make your Web application Struts 2 enabled.
5. Deploy and run the application.

Different components created here must be placed in the application folder within the subfolder specified for it. We need to create various JSP pages, action class, and configuration files. Figure 2.2 shows the directory structure of `Struts2Application` after the addition all these components in the application.

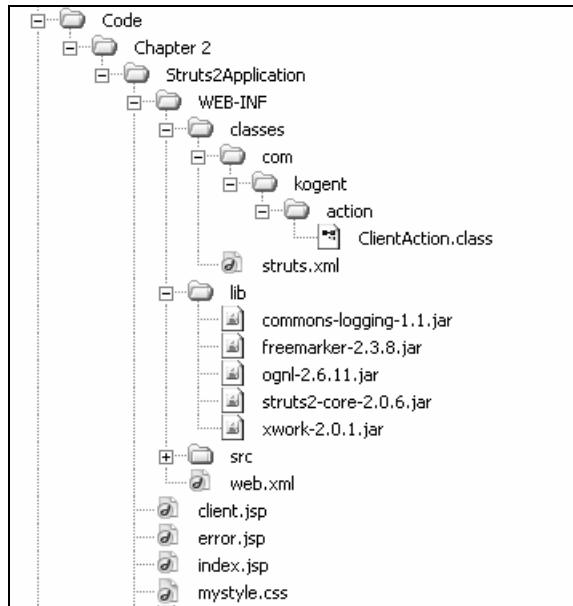


Figure 2.2: Directory structure of a Struts2Application with different components.

Figure 2.2 shows different files placed in different folders of the application. Here's Table 2.2 that shows the various code files developed in Struts 2 based Web application along with their brief description.

Table 2.2: Code files created in Struts2Application Web application

File	Description
index.jsp	This file is used to generate input screen which provides the user a form with three input fields for name, age and salary
client.jsp	This file is used to display an output when the user has submitted his/her name, age, and salary correctly
error.jsp	This file is used to display an error message when the user leaves any field empty in the index.jsp page or some other error occurs
ClientAction.java	This action class is used to process the data entered from index.jsp page and it returns an appropriate result code
web.xml	This is the Deployment Descriptor of the Web application
struts.xml	This is the Struts 2 configuration file containing action mappings
mystyle.css	This is a simple style sheet created and used in all JSP pages created (optional)

Let's develop these components for our Web application step by step. The development of various JSP files, action classes, and modification in configuration files have been discussed here followed by the deployment and running of our first Struts 2 based Web application.

Creating JSP pages

Before creating the action class for your Web application, you need to describe the Views that represent the Presentation layer of your application. To develop this Web application, you need to create three views, i.e. index.jsp, client.jsp and error.jsp. To create your view components, you need to use JSP/HTML and Struts tag library.

Creating index.jsp

The Client View represents the index.jsp. It is the first page of this application on which the user inputs his/her name, age and salary. According to the input text, the corresponding View is displayed.

Here's the code, given in Listing 2.1, for index.jsp (You can find index.jsp file in Code\Chapter 2\Struts2Application folder in CD):

Listing 2.1: index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>

    <head>
        <title>struts 2 Application</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head> <body>

        <h1>Struts 2 Application</h1>
        Enter Client Details:

        <s:form action="clientAction" method="post">
            <s:textfield label="Name" name="name"/>
            <s:textfield label="Age" name="age"/>
            <s:textfield label="Salary" name="salary"/>
            <s:submit/></s:form>
    </body>
</html>
```

The source code of Client View is like any other HTML/JSP pages containing a form, which is used to gather data from the user. To use the `<%@ taglib prefix="s" uri="/struts-tags" %>` tag, the struts2-core-2.0.6.jar file must be in the WEB-INF\lib\ directory and this jar file must be set in the class path of your Web application. The html tags, used in the body of html, is used to input some text.

The index.jsp uses a Struts 2-specific HTML tag, which is used to generate HTML content of the web pages. The Struts 2 HTML form tag `<s:form />` is used to encapsulate the Struts 2 form processing. Its attributes are method, action, etc. The method attribute is set to the 'post', which means that `post()` method of the HTTP is used to handle the request. The action attribute represents the URL to which the form will be submitted. This attribute is also used to find the appropriate action mapping in the Struts configuration file, struts.xml. The value used in this example is validField, which maps to an action mapping. The `<s:textfield />` tag is used to emulate an HTML text field on which the

user is allowed to input some text. The `<s:submit />` tag is used to emulate an HTML submit button. Upon submission, the `<s:form />` tag is created and populated with the value of the `<s:textfield />` tags.

Creating client.jsp

The Client Salary View is represented by the `client.jsp`. When the user submits name, age, and salary from `index.jsp` page, this view is displayed. In other words, this view is displayed when the user successfully submits the form, `index.jsp`.

Here's the code, given in Listing 2.2, that shows the source code for `client.jsp` (You can find `client.jsp` file in `Code\Chapter 2\Struts2Application` folder in CD):

Listing 2.2: `client.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Welcome - Submitted Successfully.</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        Thank you! <b><s:property value="name"/></b>.
        <br><br>Your Age : <b><s:property value="age"/></b>
        <br><br>Your Salary : <b><s:property value="salary"/> $</b>
    </body>
</html></html>
```

In Listing 2.2, the `<s:property value="name">` tag is used to find the `name` property automatically and print the `name` property of the client scripting variable.

Creating error.jsp

The Error View is represented by the `error.jsp`. When the user leaves any field, like name, age and salary, empty in the `index.jsp` page, the JSP Error page is displayed. This view page displays only when there is some error on `index.jsp` page.

Here's the code, given in Listing 2.3, that shows the source code for `error.jsp` (You can find `error.jsp` file in `Code\Chapter 2\Struts2Application` folder in CD):

Listing 2.3: `error.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Some Error</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>

    <body>
        <h1>Error!</h1>
        This error page is being shown because any of following reasons:
        <ul class="boldred">
```

```
<li>Field(s) left blank.</li>
<li>Invalid Data Entered.(For example: String in place of Integer.)</li>
</ul>
</body>
</html>
```

Creating Action Class

Let's create our first Struts 2 action class, which will be used to embed some Business logic. The data sent from index.jsp page is to be processed by some action class. Let's create an action class by extending the com.opensymphony.xwork2.ActionSupport class. In our case, the model is being implemented into the action class itself.

Here's the code, given in Listing 2.4, for ClientAction class (you can find ClientAction.java file in Code\Chapter 2\Struts2Application\WEB-INF\src\com\kogent\action folder in CD):

Listing 2.4: ClientAction.java

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class ClientAction extends ActionSupport{

    String name;
    int age;
    int salary;
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public int getAge(){
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getSalary(){
        return salary;
    }
    public void setSalary(int salary){
        this.salary = salary;
    }
    public String execute() throws Exception{
        if(getName().equals("") || getName()== null ||
           getAge()== 0 || getSalary()== 0){
            return ERROR;
        }
        else{
            return SUCCESS;
        }
    }
}
```

All the input fields to be submitted from `index.jsp` are declared as the properties of `ClientAction` action class with all getter/setter methods for these properties.

Now save this file in `Struts2Application\WEB-INF\src\com\kogent\action` folder. Next compile this file to get `ClientAction.class` file. Make sure that the `ClientAction.class` file is placed inside `Struts2Application\WEB-INF\classes\com\kogent\action`.

NOTE

While compiling Java source files created using Struts 2 APIs, make sure that the required JARs, like struts2-core-2.0.6.jar and xwork-2.0.1.jar, are added into the users CLASSPATH.

Configuring Actions in `struts.xml`

The Struts 2 configuration file is named as `struts.xml` and will be used in all Struts 2 based Web applications. It is used as a glue to initialize your own MVC resources. These resources are Interceptors, action classes, and Results. Interceptors are used to pre-process and post-process a request. The action classes are used to call Business logic and data access code. Results are used to prepare views, like Java Server Pages and FreeMarker templates.

There are several elements that can be used to configure in `struts.xml`. These elements are packages, namespaces, includes, action, results, Interceptors, and exception.

Here's the code, given in Listing 2.5 that simply adds the configuration shown in your copy of `struts.xml` file (you can find `struts.xml` file in `Code\Chapter 2\Struts2Application\WEB-INF\classes` folder in CD):

Listing 2.5: `struts.xml` file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <action name="clientAction" class="com.kogent.action.ClientAction">
            <result name="success">/client.jsp</result>
            <result name="error">/error.jsp</result>
            <result name="input">/error.jsp</result>
        </action>
    </package>
</struts>
```

NOTE

Make sure `struts.xml` is placed in `WEB-INF\classes` folder of your application.

This configuration file uses different elements, like package, action, and result. Let's describe them in brief before proceeding to running the application.

Package Configuration

Package is used to group actions, result types, Interceptors, and Interceptor-stacks into a logical configuration unit. The attribute of package element may be name, extends, namespace, and

abstract. The name attribute of package element acts as the key, which is used for later reference in the package. The extends attribute is an optional attribute. It allows one package to inherit the configuration of one or more previous packages. The namespace attributes helps in separating different packages into different namespace, and hence, help in avoiding action naming confliction. The abstract attribute is optional, which is used to create a base package that can omit the action configuration.

Action Configuration

The action configurations are the basic unit-of-work in the Struts Framework. The action maps an identifier to handle an action class. The action's name and the framework use the mapping to determine how to process the request, when a request is matched. The action mapping requires name attribute, and also a set of result types, a set of exception handlers, and an Interceptor stack.

The element of action attribute is name and class, and sub-element is result. The name attribute is the name of the action, which is used to match a part from the location requested by a browser. For example, if a request is `http://localhost:8080/Struts2Application/clientAction.action` then it will map to the action mapping provided with name="clientAction". The class attribute of action element is the class name of your action class to be executed here with full package description. It is used to execute Business logic and data access code of your action class.

Result Configuration

The result element is used to display your desired view. When the `execute ()` method of action class completes, it returns a String value. This String value is used to select a particular result element. In our Struts2Application example, if the `execute()` method of action class `ClientAction.class` returns `SUCCESS`, then this value is matched with the result name `success`. After matching the result value, the `client.jsp` is used to display the result. If the `execute()` method of action class returns `ERROR` then this value is matched with the particular result name value and renders a JSP page `error.jsp` to display the result. Similarly, in conversions issues, the Interceptors involved may return `INPUT` as result code. For this we have configured a result with the name `input`, which again brings `error.jsp` page for you.

Configuring Struts 2 in web.xml

You now need to configure your Web application according to the components within your directory structure. To do this, you must make some simple changes in the `web.xml` file. Struts 2 Web configuration uses `<filters>` in place of `<servlet>`. It is better for us to use `/*` as the URI pattern. This makes sure that all kinds of request patterns are served by `org.apache.struts2.dispatcher.FilterDispatcher`. Update your copy of `web.xml` with the configuration shown in Listing 2.6 and make sure that it is placed in the `WEB-INF` folder of your application

Here's the filter mapping, given in Listing 2.6, for `web.xml` (you can find `web.xml` file in `Code\Chapter 2\Struts2Application\WEB-INF` folder in CD):

Listing 2.6: `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

<display-name>Struts 2 Application</display-name>

<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

This concludes the creation of different code files for our sample Struts 2 based Web application. Make sure that all JSPs, class files, and JARs are placed according to what has been shown in Figure 2.2. Now let's deploy our application on the Tomcat Web server being used here.

Deploying Struts 2 Application

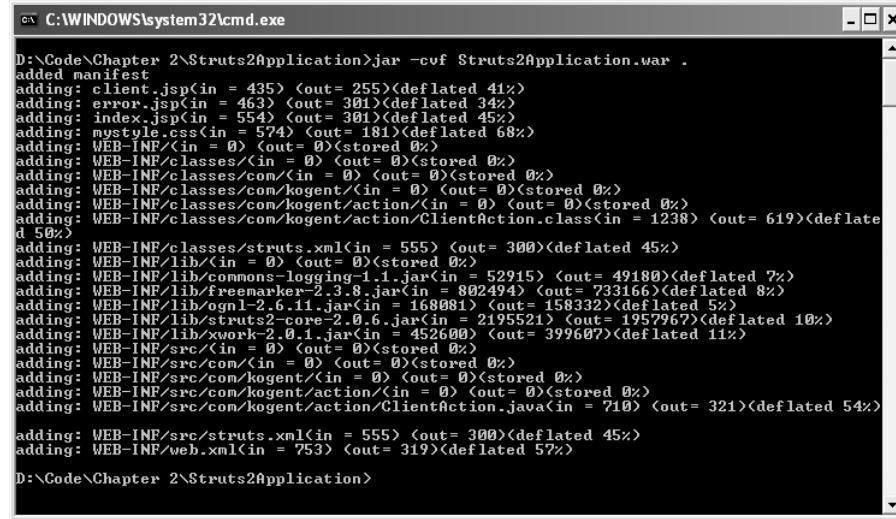
Any Web application needs to be deployed on some Web server or any application server which are compatible with the technologies used in the development of application. You can use any Web server with J2EE compatible container but here we are using Tomcat 5.5.23 as Web server to host our application. There are three different ways to deploy your Web application on Tomcat:

- ❑ You could create your application folder **Struts2Application** inside **webapp** folder of your Tomcat installation. In this case, the application is automatically deployed when you start your server.
- ❑ The second option is to develop your application anywhere on your system, as we have done in **D:\Code\Chapter 2**, and just copy the application folder **Struts2Application** into the **webapp** folder of your Tomcat installation. In that way the application is automatically deployed.
- ❑ The final and standard way is to create a WAR file for your application and deploy it using Tomcat Manager Console. You can also place the created WAR file into **webapp** folder of your Tomcat installation. Sometimes, our Web Server may not be available on the local machine and we may not access **webapp** folder of the Server on the remote machine. So, using Tomcat Manager Console is the best way to deploy the application after creating a WAR file of the application.

You can create a WAR file for the application using the following command:

```
D:\Code\Chapter 2\Struts2Application>jar -cvf Struts2Application.war .
```

You can see the code being executed in Figure 2.3. The name of the WAR file created here is **Struts2Application.war**, same as that of the application folder. You can give a name of your choice also.



```
C:\WINDOWS\system32\cmd.exe
D:\Code\Chapter 2\Struts2Application>jar -cvf Struts2Application.war .
added manifest
adding: client.jsp<in = 435> <out= 255><deflated 41%>
adding: error.jsp<in = 463> <out= 301><deflated 34%>
adding: index.jsp<in = 554> <out= 301><deflated 45%>
adding: mystyle.css<in = 574> <out= 181><deflated 68%>
adding: WEB-INF/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/classes/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/classes/com/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/classes/com/kognent/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/classes/com/kognent/action/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/classes/com/kognent/action/ClientAction.class<in = 1238> <out= 619><deflated 50%>
adding: WEB-INF/classes/struts.xml<in = 555> <out= 300><deflated 45%>
adding: WEB-INF/lib/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/lib/commons-logging-1.1.jar<in = 52915> <out= 49180><deflated 7%>
adding: WEB-INF/lib/freemarker-2.3.8.jar<in = 802494> <out= 733166><deflated 8%>
adding: WEB-INF/lib/ognl-2.6.11.jar<in = 168081> <out= 158332><deflated 5%>
adding: WEB-INF/lib/struts2-core-2.0.6.jar<in = 2195521> <out= 1957967><deflated 10%>
adding: WEB-INF/lib/xwork-2.0.1.jar<in = 452600> <out= 399607><deflated 11%>
adding: WEB-INF/src/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/src/com/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/src/com/kognent/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/src/com/kognent/action/<in = 0> <out= 0><stored 0%>
adding: WEB-INF/src/com/kognent/action/ClientAction.java<in = 710> <out= 321><deflated 54%>
adding: WEB-INF/src/struts.xml<in = 555> <out= 300><deflated 45%>
adding: WEB-INF/web.xml<in = 753> <out= 319><deflated 57%>
D:\Code\Chapter 2\Struts2Application>
```

Figure 2.3: Creating WAR file of application.

Now we can deploy this WAR file using the following steps:

1. Start your Web Server.
2. Access the Tomcat Manager Console after entering the URL <http://localhost:8080/> and proper login to see the list of applications being hosted by the server.
3. Browse to the location of WAR file created and select it to deploy (Figure 2.4)

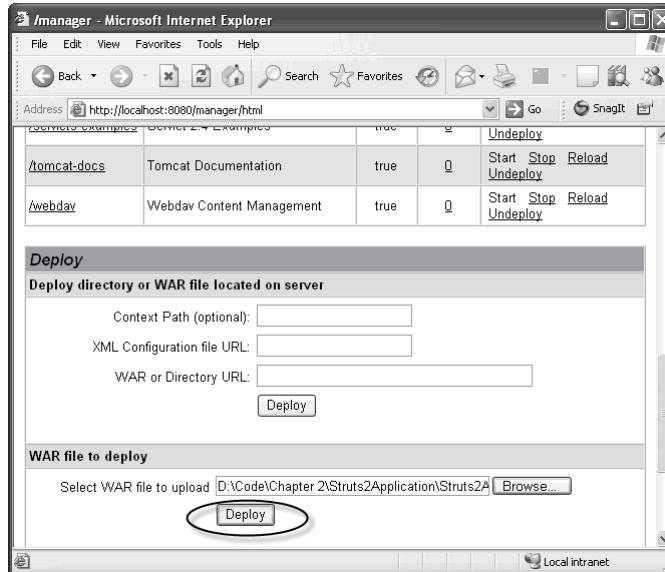


Figure 2.4: Deploying Struts2Applicaiton.war file on Tomcat server.

4. Now click on 'Deploy' button (encircled) to deploy the application (Figure 2.4).

NOTE

You may have your Web server listening to another port other than 8080. So change the port number here accordingly.

The application is now ready to be run after its successful deployment on the Web server.

Running Struts 2 Application

A deployed Struts 2 application can be accessed similar to other Web applications. You can access your first Struts 2 Web application that has been deployed on Tomcat Server by using the following URL in the address bar of your Web browser:

`http://localhost:8080/Struts2Application/`

The name of the WAR file should be used as application name here. If everything is correct, you'll see a page similar to Figure 2.5.

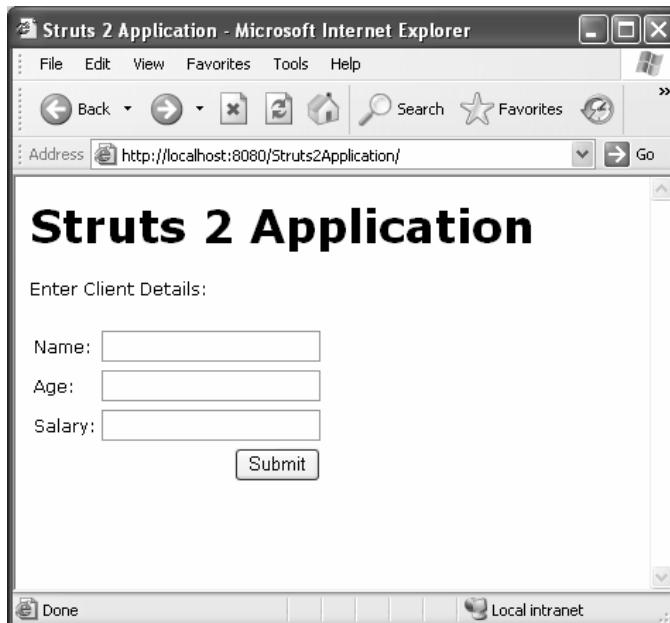


Figure 2.5: The index.jsp showing form with input fields.

When this page loads, the following events occur:

- ❑ The `<s:form>` of `index.jsp` creates the necessary HTML from page. Enter value in the field correctly and click the 'Submit' button.
- ❑ After submitting `index.jsp`, the browser invokes the URL name of `<s:form />` tags with its `action` attribute value. The JSP/Servlet container receives this request and looks for `<filter-mapping>` with the matching `<url-pattern>` in the `web.xml`. The container finds the following entry, which tells the container to send the request to a Servlet filter that is deployed with filter-name `struts2`:

```
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- ❑ The Web container finds the following `<filter>` entry with the `<filter-name>` that points to `FilterDispatcher`, which acts as a Controller for your Struts application:

```
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
</filter>
```

- ❑ After submitting the form, the value of action attribute of `<s:form action="clientAction">` is matched with the action mapped in `struts.xml` file with `name=clientAction`. The `FilterDispatcher` looks for an action mapping entry in the `struts.xml` file and finds the following entry:

```
<action name="clientAction" class="com.kogent.action.ClientAction">
    <result name="success">/client.jsp</result>
    <result name="error">/error.jsp</result>
    <result name="input">/error.jsp</result>
</action>
```

- ❑ Using action mapping in `struts.xml`, the appropriate action class, `ClientAction` action class, is instantiated which is responsible for processing data sent from `index.jsp` and forwards the result to the specified result view. Before the `execute()` method of `ClientAction` class is executed, all input properties of the action class are populated with the input parameters having the same name. For example, the request parameter name will be used by `setName()` method to set the name properties of the action class. This automatic setting of properties is assured by the use of some Interceptors.

NOTE

In Chapter 4, we'll be discussing Interceptors in detail. Here we can only tell you that we using defaultStack as Interceptor stack.

- ❑ If the `execute()` method of action `ClientAction` returns `SUCCESS` value, then the `client.jsp` is displayed as a result. The String value which is returned by `execute()` method is matched with the result name attribute of the results configured for the associated action in the `struts.xml` file. You'll see a page similar to the one shown in Figure 2.6.



Figure 2.6: Output of client.jsp showing submitted details.

- ❑ If the `execute()` method of action `ClientAction` returns `ERROR` value, the `error.jsp` is displayed as a result. The String value, which is returned by `execute ()` method, is matched with the result name attribute of the `struts.xml` file. In case of getting `INPUT` as result code by any Interceptor, the result configured with `name="input"` will be displayed, which happens to be `error.jsp` again. Then you'll see a page similar to the one shown in Figure 2.7.



Figure 2.7: The error.jsp page showing error messages.

Chapter 2

With this we complete our first Struts 2 based Web application. In this example, although we have not covered all the components of Struts 2, yet we have given you an idea of the Web application based on Struts 2 Framework. The different components of Struts 2, such as Interceptors, results, tag libraries etc. will be discussed in later chapters.

This chapter focussed on the basic requirement, standards, and techniques to develop a Struts 2 based Web application, along with downloading the latest version of Struts 2 APIs from the Web source, the platform requirement for developing a Struts 2 based Web application, and an idea about developing different components of a Struts 2 application. The discussion is supported by a running application, Struts2Application.

The next chapter will deal with a detail discussion on the action classes created in Struts 2 based applications and concentrate on the different concepts regarding creation, configuration, and use of action classes with supporting Struts 2 APIs available.



3

Creating Actions in Struts 2

If you need an immediate solution to:

See page:

Developing a Hello Action Example	67
Using Action as ActionForm	72
Configuring Action Class Method	82
Creating LoginAction with ApplicationAware and SessionAware Interface	84
Creating LogoutAction—a POJO Action	88

In Depth

Struts 2 is a Front Controller based framework, i.e. there is a single controller which handles all requests and invokes appropriate class containing some business function-specific logic implementations. These classes, which are executed to perform some requested functions here, are known as *Action classes*. These action classes help in applying separation of concerns as we can see a required functionality as a sequence of actions and each action class being executed for a specific functionality. We can have a number of action classes embedding different business logics for a particular user action. The way an action class is executed in the framework environment by the controller artifacts, has made Struts 2 more powerful and more sophisticated. The flow of execution of an action class involves a sequential invocation of different methods. This sequence of method invocations are handled by different Interceptors, which simply preprocess the request before executing the main logic embedded in the action class.

The concept of handling actions, creating action classes in Struts 2 Framework is similar to the actions implemented in Struts 1. But the way action classes are created, configured and used, is quite different in Struts 2 when compared with the same in Struts 1. Unlike Struts 1, the Struts 2 actions can be designed in more ways, which further makes development of this component easier. Struts 2 has introduced a simpler implementation for its action class which can be created by implementing some interface, by extending some class or even by using POJOs (Plain Old Java Objects). The implementation of Inversion of Control (IoC) through enabler interfaces allows the handling of only optional and custom services. In few words, the creation and configuration of action classes have become more flexible and simpler in Struts 2.

In this chapter, we have covered different aspects of creating actions and configuring them in Struts 2 Framework. The discussion here includes different interfaces and classes from Struts 2 APIs which are used to develop actions in Struts 2. In addition, all available features of Struts 2 actions have been thoroughly explained with suitable examples. Let's forge ahead to find out how actions are created, configured, and used in Struts 2 Framework and how Struts 2 actions are easier to develop and configure.

Handling Struts 2 Actions

Any required function can be performed with the execution of a single action class or set of action classes. Like Struts 1, in Struts 2 framework too, we can map a user action with an action class which consequently serves the user request. Hence, we need to create different action classes in a single Struts 2 based Web application. An action class is created to handle a given action, so you can have actions like AddEmployeeAction, GetProductAction etc. Struts 2 action classes are similar to the action classes created in Struts 1 Framework in many aspects. For example, the action classes are configured to be executed when a particular request for these actions is submitted. Both Struts 1 and Struts 2 actions have `execute()` method as the default entry point where the execution of an action class starts. The return from the action class, whether we are talking about Struts 1 or Struts 2, is used to decide the next process, which may include rendering of the next JSP page or invocation of another action. In addition to these basic conceptual similarities with Struts 1, the action classes in Struts 2 have a different way of

implementation, which makes it easy to develop and test. You can see the difference between the signature of execute() method of action classes in Struts 1 and Struts 2 as shown here:

```
//execute() method of Struts 1 action classes
public ActionForward execute (ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
                             throws IOException, ServletException {
    //some logic implementation
}

//execute() method of Struts 2 action classes
public String execute() throws Exception {
    //some logic implementation
}
```

As compared to Struts 1 action classes, this is one of the numerous points where you can find the structure of Struts 2 action class simple. The return type of execute() method of a Struts 2 action class is a String. Though, it can return any string, some standards are ‘success’, ‘input’, ‘error’, ‘none’, and ‘login’. The result configured for this action having the name matching this returned string is executed next.

In Struts 1, an action class can be created by extending org.apache.struts.action.Action class only, while in Struts 2 there are different ways to create an action class. Struts 2 provides flexibility in creating a new action class by implementing com.opensymphony.xwork2.Action interface or by extending com.opensymphony.xwork2.ActionSupport class. In addition, Struts 2 supports using POJOs (Plain Old Java Objects) as Actions, which enable us to create action classes without implementing any interface and extending any class. These action classes are known as POJO actions and does not depend on any container.

The only convention to be followed is the availability of execute() method in the action classes. The execution of action class starts with invocation of its execute() method, but now in Struts 2, we can invoke any method of action class having signature String methodName() using configuration. This is again a new option which can be implemented.

We can create different number of action classes for different use cases. For every request, a matching action mapping is searched and the associated action class methods are invoked. Struts 2 actions are not singletons, i.e. for each request a different instance of action class is created. Hence, Struts 2 action need not be thread safe like Struts 1 actions.

The action classes are now simple and do not depend on other APIs. Hence, testing of these classes is much easier now. So how can we access different objects to work with, like request, response, and session, etc. The solution of this problem is *Dependency Injection* pattern. The Dependency Injection mechanism used in Struts 2 is *Interface Injection*. This means, we can implement various interfaces, which can expose different methods to the action class to make the required objects available in the action class. These exposed methods should be invoked in the predefined order, which is assured by the use of appropriate Interceptors configured for the action.

Let's start the discussion about the various interfaces and classes, which are used for the creation, configuration and execution of action class.

Action Interface

Though, we can create action classes without implementing any interface and extending any class, Struts 2 Framework provides an interface which can be used to create action classes. This is the `com.opensymphony.xwork2.Action` interface which can be used as helper interface exposing the `execute()` method to the action class implementing it. In addition, Action interface provides common results ('success', 'error', etc.) as string constants which are listed in Table 3.1.

Table 3.1: String constants for common results	
Field Name	Description
<code>static String ERROR</code>	Execution of action failed
<code>static String INPUT</code>	The action execution requires more input in order to succeed
<code>static String LOGIN</code>	The action execution needs that user is not logged in
<code>static String NONE</code>	Successful execution but no view
<code>static String SUCCESS</code>	Execution of action was successful

The only method of Action interface is `String execute()` and all action classes implementing this interface must override this method. A sample action class created by using Action interface is shown here:

```
import com.opensymphony.xwork2.Action;
public class Somection implements Action{
    public String execute() throws Exception {
        //Some business logic.
        return SUCCESS;
    }
}
```

We can use string constants, like `SUCCESS`, `ERROR`, instead of using strings as seen in the sample action class code shown earlier. Though the use of this interface is not required, it is here to assist you when creating your first action. We can use any Java class (POJO) as action, which agrees with the contract defined by Action interface.

ActionSupport Class

Creation of action classes by implementing `com.opensymphony.xwork2.Action` interface is useful for simple action classes only. For adding further functionalities and different methods to our action class, Struts 2 Framework provides a class, which has default implementations for various services required by common action classes. This is the `com.opensymphony.xwork2.ActionSupport` class. `ActionSupport` class implements interfaces, like `Action`, `LocaleProvider`, `TextProvider`,

Validateable, ValidationAware, and Serializable providing implementations to all methods from these interfaces. Hence, we can start developing our action by extending ActionSupport class and just overriding the required methods, accordingly.

ActionSupport class provides basic implementation for common actions. In addition to Action interface, other interfaces implemented by ActionSupport class and their methods are listed here.

Validateable Interface

The com.opensymphony.xwork2.Validateable interface provides a validate() method that allows your action to be validated. You must override this validate() method in your action class to implement your logic to validate the data.

LocaleProvider Interface

The com.opensymphony.xwork2.LocaleProvider interface provides a getLocale() method to provide the Locale to be used for getting localized messages. When LocaleProvider interface is implemented in a class, the class has its own Locale. It is used in an action to override the default locale, whenever needed. A Locale is an object, which represents a specific geographical, political, or cultural region.

ValidationAware Interface

The com.opensymphony.xwork2.ValidationAware interface provides methods for saving and retrieving action class level and field-level error messages. Collections are used to store/retrieve action-level error messages and Maps are used to store/retrieve field-level error messages. See Table 3.2 for the description of different methods of ValidationAware interface.

Table 3.2: Methods of ValidationAware interface

Methods	Description
void addActionError(String anErrorMessage)	It adds an Action-level error message to this Action
void addActionMessage(String aMessage)	It adds an Action-level message to this Action
void addFieldError(String fieldName, String errorMessage)	It adds an error message for a given field
Collection getActionErrors()	It gets the Collection of Action-level error messages for this action
Collection getActionMessages()	It gets the Collection of Action-level messages for this action
Map getFieldErrors()	It gets the field specific errors associated with this action
boolean hasActionErrors()	It checks whether there are any Action-level error messages

Table 3.2: Methods of ValidationAware interface

Methods	Description
boolean hasActionMessages()	It checks whether there are any Action-level messages
boolean hasErrors()	It checks whether there are any errors (action level or field level) set or not
boolean hasFieldErrors()	It checks whether there are any field errors associated with this action
void setActionErrors(Collection errorMessages)	It sets the Collection of Action-level String error messages
void setActionMessages(Collection messages)	It sets the Collection of Action-level String messages (not errors)
void setFieldErrors(Map errorMap)	It sets the field error map of fieldname (String) to Collection of String error messages

TextProvider Interface

The `com.opensymphony.xwork2.TextProvider` interface provides methods for getting localized message texts. `TextProvider` is used to access text messages in the resource bundle. Table 3.3 describes various methods of `TextProvider` interface.

Table 3.3: Methods of TextProvider interface

Methods	Description
String getText(String key)	It gets a message based on a message key, or null if no message is found
String getText(String key, List args)	It gets a message based on a key using the supplied args, as defined in <code>MessageFormat</code> , or null if no message is found
String getText(String key, String defaultValue)	It gets a message based on a key, or, if the message is not found, a supplied default value is returned
String getText(String key, String[] args)	It gets a message based on a key using the supplied args, as defined in <code>MessageFormat</code> , or null if no message is found
String getText(String key, String defaultValue, List args)	It gets a message based on a key using the supplied args, as defined in <code>MessageFormat</code> , or, if the message is not found, a supplied default value is returned
String getText	It gets a message based on a key using the supplied args,

Table 3.3: Methods of TextProvider interface	
Methods	Description
(String key, String defaultValue, List args, ValueStack stack)	as defined in MessageFormat, or, if the message is not found, a supplied default value is returned
String getText (String key, String defaultValue, String obj)	It gets a message based on a key using the supplied obj, as defined in MessageFormat, or, if the message is not found, a supplied default value is returned
String getText (String key, String defaultValue, String[] args)	It gets a message based on a key using the supplied args, as defined in MessageFormat, or, if the message is not found, a supplied default value is returned
String getText (String key, String defaultValue, String[] args, ValueStack stack)	It gets a message based on a key using the supplied args, as defined in MessageFormat, or, if the message is not found, a supplied default value is returned
ResourceBundle getTexts()	It gets the resource bundle associated with the implementing class (usually an action)
ResourceBundle getTexts (String bundleName)	It gets the named bundle, such as “com/acme/Foo”

The `com.opensymphony.xwork2.ActionSupport` class provides default definitions of methods of all interfaces implemented by it. Hence, we can create our action classes by extending `ActionSupport` classes and use these methods in our action classes.

Here's the sample action class creation, given in Listing 3.1, by extending the `ActionSupport` class:

Listing 3.1: A sample action class

```

package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
}

```

```

public void setPassword(String password) {
    this.password = password;
}

public String execute() throws Exception {
    if(username.equals(password))
        return SUCCESS;
    else{
        this.addActionError(getText("app.invalid"));
        return ERROR;
    }
}

public void validate() {
    if ( (username == null ) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
    }
    if ( (password == null ) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
    }
}

```

The `validate()` method can be overridden for our own validation logic implementation. The `DefaultWorkflowInterceptor` Interceptor makes sure that the `validate()` method is executed before `execute()` method, and if data is invalid the `execute()` is never executed.

The order of execution in the workflow is as follows:

- ❑ If the action to be executed implements `Validateable`, the action's `validate()` method is executed before executing logic in `execute()` method.
- ❑ Next, if the action implements `ValidationAware`, the action's `hasErrors()` method is called.
- ❑ If the `hasErrors()` method returns true, the Interceptor stops further execution and returns `Action.INPUT`. This searches for some result having name="input" to be rendered.

Therefore, to make all methods work, the Interceptor stack must contain workflow Interceptor configured for the action. If our package extends `struts-default` package defined in `struts-default.xml` file, we'll use `defaultStack` as the Interceptor stack for all actions here even if we do not configure any Interceptor in this package. If you give your own Interceptor configuration make sure that the workflow Interceptor is included to enable this flow of execution.

In this way, we find that creation of an action class, using `ActionSupport` as super class, is a better option as it eliminates the need for implementing a number of Interceptor, which are required to apply validation and internationalization.

Action Mapping

An action mapping is a configuration file entry that is associated with an action name and action class. It provides a mapping between HTTP requests and action invocation requests and vice-versa. For a given `HttpServletRequest`, the `org.apache.struts2.dispatcher.mapper.ActionMapper` interface may return null if no action invocation request matches; otherwise it may return an instance of `org.apache.struts2.dispatcher.mapper.ActionMapping` class that describes an action invocation for the Web application.

ActionMapping is a simple class that holds the action mapping information used to invoke a Struts action. Tables 3.4 and 3.5 show the fields and methods, respectively, of this class.

Table 3.4: Fields of ActionMapping class

Field Name	Description
private String method	It is method name
private String name	It is name of the action
private String namespace	It is namespace of the action
private Map params	It is optional set of parameters
private Result result	It is result of the action

Table 3.5: Methods of ActionMapping class

Methods	Description
String getMethod()	It returns method name
String getName()	It returns name of the action
String getNamespace()	It returns namespace for the action
Map getParams()	It returns Map containing name/value pairs for action parameters
Result getResult()	It returns result
void setMethod (String method)	It sets method name
void setName(String name)	It sets action name
void setNamespace (String namespace)	It sets namespace name
void setParams (Map params)	It sets a Map of parameters
void setResult (Result result)	It sets Result

The action mappings are the basic unit-of-work in the Struts 2 Framework, which maps an identifier to a handler class. When a request matches with the action's name, the Struts 2 Framework uses the mapping to determine how to process the request. The action mapping specifies a set of result types, a set of exception handlers, and an Interceptor stack, but only the name attribute is required and other attributes can be provided at package scope also.

In a Web application, every resource must be referred to a Uniform Resource Identifier (URI), which includes HTML pages, JSP pages, and any custom actions. The Struts 2 Framework provides this facility by using action mapping, which is defined in the `struts.xml` configuration file.

Here's a sample action mapping, given in Listing 3.2, in `struts.xml` file:

Listing 3.2: Sample action mapping in `struts.xml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">
    <action name="login" class="com.kogent.action.LoginAction">

        <interceptor-ref name="exception"/>
        <interceptor-ref name="params"/>
        <interceptor-ref name="workflow"/>

        <result name="success"/>/login_success.jsp</result>
        <result name="error"/>/login.jsp</result>
        <result name="input"/>/login.jsp</result>
    </action>
</package>
</struts>
```

Default Action

Default action is used when an action is requested in the `struts.xml` configuration, and the Struts2 Framework couldn't map the request with the given requested action name. However, you can specify a default action, if you don't want some error page to appear. In other words, if no action matches in the `struts.xml` then the default action is used to display your JSP result.

The following code snippet shows a default action configured in your `strut.xml` file:

```
<package name="my-default" extends="struts-default">
<default-action-ref name="somedefault">

    <action name="somedefault" >
        <result>/some_page.jsp</result>
    </action>
</package>
```

If no action matches in the `strut.xml`, then the default action `somedefault` is used to generate the Result View. There should be only one default action per namespace.

Action Methods

The main method in any action class is its `String execute()` method, which is executed by default to accomplish some defined business logic. But Struts 2 action classes provide flexibility of defining some other `String methodName()` like methods and invoking them directly using action

configuration. Hence, for different requests, we can execute different methods of the same action class, provided these methods have a signature similar to `execute()` method, i.e. `String methodName()`. This eliminates the need for creating different action classes for different actions. We can group similar type of actions into a single action class, e.g. two actions for adding a employee (`AddEmployeeAction`) and editing an employee (`EditEmployeeAction`). The `execute()` method of both these classes will operate on a similar set of data and require similar set of validation rules to be followed. We can create a single action class (`UpdateEmployeeAction`) and two `execute()` methods of the previous two action classes can be converted into two different methods of this new class. These methods can be `String addEmployee()` and `String editEmployee()`.

Here's the sample structure, given in Listing 3.3, of this `UpdateEmployeeAction` class.:

Listing 3.3: An action class with `String methodName()` methods

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;

public class UpdateEmployeeAction extends ActionSupport {

    public String execute() throws Exception {
        //some logic
        return SUCCESS;
    }
    public String addEmployee() throws Exception {
        //Logic fo add new employee
        return SUCCESS;
    }
    public String editEmployee() throws Exception {
        //Logic to edit employee
        return SUCCESS;
    }
}
```

We can provide two action mappings to add and edit employee, using the same action class but configuring different methods to be invoked. This is shown in Listing 3.4.

Listing 3.4: Configuring different action methods in struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">
    <action name="add"
        class="com.kogent.action.UpdateEmployeeAction"
        method="addEmployee">

        <result name="success">/add_success.jsp</result>
        <result name="error">/add.jsp</result>
        <result name="input">/add.jsp</result>
    </action>
```

```

<action name="edit"
       class="com.kogent.action.UpdateEmployeeAction"
       method="editEmployee">
    <result name="success">/edit_success.jsp</result>
    <result name="error">/edit.jsp</result>
    <result name="input">/edit.jsp</result>
</action>
</package>
</struts>

```

The two action mappings provided in this example will invoke different methods of the same action class. So, this is yet another example of changing traditional approach and allowing some more flexibility, while developing actions in Struts 2.

ActionContext Class

An action context can be defined here as a container, which contains objects that an action requires for its execution. This context is represented by `ActionContext` class and can provide objects, like request, response, session, parameters locale, etc. The objects stored in the `ActionContext` are unique per thread as this class is thread locale. We can obtain the reference of this class using its own static `getContext()` method as shown here:

```
ActionContext context = ActionContext.getContext();
```

The `execute()` method of Struts 2 action classes does not take any argument and, hence, we need some technique to access objects, like request, response, session, etc. `ActionContext` helps in obtaining these objects in our action class. The following code sample shows the use of `ActionContext` class:

```

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ActionContext;
import org.apache.struts2.ServletActionContext;
public class SomeAction extends ActionSupport {

    .
    .
    public String execute() throws Exception {
        .
        .
        .
        ActionContext acx=ActionContext.getContext();

        HttpServletRequest request=
            (HttpServletRequest)acx.get(ServletActionContext.HTTP_REQUEST);
        request.setAttribute("name", "John");

        HttpSession session=
            (HttpSession)acx.get(ServletActionContext.SESSION);
        session.setAttribute("user", "kogent");

        return SUCCESS;
    }
}

```

After obtaining an `ActionContext` object using `getContext()` method, we can use the `get()` method on this object to obtain some other useful objects. The `ActionContext.get()` method returns an object stored in the current `ActionContext` by looking up the value for the key passed as an argument to it. Tables 3.6 and 3.7 show the fields and methods, respectively, of the `ActionContext` class.

Table 3.6: Fields of ActionContext class	
Field Name	Description
<code>static String ACTION_INVOCATION</code>	It is constant for the action's invocation context
<code>static String ACTION_NAME</code>	It is constant for the name of the action being executed
<code>(package private) static ThreadLocal actionContext</code>	It is a static thread local variable which is used by different threads independently
<code>static String APPLICATION</code>	It is constant for the action's application context
<code>(package private) Map context</code>	It is a Map type of field to be used to store key/value pairs for a given context.
<code>static String CONVERSION_ERRORS</code>	It is constant for the map of type conversion errors
<code>static String LOCALE</code>	It is constant for the action's locale
<code>static String PARAMETERS</code>	It is constant for the action's parameters
<code>static String SESSION</code>	It is constant for the action's session
<code>static String TYPE_CONVERTER</code>	It is constant for the action's type converter
<code>static String VALUE_STACK</code>	It is constant for the OGNL Value Stack

Table 3.7: Methods of ActionContext class	
Method	Description
<code>Object get(Object key)</code>	It returns a value that is stored in the current <code>ActionContext</code> by doing a lookup using the value's key
<code>ActionInvocation getActionInvocation()</code>	It gets the action invocation (the execution state)
<code>Map getApplication()</code>	It returns a Map of the <code>ServletContext</code> when in a Servlet environment or a generic application level Map otherwise
<code>static ActionContext getContext()</code>	It returns the <code>ActionContext</code> specific to the current thread
<code>Map getContextMap()</code>	It gets the context map

Table 3.7: Methods of ActionContext class

Method	Description
Map getConversionErrors()	It gets the map of conversion errors, which occurred while executing the action
Locale getLocale()	It gets the Locale of the current action
String getName()	It gets the name of the current Action
Map getParameters()	It returns a Map of the HttpServletRequest parameters when in a Servlet environment or a generic Map of parameters, otherwise
Map getSession()	It gets the Map of HttpSession values when in a Servlet environment or a generic session map, otherwise
ValueStack getValueStack()	It gets the OGNL Value Stack
void put(Object key, Object value)	It stores a value in the current ActionContext
void setActionInvocation (ActionInvocation actionInvocation)	It sets the action invocation (the execution state)
void setApplication (Map application)	It sets the action's application context
static void setContext (ActionContext context)	It sets the action context for the current thread
void setContextMap (Map contextMap)	It sets the action's context map
void setConversionErrors (Map conversionErrors)	It sets conversion errors which occurred when executing the action
void setLocale (Locale locale)	It sets the Locale for the current action
void setName(String name)	It sets the name of the current Action in the ActionContext
void setParameters (Map parameters)	It sets the action parameters
void setSession (Map session)	It sets a map of action session values
void setValueStack (ValueStack stack)	It sets the OGNL Value Stack

Another class, which stores web-specific context information for actions is `org.apache.struts2.ServletActionContext` class. `ServletActionContext` class is a subclass of `ActionContext`. In addition to extending fields and methods from `ActionContext`, the `ServletActionContext` class also implements `org.apache.struts2.StrutsStatics` interface. Tables 3.8 and 3.9 show the fields and methods, respectively, of `ServletActionContext` class.

Table 3.8: Fields of `ServletActionContext` class

Field Name	Description
<code>static String ACTION_MAPPING</code>	A static final String type field having value "struts.actionMapping"
<code>private static long serialVersionUID</code>	Serial version id of the <code>ServletActionContext</code> class and its value is -666854718275106687L
<code>static String STRUTS_VALUESTACK_KEY</code>	Key for the associated Value Stack and its value is "struts.valueStack"

Table 3.9: Methods of `ServletActionContext` class

Methods	Description
<code>static ActionContext getActionContext(HttpServletRequest req)</code>	It gets the current action context
<code>static ActionMapping getActionMapping()</code>	It gets the action mapping for this context
<code>static PageContext getPageContext()</code>	It returns the HTTP page context
<code>static HttpServletRequest getRequest()</code>	It gets the HTTP Servlet request object
<code>static HttpServletResponse getResponse()</code>	It gets the HTTP servlet response object
<code>static ServletContext getServletContext()</code>	It gets the Servlet context
<code>static ValueStack getValueStack(HttpServletRequest req)</code>	It gets the current Value Stack for this request
<code>static void setRequest(HttpServletRequest request)</code>	It sets the HTTP Servlet request object

Table 3.9: Methods of ServletActionContext class

Methods	Description
<code>static void setResponse (HttpServletResponse response)</code>	It sets the HTTP Servlet response object
<code>static void setServletContext(Servlet Context servletContext)</code>	It sets the current Servlet context object

In addition to these fields and methods, `ServletActionContext` class inherits all fields and methods from its parent class, i.e. `ActionContext` and the `StrutsStatics` interface implemented by it. The `StrutsStatics` interface contains constants used by Struts. Table 3.10 shows these constants fields.

Table 3.10: Constants fields of StrutsStatics interface

Field Name	Description
<code>static String HTTP_REQUEST</code>	It is constant for the HTTP request object
<code>static String HTTP_RESPONSE</code>	It is constant for the HTTP response object
<code>static String PAGE_CONTEXT</code>	It is constant for the JSP page context
<code>static String SERVLET_CONTEXT</code>	It is constant for the Servlet context object
<code>static String SERVLET_DISPATCHER</code>	It is constant for an HTTP request dispatcher
<code>static String STRUTS_PORTLET_CONTEXT</code>	It is constant for the PortletContext object

Using POJO as Action

POJO (Plain Old Java Objects) are ordinary Java objects which do not implement any interface or extend any other Java class and hence does not depend on other APIs. The creation of POJO objects is simple and hence better to design, debug, and test. Struts 2 provides the facility of taking POJO as actions. Hence, the classes which do not extend any pre-specified classes and do not implement any pre-specified interfaces can be configured as action class and can be `executed()` in the same manner as other action classes created by implementing `Action` interface or by extending `ActionSupport` class. The methods of POJO action must agree with the contracts defined for them by Struts 2, for example the signature of `execute()` method which returns `String` and takes no argument.. A simple example of POJO as action is shown here:

```
public class SomeAction{
    public String execute() throws Exception {
        return "success";
    }
}
```

The importance of POJO as action is that we don't need to use extra objects in the Struts 2 Framework. It is faster, simpler, and easier to develop. It also shows how to organize and encapsulate the domain logic, access the database, manage transactions and handle the database concurrency. Testing of Struts 2 action classes can be done outside the container.

Using Actions as ActionForms

In Struts 1, an `ActionForm` object is used to retain input data. The `ActionForms` in Struts 1 extends a base class `ActionForm`. These are JavaBeans like classes with `validate()` and `reset()` methods. The developers had to create these additional classes to be associated with each action. These `ActionForms` are automatically populated with corresponding data submitted from JSP pages before executing action classes.

In Struts 2, the action class properties can be used as input properties. This eliminates the need of any `ActionForm`, like classes in Struts 2. The action class itself can describe the list of properties with getter and setter methods for them. These properties can be accessed in a web page using Struts 2 tag libraries. Hence, we can create different action classes with getters/setters for their own set of input fields. It means the actions in Struts 2 can also behave as `ActionForms` of Struts 2.

In addition, Struts 2 also supports the `ActionForm` pattern and POJO form objects. The concept of `ModelDriven` action is used to inject the POJO form object (model) into the action. This is discussed later in this section.

Here's a simple action, given in Listing 3.5, with different input properties and getter/setter methods.

Listing 3.5: An action class with input fields

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    //Input properties;
    private String username;
    private String password;

    //Getter and Setter methods for input properties.
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
}
```

```

    }

    public void setPassword(String password) {
        this.password = password;
    }

    //Simple execute() method containing business logic.
    public String execute() throws Exception {

        //Some logic.
        return SUCCESS;
    }
    public void validate() {
        //Logic to validate
    }

}

```

The two input properties created in Listing 3.5 are `username` and `password`. The action class also provides methods to set these input properties. These fields can automatically be populated with the data sent as request parameters with the same name, i.e. `username` and `password`. These input fields should be set prior to the execution of the `execute()` method.

Now there is a question for you. What is required for setting all input parameters before the `execute()` method is invoked? This is an Interceptor named `param (ParametersInterceptor)`. This Interceptor is part of `basicStack` and `defaultStack` Interceptor stack. So, in all applications where the `defaultStack` is being used as default Interceptor stack, we are using `param` Interceptor too. This Interceptor finds setter method for all incoming request parameters and invokes them. In this way, `ParametersInterceptor` sets all parameters on the Value Stack using `ValueStack.setValue(String, Object)` methods. The Interceptors are discussed in detail in Chapter 4.

Using ModelDriven Actions

Sometimes we may want to implement `ActionForm` like approach in which the data is held by a separate class, instead of having input properties in action class. But, as the data should be available for the action to process over it, the model of the action needs to be added to the action in its Value Stack making it available in the action. The model class created here can be any JavaBean like class. These classes are used as POJO forms in Struts 2.

This approach needs some additional implementations. The action class must implement the `com.opensymphony.xwork2.ModelDriven` interface. The only method which is exposed by `ModelDriven` interface is its `Object getModel()` method. This method provides the model to be pushed into the Value Stack.

Let's see a sample example of POJO form, which can be injected in the action as its model. Listing 3.6 shows a POJO form, which is a simple JavaBeans class:

Listing 3.6: A model class

```

package com.kogent;

public class User {
    String username;

```

```
        String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

The `LoginAction`, described earlier, can be converted into `ModelDriven` action by removing input properties with their getter/setter methods and implementing interface `ModelDriven <T>`. The only method of this interface is implemented in this action to return the object of the model. The returned model is pushed into the Value Stack of the action if not found null. Listing 3.7 shows a sample model driven action:

Listing 3.7: A model driven action

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.opensymphony.xwork2.Preparable;

import com.kogent.User;

public class LoginAction extends ActionSupport
    implements ModelDriven, Preparable {

    User user;

    public void prepare(){
        user=new User();
    }

    public User getModel() {
        return user;
    }

    public String execute() throws Exception {
        if(user.getUsername().equals("kogent")){
            return SUCCESS;
        }
        else{
            this.addActionError("app.invalid");
        }
    }
}
```

```

        return ERROR;
    }
}

public void validate() {
}
}

```

The `getModel()` method must be invoked earlier for populating its input properties with the data from request parameters. To make sure that the model is available in the Value Stack when input parameters are being set, the `ModelDrivenInterceptor` is configured for the action. This Interceptor should be configured before `ParametersInterceptor` and `StaticParametersInterceptor` in the Interceptor stack. We can use the `defaultStack`, which is the default Interceptor stack to be used. The model object to be pushed into stack cannot be null, hence, the model must be initialized before it is returned by `getModel()` method to be populated and pushed into Value Stack. Though, we can initialize model object at a number of points, Struts 2 provides an interface/Interceptor duo which can be used to prepare action for execution. The interface to be implemented here is `Preparable`, which exposes the `prepare()` method, and the configuration of `PrepareInterceptor` makes sure that this method is invoked. That is why, this sample model driven action implements `Preparable` and its `prepare()` method. So the flow of execution of various methods can be controlled by the position of Interceptors in the Interceptor stack. The `defaultStack` is the good one to be used. Chapter 4 discusses more about Interceptors.

Understanding Dependency Injection and Inversion of Control

Inversion of Control (IoC) and Dependency Injection (DI) are programming design patterns, which can be used to reduce coupling in computer programs. They follow the principles given here:

- You do not need to create your objects. You need to only describe how they should be created.
- You do not need to instantiate or directly locate the services needed by your components; instead, you only need to describe which services are needed by which components.

Dependency injection helps in removing the task of creating and linking objects from the objects themselves. This task is accomplished using a factory. IoC container provides the factory..The default factory used in struts 2 is `ObjectFactory`. Inversion of Control (IoC) is based on how services are defined and how they should locate with the other services on which they depend; obtaining, such services is dependent on a container. This container is responsible for various tasks, such as keeping a collection of the available services, providing the services they depend on, etc. IoC provides a way to inject logic into client code rather than writing it there.

There are two ways to implement IoC in Struts:

- With instantiation** – In this type of implementation, a given action Object is instantiated with the resource Object as a constructor parameter.
- Using an enabler interface** – In this implementation, the action class will implement an interface with some method like `setResources(ResourceObject r)`. This will allow the resource to be passed to the said action Object after it is instantiated.

Struts 2 Aware Interfaces

For getting the HTTP-specific object when you need them in your Action, Struts 2 uses a technique known as Dependency Injection and Inversion of Control. In this technique, the aware interfaces that we want to use are injected in the action class. These interfaces are called aware interfaces because the interface names always end with Aware. These interfaces have methods to set specific resources into the implementing class and to make the resource available.

Struts 2 implements Interface Injection which is one of the forms of Dependency Injection pattern. It is something like implementing the required interface, which exposes some methods to the implementing class. This enhances the capability of your class and helps in decoupling your class from other APIs, until it is required. Common aware interfaces that Struts 2 supports are discussed here.

ApplicationAware Interface

The `org.apache.struts2.interceptor.ApplicationAware` interface is used to expose a method to an action class which sets an Application Map object in this action class. This Map object contains different objects which are accessible from whole application, i.e. these are objects stored in application scope. The key/values added into this Map is available similar to other application scope attributes. The `ApplicationAware` interface has one method, `setApplication()`, which sets the map of application properties in implementing class.

Listing 3.8 shows an action that implements the `ApplicationAware` interface:

Listing 3.8: An action implementing `ApplicationAware`

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ApplicationAware;

public class SomeAction extends ActionSupport implements ApplicationAware {

    Map app;

    public void setApplication(Map app) {
        this.app=app;
    }

    public String execute() throws Exception {
        app.put("company", "Kogent Solutions Inc.");
        return SUCCESS;
    }
}
```

The Interceptor puts the Map application into this action before it is executed, so that it can just use the application.

All key/value pairs stored in Application Map can be accessed from any action or JSP page. The Struts 2 tags support the display of different application scope objects using a corresponding key. For example, the single key/value pair stored in Application Map app, shown in Listing 3.8, can be accessed in a JSP page as shown here:

```
<s:property value="#application.name"/>
or
<s:property value="#application['name']"/>
```

SessionAware Interface

The `org.apache.struts2.interceptor.SessionAware` interface is used to handle client session within your Action. The `SessionAware` interface has the method `setSession()` to set the Map of session attributes in the implementing action class. The action implementing `SessionAware` interface can access session attributes in the form of Session Map. We can add, remove, and obtain different session attributes by manipulating this Map. The interface is relevant only when the action is being used in Servlet environment. The implementation of the `SessionAware` interface by the action class makes this action class tied to Servlet API. This consequently makes action class difficult for unit testing.

Here's a sample action, given in Listing 3.9, for implementing `SessionAware` interface:

Listing 3.9: An action implementing `SessionAware`

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.SessionAware;

public class SomeAction extends ActionSupport implements SessionAware {

    Map session;

    public void setSession(Map session) {
        this.session=session;
    }

    public String execute() throws Exception {
        session.put("user", "John");
        return SUCCESS;
    }
}
```

The single key/value pair stored in session Map `session`, shown in Listing 3.9, can be accessed in a JSP page as shown here:

```
<s:property value="#session.user"/>
or
<s:property value="#session['user']"/>
```

ParameterAware Interface

The `org.apache.struts2.interceptor.ParameterAware` interface is used when an action wants to handle input parameters. All the input parameters with its name/value are set in a parameter Map. When the actions require the HTTP request parameter Map, then this interface is implemented within your Action. Another common use for this interface is to propagate parameters to internally instantiate data objects. All the parameter values for a given name will return String data, so the type of the objects in the map is `String[]` date type.

The `ParameterAware` interface has one method `setParameter(Map map)`, which sets the map of input parameters in the implementing class. Listing 3.10 shows an action that implements `ParameterAware` interface (the Interceptor puts a Map of parameters (name/value) which can be used in the action and the following JSP page also):

Listing 3.10: An action implementing `ParameterAware`

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ParameterAware;

public class SomeAction extends ActionSupport implements ParameterAware {
    Map params;

    public void setParameters(Map params) {
        this.params=params; }
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

The `params` is a Map containing all input parameters set as the key/value pairs and the action class can use this map to process input parameters or to propagate them to another object. These input parameters can also be displayed on JSP using the following syntax which shows the display of two input parameters, i.e. name and city:

```
<s:property value="#parameters.name"/>
<s:property value="#parameters.city"/>
or
<s:property value="#parameters['name']"/>
<s:property value="#parameters['city']"/>
```

ServletRequestAware Interface

The `HttpServletRequest` object is not available in the action. But this can be injected into the action by implementing `org.apache.struts2.interceptor.ServletRequestAware` interface. The exposed method `setServletRequest(HttpServletRequest request)` sets `HttpServletRequest` object in the action class. This allows an action to use the

HttpServletRequest object in a Servlet environment. Listing 3.11 shows an action that implements the ServletRequestAware interface:

Listing 3.11: An action implementing ServletRequestAware

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletRequestAware;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class SomeAction extends ActionSupport implements ServletRequestAware{
    HttpServletRequest request;
    public void setServletRequest(HttpServletRequest request) {
        this.request=request; }
    public String execute() throws Exception {
        request.setAttribute("userid", 101);
        HttpSession session=request.getSession();
        return SUCCESS;
    }
}
```

ServletResponseAware Interface

The `org.apache.struts2.interceptor.ServletResponseAware` is similar to `ServletRequestAware` interface. The only difference is the object injected into action, which is `HttpServletResponse` here. This interface is used to inject the `HttpServletResponse` object into your action. This interface has the `setServletResponse(HttpServletResponse response)` method, which sets `HttpServletResponse` object. Listing 3.12 shows an action that implements the `ServletResponseAware` interface and gets an `HttpServletResponse` object to work with:

Listing 3.12: An action implementing ServletResponseAware

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletResponseAware;

import javax.servlet.http.HttpServletResponse;

public class SomeAction extends ActionSupport implements ServletResponseAware{

    HttpServletResponse response;

    public void setServletResponse(HttpServletResponse response) {
        this.response=response;
    }
}
```

```
public String execute() throws Exception {  
    response.setContentType("text/html");  
    int buffer=response.getBufferSize();  
    . . .  
    . . .  
    return SUCCESS;  
}  
}
```

In the “In Depth” section, we have discussed various ways of designing a Struts 2 action, and how a simple Struts 2 action can be made capable of handling different functionalities by implementing Interface Injection, which is a form of Dependency Injection. Now let’s proceed towards “Immediate Solution” section to develop a running application in order to show how different action classes can be created, configured and used to perform different actions. We’ll try to create different action classes by using different techniques discussed and add the functionality of our action classes using different framework interfaces and classes as required.

Immediate Solutions

Developing a Hello Action Example

The application created here basically adds some users in the application context and the list of available users is maintained throughout the running application. We can edit and delete user information also. The directory structure of the application is similar to other standard directory structure used for earlier Struts 2 application created in Chapter 2. The directory structure of this Web application, Struts2Action, is shown in Figure 3.1.

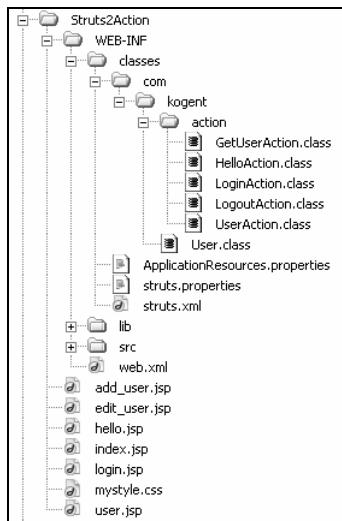


Figure 3.1: Directory structure of Struts2Action application.

You can copy web.xml, struts.properties, and ApplicationResouces.properties file from the CD provided with the book. Create a directory structure similar to Figure 3.1 with Struts2Action folder in webapp folder of your Tomcat installation. Let's create a home page in our application, index.jsp. This page simply gives you some hyperlinks to run different JSP pages and actions. Here's the code, given in Listing 3.13, for index.jsp (you can find index.jsp file in Code\Chapter 3\Struts2Action folder in CD):

Listing 3.13: index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" /></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
```

```
</head>
<body>
    <center>
        <h2>
            Struts 2 Actions
        </h2>
        <br>
        <br>
        Welcome
        <s:property value="#session.user" default="Guest" />!
        <s:if test="#session.user!=null">
            <s:url id="logout" action="logout" />
            | <s:a href="#">Logout</s:a> |
        </s:if>
        <br>
        <table cellspacing="5" width="180">
            <tr bgcolor="#f0edd9" height="25" align="center">
                <td>
                    <s:url id="hello" action="hello" />
                    <s:a href="#">Hello Action</s:a>
                </td>
            </tr>
            <tr bgcolor="#f0edd9" height="25" align="center">
                <td>
                    <s:a href="#">Add User</s:a>
                </td>
            </tr>
            <tr bgcolor="#f0edd9" height="25" align="center">
                <td>
                    <s:a href="#">View Users</s:a>
                </td>
            </tr>
            <tr bgcolor="#f0edd9" height="25" align="center">
                <td>
                    <s:a href="#">Login</s:a>
                </td>
            </tr>
        </table>
    </center>
</body>
</html>
```

The hyperlinks provided here are for different JSP pages which are yet to be created. To make these hyperlinks work, we'll create different JSP pages, action classes, and other helping Java classes in the following sections.

Creating HelloAction Class

Let's design some action classes for this application. The first action class to be created here is `HelloAction`. This action class is created implementing the `com.opensymphony.xwork2.Action` interface. The `HelloAction` class implements the `execute()` method, which sets a simple string message field with the value `Hello From Struts!`. The action can use string constants from `Action` interface, like `SUCCESS`, `ERROR` etc.

Here's the code, given in Listing 3.14, for HelloAction class (you can find HelloAction.java file in Code\Chapter3\Struts2Action\WEB-INF\src\com\kogent\action folder in CD):

Listing 3.14: HelloAction.java

```
package com.kogent.action;

import com.opensymphony.xwork2.Action;

public class HelloAction implements Action {

    String message;

    public String getMessage() {
        return message;
    }

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

    public String execute() throws Exception {
        setMessage("Hello From Struts!");
        return SUCCESS;
    }
}
```

Compile and place HelloAction.class file in WEB-INF/classes/com/kogent/action folder, as shown in the directory structure of Figure 3.1. All the properties defined in an action are pushed into its Value Stack. The value of these properties can be accessed in the next JSP page to be shown as a consequence of string returned from this action class. The action class must have getter methods for all properties to access their values, e.g. getMessage() for message.

Configuring HelloAction

All action classes need to be configured in Struts configuration file struts.xml. This can be provided by adding action mapping for the corresponding action class. Here's the syntax, given in Listing 3.15, to create your copy of struts.xml (you can find struts.xml file in Code\Chapter3\Struts2Action\WEB-INF\classes\ folder in CD):

Listing 3.15: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>

    <package name="my-default" extends="struts-default">

        <action name="hello" class="com.kogent.action.HelloAction" >
            <result name="success">hello.jsp</result>
    
```

```
</action>

</package>
</struts>
```

The action mapping for the `HelloAction` action class is provided in it. We can configure the list of Interceptors and results for a particular action. The request path having `hello.action` will invoke the `HelloAction` action and execute its `execute()` method.

A success returned from the `HelloAction` action will consequently show you `hello.jsp` page. The `org.apache.struts2.dispatcher.mapper.ActionMapper` provides a mapping between HTTP requests and action invocation requests. This returns an object `org.apache.struts2.dispatcher.mapper.ActionMapping`, if an action invocation request matches, otherwise it returns null. The object `ActionMapping` class holds action mapping information used to invoke a Struts action.

Here's the code, given in Listing 3.16, for `hello.jsp` (you can find `hello.jsp` file in `Code\Chapter 3\Struts2Action` folder in CD):

Listing 3.16: `hello.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" /></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <center>
            <h2>
                <s:property value="message" default="Struts Actions" />
            </h2>

            welcome <s:property value="#session.user" default="Guest" />!
            <br><br>
            <s:if test="#session.user!=null">
                <s:url id="logout" action="logout" />
                | <s:a href="#">Logout</s:a> |
            </s:if>
            <br><br>| <s:a href="index.jsp">Back</s:a> |
        </center>
    </body>
</html>
```

This JSP simply displays a text message from the Value Stack available.

The `<s:property value="message" default="Struts action" />` tag will invoke the `getMessage()` method and obtain message set by `HelloAction` action. If not set, the default message is printed. Another `<s:property/>` tag displays the value of `user` attribute from the session. If there is no `user` attribute in session scope, the default 'Guest' string is displayed. This session attribute handling is discussed later in the application.

Now let's run the application to see the output of our `index.jsp` and `hello.jsp` page. Start your server and use the URL `http://localhost:8080/Struts2Action/` to access this application. The first page which appears is the `index.jsp`, as shown in Figure 3.2.



Figure 3.2: The index.jsp showing different hyperlinks.

In Listing 3.13, observe the code for creating first hyperlink 'Hello Action':

```
<s:url id="hello" action="hello" />
<s:a href="#">Hello Action</s:a>
```

The hyperlink 'Hello Action' will create a request for `hello.action` and invoke the action configured with the name 'hello'. This will cause you to see the `hello.jsp` page, which has been configured as a result for this action mapping. The output of `hello.jsp` page is shown in Figure 3.3.



Figure 3.3: The hello.jsp showing message set by HelloAction action.

Using Action as ActionForm

Struts 2 removes the requirement of creating a separate ActionForm class to be populated with the input data automatically. We can use the action fields as input properties. This requires that all input properties have a corresponding setter and getter methods following the naming conventions similar to simple JavaBeans. For example, for a request parameter named ‘username’, we can declare a field ‘username’ of String type in the action class with two methods void setUsername(String) and String getUsername(). The action class field will automatically be populated with the data sent as the request parameter. This becomes possible only when the ParameterInterceptor is configured as one of the Interceptors for the action.

Creating UserAction

Let's design some JSPs and an action class to show how it works. We are creating a two JSP pages—add_user.jsp and user.jsp. The action class created here is UserAction.

Here's the code, given Listing 3.17 for UserAction action class (you can find the UserAction.java file in Code\Chapter 3\Struts2Action\WEB-INF\src\com\kogent\action folder in CD):

Listing 3.17: UserAction.java

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;

import org.apache.struts2.interceptor.ApplicationAware;

import com.kogent.User;
import com.opensymphony.xwork2.ActionSupport;

public class UserAction extends ActionSupport implements ApplicationAware{

    String username;
    String password;
    String city;
    String email;
    String type;

    Map application;

    public void setApplication(Map application) {
        this.application=application;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
}
```

```
public void setPassword(String password) {
    this.password = password;
}
public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getType() {
    return type;
}
public void setType(String type) {
    this.type = type;
}
public String execute() throws Exception {
    ArrayList users=(ArrayList)application.get("users");
    if(users==null){
        users=new ArrayList();
    }
    if(getUser(username)==null){
        users.add(buildUser());
        application.put("users", users);
    }else{
        this.addActionError("User Name is in use.");
        return ERROR;
    }
    return SUCCESS;
}

public User buildUser(){
    User user=new User();
    user.setUsername(username);
    user.setPassword(password);
    user.setCity(city);
    user.setEmail(email);
    user.setType(type);
    return user;
}
public User getUser(String username){
    User user=new User();
    boolean found=false;
    ArrayList users=(ArrayList)application.get("users");
    if(users!=null){
        Iterator it=users.iterator();
        while(it.hasNext()){
            user=(User)it.next();
            if(username.equals(user.getUsername())){
                found=true;
                break;
            }
        }
    }
    return user;
}
```

```
        }
    }
    if(found){
        return user;
    }
}
return null;
}
public void validate() {
    if ( (username == null) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
    }
    if ( (password == null) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
    }
    if ( (email == null) || (email.length() == 0) ) {
        this.addFieldError("email", getText("app.email.blank"));
    }
}
}
```

This action class extends `com.opensymphony.xwork2.ActionSupport` class and implements `org.apache.struts2.interceptor.ApplicationAware` interface. The `ActionSupport` class itself implements different interfaces, like `Action`, `LocaleProvider`, `TextProvider`, `Validateable`, `ValidationAware`, etc. and provides default implementations for the methods from these interfaces, which we can use in our action class.

The extending of `ActionSupport` class enables you to use methods, like `addFieldError()`, `addActionError()`, `getText()`, etc. The implementation of `ApplicationAware` interface brings an Application Map object into role, which can be managed in the action to store attributes in application scope. The objects stored in Application Map is available in the whole application.

In `UserAction` action class, the `execute()` method has the logic to add a new user into an `ArrayList`, which is maintained in application scope. The `getUser()` method searches for the user with the given username and returns an object of `User` class. This class is a simple JavaBean which is used to group single user information. The `ArrayList` ‘users’ in application scope basically contains the objects of `User` class.

Here’s the code, given in Listing 3.18 for `User` class (you can find `User.java` file in `Code\Chapter 3\Struts2Action\WEB-INF\src\com\kogent` folder in CD):

Listing 3.18: User.java

```
package com.kogent;

public class User {

    String username;
    String password;
    String city;
    String email;
    String type;

    public String getUsername() {
        return username;
    }
```

```

    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}

```

Another method of UserAction action class is buildUser() which returns an object of User class to be added into the ArrayList after populating it with the values of corresponding input properties. The execute() method makes sure that no two User objects with the same username are added into the application scoped ArrayList.

Now add the following action mapping, shown in Listing 3.19, for this new action class in your struts.xml file.

Listing 3.19: struts.xml with action mapping for UserAction action.

```

<struts>
    <include file="struts-default.xml"/>
    <package name="my-default" extends="struts-default">
        <action name="hello" . . . .>
            . . .
        </action>
        <action name="adduser" class="com.kogent.action.UserAction" >
            <result name="input">add_user.jsp</result>
            <result name="error">add_user.jsp</result>
            <result name="success">user.jsp</result>
        </action>
    </package>
</struts>

```

The two JSP files configured as input, success, and error results are created now. The add_user.jsp page provides a form with five input fields having names matching with the input properties defined in UserAction class.

Here's the code, given in Listing 3.20, for add_user.jsp page (you can find add_user.jsp file in Code\Chapter 3\Struts2Action folder in CD):

Listing 3.20: add_user.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" />
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <center>
            Adding User!
            <br><br>
            <table bgcolor="#f0edd9">
                <tr><td>
                    <s:actionerror />
                    <s:form action="adduser">
                        <s:textfield name="username" label="User Name" />
                        <s:password name="password" label="Password" size="15" />
                        <s:textfield name="city" label="City" />
                        <s:textfield name="email" label="E-Mail" />
                        <s:select label="Type" name="type" headerKey="---"
                                headerValue="Select Type"
                                list="#{'Admin':'Admin', 'Client':'Client'}" />
                        <s:submit value="Add User" />
                    </s:form>
                </td></tr>
            </table>
            <br>| <s:a href="index.jsp">Back</s:a> |
        </center>
    </body>
</html>
```

The action attribute of <s:form/> used in this JSP page has the value adduser and will use UserAction for the processing of data entered into this form. We can click on “Add User” hyperlink, created in index.jsp and shown in Figure 3.2, to get the add_user.jsp page. The output of add_user.jsp page is shown in Figure 3.4.



Figure 3.4: The add_user.jsp page showing form to add new user.

Enter some data into the fields and click on ‘Add User’ button to add a new user. If the username entered is unique, a new User object is added into an ArrayList, which is added into application scope. The successful addition of new user brings user.jsp page that lists all the users added.

Here’s the code, given in Listing 3.21, for user.jsp page (you can find user.jsp file in Code\Chapter 3\Struts2Action folder in CD):

Listing 3.21: user.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<s:set name="users" value="#application.users"/>
<table cellspacing="5" cellpadding="2" width="600">
    <tr bgcolor="#f0edd9">
        <td>User Name</td>
        <td>Password</td>
        <td>City</td>
        <td>Email</td>
        <td>Type</td>
        <td colspan="2" style="text-align: center;">&ampnbsp&ampnbsp

```

```
<td><s:property value="type"/></td>
<td>
    <s:url id="url" action="getuser">
        <s:param name="username"><s:property value="username"/></s:param>
    </s:url>
    <s:a href="#">Edit</s:a>
</td>
<td>
    <s:url id="url" action="delete">
        <s:param name="username"><s:property value="username"/></s:param>
    </s:url>
    <s:a href="#">Delete</s:a>
</td>
</tr>
</s:iterator>
</table>
<br>| <s:a href="#">Home</s:a> | 
<s:a href="#">Add More</s:a> |
</body>
</html>
```

Listing 3.21 uses the `<s:iterator/>` tag to iterate over the `ArrayList` and displays values of all fields set for each user. The output of `user.jsp` page is shown in Figure 3.5.

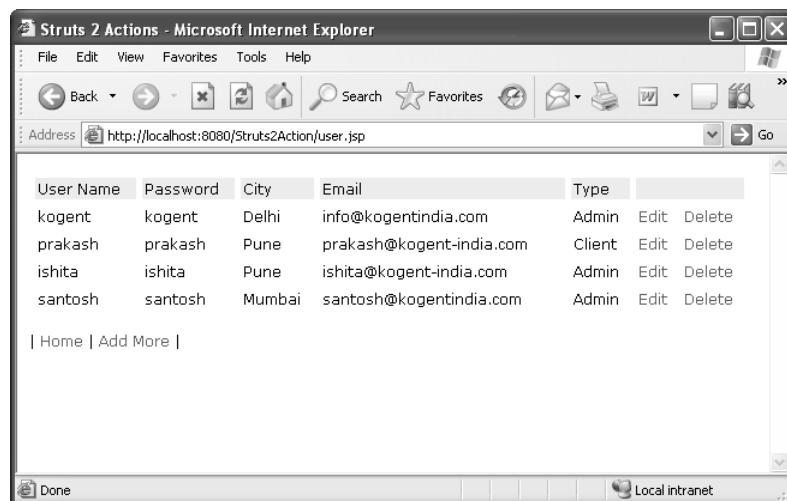


Figure 3.5: The user.jsp showing a list of users added.

You can see two hyperlinks rendered in front of each user record, i.e. 'Edit' and 'Delete'. These hyperlinks are not functional yet. It needs creation of some other action class and JPS pages to make them work.

*Creating **GetUserAction***

This action class implements `Action`, `ServletRequestAware` and `ApplicationAware` interfaces. The only interface which needs discussion here is the `ServletRequestAware` interface, which exposes

the `setServletRequest(HttpServletRequest)` method and sets the `HttpServletRequest` object to be used in the action.

This `execute()` method of this action just obtains an `User` object having `username` matching with the `username` passed as the request parameter to this action. The `HttpServletRequest` object is used to get this `username` parameter and to save the obtained `User` object into the request scope so that it is available on the coming JSP page.

Here's the code, given in Listing 3.22, for `GetUserAction` action class (you can find `GetUserAction.java` file in `Code\Chapter 3\Struts2Action\WEB-INF\src\com\kogent\action` folder in CD):

Listing 3.22: `GetUserAction.java`

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import com.opensymphony.xwork2.Action;

import org.apache.struts2.interceptor.ApplicationAware;
import org.apache.struts2.interceptor.ServletRequestAware;
import com.kogent.User;

public class GetUserAction implements
    Action, ServletRequestAware, ApplicationAware {
    HttpServletRequest request;
    Map application;
    public void setServletRequest(HttpServletRequest request) {
        this.request = request;
    }

    public void setApplication(Map application) {
        this.application = application;
    }
    public String execute() throws Exception {
        String username=request.getParameter("username");
        ArrayList users=(ArrayList)application.get("users");

        User user;
        if(users!=null){
            Iterator it=users.iterator();
            while(it.hasNext()){
                user=(User)it.next();
                if(username.equals(user.getUsername())){
                    request.setAttribute("user", user);
                    break;
                }
            }
        }
        return SUCCESS;
    }
}
```

Configure this action class by adding new action mapping into your struts.xml file, similar to earlier action mappings provided. The new action mapping is shown here:

```
<action name="getuser" class="com.kogent.action.GetUserAction" >
    <result name="success">edit_user.jsp</result>
</action>
```

Now observe the hyperlink, ‘Edit’, shown in Figure 3.5 and the code for creating this hyperlink from Listing 3.21 here:

```
<s:url id="url" action="getuser">
    <s:param name="username"><s:property value="username"/></s:param>
</s:url>
<s:a href="#">Edit</s:a>
```

The ‘Edit’ hyperlink invokes GetUserAction class and passes a request parameter named username having different values for different user records. The execution of GetUserAction gives edit_user.jsp.

Here’s the code, given in Listing 3.23, for edit_user.jsp (you can find edit_user.jsp file in Code\Chapter 3\Struts2Action folder in CD):

Listing 3.23: edit_user.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<center>
Editing User!<br><br>
<table bgcolor="#f0e0d9">
<tr>
<td>
<s:actionerror/>
<s:form action="edit">
    <s:textfield name="username" label="User Name" readonly="true">
        <s:param name="value">
            <s:property value="#request.user.username"/>
        </s:param>
    </s:textfield>
    <s:textfield name="password" label="Password" size="15">
        <s:param name="value">
            <s:property value="#request.user.password"/>
        </s:param>
    </s:textfield>

    <s:textfield name="city" label="City">
        <s:param name="value">
            <s:property value="#request.user.city"/>
        </s:param>
    </s:textfield>

```

```

        </s:param>
    </s:textfield>

    <s:textfield name="email" label="E-Mail">
        <s:param name="value">
            <s:property value="#request.user.email"/>
        </s:param>
    </s:textfield>

    <s:select label="Type"
              name="type"
              headerKey="--" headerValue="Select Type"
              list="#{'Admin':'Admin', 'Client':'Client'}">
        <s:param name="value">
            <s:property value="#request.user.type"/>
        </s:param>
    </s:select>
    <s:submit value="Edit"/>
</s:form>
</td></tr>
</table>
<br>| <s:a href="index.jsp">Home</s:a> |
</center>
</body>
</html>

```

This JSP page creates a form similar to what is created in `add_user.jsp`. But the fields this time are populated with the values of different field of the `User` object, which is obtained here from request. This `User` object is set into request scope by the `GetUserAction`, according to the username passed as the request parameter when the 'Edit' hyperlink is clicked. The output of `edit_user.jsp` page is shown in Figure 3.6.

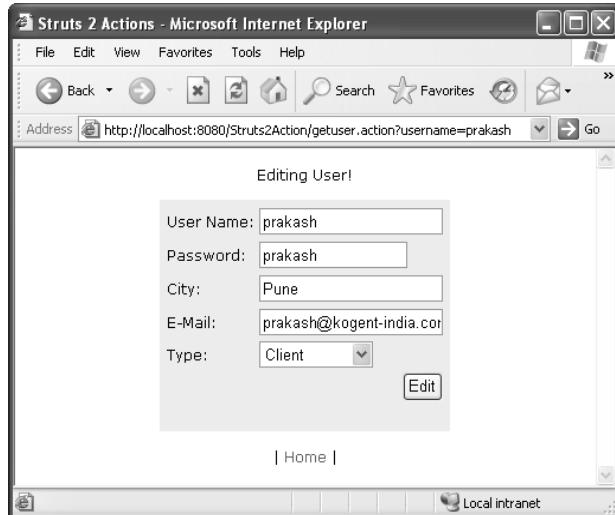


Figure 3.6: The `edit_user.jsp` page showing fields filled with current values.

Configuring Action Class Method

To edit the user record, shown in Figure 3.6, we need to click over the ‘Edit’ button. But, before that, we have to decide which action is going to handle this form. The form is similar to what is prepared for adding a new user; hence we need the same set of input properties to be set as that of `UserAction` action class. Similarly, the validation logic for these fields will be same as what is implemented in the `validate()` method of `UserAction`. So, when all these things are similar and can be reused, there is no need for creating a new action class here.

Struts 2 allows configuration of any method having signature `String methodName()` to be configured and invoked, which can work similar to the `execute()` method of action class.

Add a new method into your existing `UserAction` class. This method contains the logic to edit the users. The logic is to replace the existing `User` object in the `ArrayList` with the new `User` object created with new data.

Here’s the code, given in Listing 3.24, for `edit()` method:

Listing 3.24: `edit()` method of `UserAction.java`

```
public class UserAction extends ActionSupport implements ApplicationAware{

    String username;
    String password;
    String city;
    String email;
    String type;

    Map application;

    public void setApplication(Map application) {
        this.application=application;
    }

    //Setter and getter methods..
    . . .

    public String execute() throws Exception {
        . . .
        return SUCCESS;
    }

    //New method added.
    public String edit() throws Exception{
        ArrayList users=(ArrayList)application.get("users");
        User user=null;
        int index=0;
        Iterator it=users.iterator();
        while(it.hasNext()){
            user=(User)it.next();
            if(user.getUsername().equals(username)){
                break;
            }
            index++;
        }
    }
}
```

```

        User newuser=buildUser();
        users.set(index, newuser);
        application.put("users", users);
        return SUCCESS;
    }
    public User buildUser(){
        . . .
    }
    public User getUser(String username){
        . . .
        . . .
    }
    public void validate() {
        . . .
        . . .
    }
}

```

Recompile the `UserAction` action class and get the newly compiled class file in `WEB-INF\classes\com\kogent\action` folder. Look at the `action` attribute of the `<ss:form/>` tag used in `edit_user.jsp` page, which is set with the value ‘edit’. Add a new action mapping with the name ‘edit’, which invokes this new `edit()` method of `UserAciton` class having the logic to edit the user information. This is done by providing a method parameter in the action mapping as shown here:

```

<action name="edit" class="com.kogent.action.UserAction" method="edit" >
    <result name="input">edit_user.jsp</result>
    <result name="success">user.jsp</result>
</action>

```

Now, we can click on the ‘Edit’ button, shown in Figure 3.6, to use the `edit()` method of `UserAction` class. The `edit()` method updates the `ArrayList` and return from this method consequently shows you `user.jsp` page displaying updated data. Similarly, we can add one more method to delete the user information. The logic embedded in this method is just to remove the matching `User` object from the `ArrayList`.

Here’s the new method `deleteUser()`, given in Listing 3.25, added into `UserAction.java`. Recompile the action after adding the new method.

Listing 3.25: `deleteUser()` method of `UserAction.java`

```

public String deleteUser() throws Exception{
    ArrayList users=(ArrayList)application.get("users");
    User user=null;
    int index=0;
    Iterator it=users.iterator();
    while(it.hasNext()){
        user=(User)it.next();
        if(user.getUsername().equals(username)){
            break;
        }
        index++;
    }
}

```

```
        users.remove(index);
        application.put("users", users);
        return SUCCESS;
    }
```

Consider the ‘Delete’ hyperlink shown in Figure 3.5, and observe the following code which creates this hyperlink:

```
<s:url id="url" action="delete">
    <s:param name="username"><s:property value="username"/></s:param>
</s:url>
<s:a href="#">

---


```

Now provide a new action mapping to match with this action request, i.e. delete.action. The new action mapping to be added in struts.xml is shown here:

```
<action name="delete" class="com.kogent.action.UserAction" method="deleteUser" >
    <interceptor-ref name="basicStack"/>
    <result name="input">edit_user.jsp</result>
    <result name="success">user.jsp</result>
</action>
```

The mapping provided here will invoke the deleteUser() method of the UserAciton action class, instead of executing its execute() method, as the method configured here is deleteUser. However, we do not want to execute the validate() method, as this time only the username is being passed as request parameter to the action. The validate() method of UserAciton would have certainly been executed if we would have used defaultStack as Interceptor stack here (all actions configured here uses the default Interceptor stack defaultStack).

The reason behind this concept is that the defaultStack Interceptor stack includes DefaultWorkflowInterceptor Interceptor, which basicStack does not. This Interceptor is responsible for invoking the validate() method before the real business logic method is invoked.

Now, we are ready to delete the existing users from the ArrayList. See the ‘Edit’ and ‘Delete’ options provided in Figure 3.5.

Creating LoginAction with ApplicationAware and SessionAware Interface

We already have created some actions in this application, which implements ApplicationAware and ServletRequestAware interfaces. Here, once again, we are creating a new action class which implements both ApplicationAware and SessionAware interface. The implementation of SessionAware interface enables this action to handle a Session Map which stores objects in session scope. In this way, we become capable of handling session and session attributes.

We are creating a class to authenticate the user for its username and password. The success result will be rendered only when such user exists. This sets the username as the session attribute after finding the valid login attempt.

Here's the code, given in Listing 3.26 for LoginAction action class (you can find LoginAcion.java file in Code\Chapter 3\Struts2Action\WEB-INF\src\com\kogent\action folder in CD):

Listing 3.26: LoginAcion.java

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ApplicationAware;
import org.apache.struts2.interceptor.SessionAware;

import com.kogent.User;
public class LoginAction extends ActionSupport implements
    ApplicationAware, SessionAware{

    String username;
    String password;

    Map session;
    Map application;

    public void setSession(Map session) {
        this.session=session;
    }
    public void setApplication(Map application) {
        this.application=application;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String execute() throws Exception {
        boolean found=false;
        ArrayList users=(ArrayList)application.get("users");
        if(users!=null){
            Iterator it=users.iterator();
            while(it.hasNext()){
                User user=(User)it.next();
                if(username.equals(user.getUsername()) &&
                    password.equals(user.getPassword())){
                    found=true;
                    session.put("user", username);
                    break;
                }
            }
        }
        return "success";
    }
}
```

```
        }
    }
    if(found){
        return SUCCESS;
    }
}
this.addActionError("Invalid User Name or Password");
return ERROR;
}
}
```

Let's create a JSP page to provide input to this action class. This JSP page is a login page providing two input fields for username and password.

Here's the code, given in Listing 3.27, for login.jsp page (you can find login.jsp file in Code\Chapter 3\Struts2Action folder in CD):

Listing 3.27: login.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" />
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <center>
            Enter User Name and Password.<br><br>
            <table bgcolor="#f0edd9">
                <tr>
                    <td>
                        <s:actionerror />
                        <s:form action="login">
                            <s:textfield name="username" label="User Name"/>
                            <s:password name="password" label="Password" />
                            <s:submit value="Login"/>
                        </s:form>
                    </td>
                </tr>
            </table>
            <br> | <s:a href="index.jsp">Home</s:a> |
        </center>
    </body>
</html>
```

Click on the 'Login' hyperlink, shown in Figure 3.2, and created in Listing 3.13. This consequently shows you login.jsp page. The page looks like Figure 3.7.



Figure 3.7: The login.jsp page showing login form.

Add the following action mapping in the struts.xml file matching the action requested from this login form created here:

```
<action name="login" class="com.kogent.action.LoginAction" >
    <result name="input">login.jsp</result>
    <result name="error">login.jsp</result>
    <result name="success">hello.jsp</result>
</action>
```

Now we can login by providing a valid set of username and password and clicking on the ‘Login’ button shown in Figure 3.7. The successful login brings you hello.jsp. However, the output of hello.jsp this time (Figure 3.8) is different from what is shown in Figure 3.3.



Figure 3.8: The hello.jsp showing username from session and a Logout option.

Similarly, we can see our index.jsp page after getting logged in. The welcome message on that page also changes. See the output of index.jsp shown in Figure 3.9.



Figure 3.9: The index.jsp page with different welcome message and Logout option.

Now the only option left here to be made functional is the Logout option. This again is achieved by creating a new action class.

Creating **LogoutAction**—a POJO Action

This action class is unique in the sense that this class does not extend any class and does not even implement any interface. But it has access to HttpSession object using ActionContext object. This class simply removes the session attribute which stores the username of the currently logged in user.

Here's the code, given in Listing 3.28, for LogoutAction action class (you can find LogoutAction.java file in Code\Chapter 3\Struts2Action\WEB-INF\src\com\kogent\action folder in CD):

Listing 3.28: LogoutAction.java

```
package com.kogent.action;

import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionContext;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class LogoutAction {

    public String execute() throws Exception {
        ActionContext ctx = ActionContext.getContext();
        HttpServletRequest request = (HttpServletRequest) ctx
```

```
        .get(ServletActionContext.HTTP_REQUEST);
HttpSession session = request.getSession();
session.removeAttribute("user");

        return "success";
}

}
```

Add a new action mapping to invoke this action. The action mapping added in `struts.xml` file for `LogoutAction` action class is shown here:

```
<action name="logout" class="com.kogent.action.LogoutAction" >
    <result name="success">login.jsp</result>
</action>
```

Now we are ready to logout by clicking over the ‘Logout’ hyperlink provided in `index.jsp` and `hello.jsp` (see Figures 3.8 and 3.9).

With this we complete our discussion on the different techniques introduced to create an action class and configure it with other components, like Interceptors and results. We also dealt with the use of `Action` interface and `ActionSupport` class in creating Struts 2 actions along with its full implementation. The Struts 2 support for POJOs as actions has also been described with all new flexibility added by this approach. There are lots of other flexibilities provided by Struts 2 Framework, while designing action classes like the structure of its `execute()` methods. The concept of Inversion of Control has further simplified the structure of Struts 2 action classes. All the framework interfaces and classes are discussed and implemented in running application, which provides a clear idea about the capability and flexibility introduced by Struts 2 action classes.

The next chapter will be discussing Interceptors. The discussion in the coming chapter includes all the available Struts 2 Interceptors along with the functionalities provided by them. We will also describe the configuration of different Interceptors for actions.



4

Implementing Interceptors in Struts 2

If you need an immediate solution to:

Implementing Interceptors

See page:

131

In Depth

Interceptors are one of the most powerful features of Struts 2. The introduction of Interceptors into Struts 2 Framework set it apart from other frameworks. An Interceptor, as the name suggests, intercepts the requests, and provides some additional processing before and after the execution of action and result. Common functionalities required by each action are implemented as Interceptors. These functionalities may include validation of input data, pre-processing of file upload, protection from double submit, and sometimes pre-population of controls with some data before the web page appears, etc. Some common functionalities of a Web application and advance features have been implemented in the form of Interceptors. The Interceptor approach helps in modularizing common code into reusable classes.

The framework provides a set of Interceptors, which can be used to provide the required functionalities to an action. The Interceptors, including our custom Interceptors, can be implemented in a particular order to get the functionalities required for each action. The Interceptor or a stack of Interceptors is configured for an action, which is executed before the action is executed, to provide all pre-processing of the request. Similarly, these configured Interceptors are again executed after the execution of action to provide addition processing, if any.

This chapter discusses basic concept of Interceptors, their configuration and implementation in the Struts 2 Framework. The Interceptor Stack and the order of execution of Interceptor classes in the stack are also described here. All framework Interceptors are defined and described with example.

Understanding Interceptors

The Struts 2 Framework uses the concept of Interceptors to share the solutions for some common concerns by different actions. The framework invokes a particular Action object on the submission of a request for it. But before the execution of Action, the invocation is intercepted by some other object to provide additional processing required. Similarly, after the execution of Action, the invocation can be intercepted again. This intercepting object is known as Interceptor. An Interceptor is a class, which contains logic implementation to provide a specific functionality.

Every Interceptor is pluggable and the required Interceptor or Interceptor stack can be configured on a per-action basis. The separation of core functionality code in the form of Interceptors makes Action more lightweight, as it need not carry additional burden of implementing extra code and references in it. The purpose of using Interceptors is to allow greater control over controller layer and separate some common logic that applies to multiple actions. Figure 4.1 shows the request life-cycle in Struts 2 Framework, which represents how a request is intercepted by an Interceptor or a stack of Interceptors.

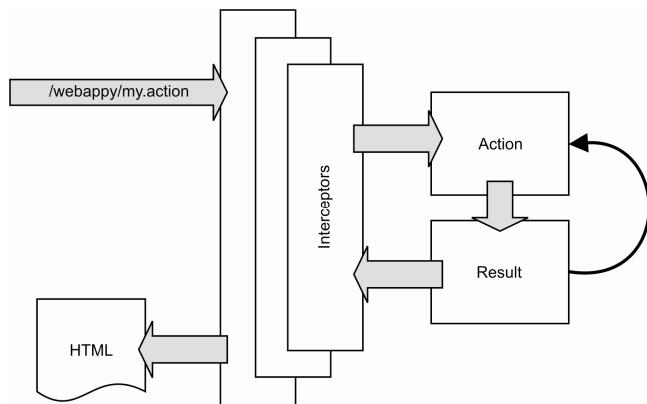


Figure 4.1: Request life-cycle

Observe that in Figure 4.1 the Action class is executed, then a request passes through the set of Interceptors (Interceptor1, Interceptor2, . .) which provide request pre-processing which is required before the action is executed. Similarly, after the execution of Action class, all Interceptors are executed once more to provide post-processing, if any. But, this time, the execution order of Interceptors is reversed (. ., Interceptor2, Interceptor1).

An Interceptor class can be created by implementing `com.opensymphony.xwork2.interceptor.Interceptor` interface or by extending `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class. Every Interceptor class has a `String intercept (ActionInvocation invocation)` method that allows some processing on the request before and/or after rest of the processing of the request by `ActionInvocation` or to short-circuit the processing and just return a String return code. We have a number of Interceptor classes already created, compiled, tested, and bundled with Struts 2 distribution.

Interceptors as RequestProcessor

In Struts1, we have a `RequestProcessor` class which provides most of the request processing through its methods. For every request, an object of `RequestProcessor` is created and its `process()` method is executed. There is a different method for every step in request processing and these methods are invoked in a sequence. These `RequestProcessor` methods provide core functionalities, like finding `ActionForm` class associated with request, getting Locale for the current request, setting `contentType` for the response, and many more. We can also create our own `RequestProcessor` class by extending the default `RequestProcessor` class. The custom `RequestProcessor` class can be used to execute some business logic at any point during the request-processing phase. For example, we can create a custom `RequestProcessor` class to find whether the user is logged in and has role to execute a particular action before executing the request.

In Struts 2, the Interceptors have replaced `RequestProcessor`. Similar to `RequestProcessor`, Interceptors provide all basic pre-processing required before executing Action. Like `RequestProcessor` class, we now have Interceptors to contain common logic to be applied to a number of actions. In most of the Struts 2 applications, we do not need to create our custom Interceptors as all the required pre-processing is provided by the default Interceptor stack (an Interceptor stack is a set of Interceptors configured in a specific order) which is declared in the `struts-default.xml` file. Therefore, the default Interceptor stack can be taken as `RequestProcessor` in Struts 2. Only a small percentage of Struts 2 applications need to define their custom Interceptors for specific functionality.

Configuring Interceptors

The Interceptors to be used in the application must be declared in `struts.xml` file. The Interceptor classes can be defined using a name-class pair specified in Struts configuration file. The other approach is to include another `.xml` file containing the Interceptors configuration into our `struts.xml` file. All the Interceptors, which are required to preprocess the request for a given action, should be defined in the action mapping provided for that specific action.

For example, let's assume that we have two custom Interceptors classes like `Interceptor1_class_name.class` and `Interceptor2_class_name.class`, which are to be used. These two Interceptor classes are defined in `struts.xml` file using `<interceptor>` element as shown in Listing 4.1:

Listing 4.1: Sample Interceptor configuration provided in `struts.xml`

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>

<include file="struts-default.xml"/>
<package name="default" extends="struts-default">

<interceptors>
    <interceptor name="interceptor1" class="Interceptor1_class_name"/>
    <interceptor name="interceptor2" class="Interceptor2_class_name " />
</interceptors>

<action name="login" class="LoginAction">
    <interceptor-ref name="interceptor1"/>
    <interceptor-ref name="interceptor2"/>

    <result name="input">login.jsp</result>
    <result name="success">success.jsp</result>
</action>
</package>
</struts>

```

Every Interceptor class is given a name, which is used to refer that Interceptor. Further, the list of Interceptor classes used to intercept the action request is declared in action mapping using `<interceptor-ref name=". . .">` elements. The order of Interceptors is important here, as they are executed in the same order in which they are declared in action mapping.

For the request for `LoginAction` action, the two Interceptors declared are `Interceptor1` and `Interceptor2`. Before the execution of action class the corresponding Interceptor classes will be executed to provide all pre-processing.

Interceptor Stacks

For different applications and for different actions in an application, we may need to apply a same set of Interceptors repeatedly. Therefore, instead of writing the whole list of Interceptors in every action mapping repeatedly, we can group these set of Interceptors as an Interceptor stack and can use stack

name in the action mapping. In Listing 4.2, we have defined two Interceptors and an Interceptor stack (`custom_stack`) which includes these two Interceptors. The Interceptor stack is defined using `<interceptor-stack>` element. The action mapping provided in Listing 4.2 uses a single `<interceptor-ref>` element to declare Interceptor stack to be used, instead of giving two `<interceptor-ref>` elements for each Interceptor class, as shown in Listing 4.1:

Listing 4.2: Sample Interceptor Stack defined in struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <interceptors>
            <interceptor name="interceptor1" class="Interceptor1_class_name"/>
            <interceptor name="interceptor2" class="Interceptor2_class_name " />

            <interceptor-stack name="custom_stack">
                <interceptor-ref name="interceptor1"/>
                <interceptor-ref name="interceptor2"/>
            </interceptor-stack>
        </interceptors>

        <action name="login" class="LoginAction">
            <interceptor-ref name="custom_stack"/>
            <result name="input">login.jsp</result>
            <result name="success">success.jsp</result>
        </action>
    </package>
</struts>
```

The name attribute of `<interceptor-ref>` can take the name of an Interceptor or an Interceptor stack.

Default Interceptor Configuration—struts-default.xml

We have developed Struts 2 applications in previous chapters, but most of the applications do not define any Interceptor or Interceptor stack in its Struts configuration file. All framework Interceptors are defined in `struts-default.xml`. In addition to defining all framework supported results, the `struts-default.xml` file contains the definitions for all Interceptors and Interceptor stacks. The Interceptor and Interceptor stack definitions can be extended to your application from `struts-default.xml`. To use the Interceptors bundled with framework, we can include `struts-default.xml` into our `struts.xml` file and extend our package from the `struts-default` package.

Here's one more sample `struts.xml` file, given in Listing 4.3, for a Struts 2 application:

Listing 4.3: Sample struts.xml with three action mappings

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
```

```

<include file="struts-default.xml"/>
<package name="mypackage" extends="struts-default">

    <action name="action1" class="someAction1">
        <interceptor-ref name="params"/>
        <interceptor-ref name="servlet-config"/>
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="workflow"/>

        <result name="success">success.jsp</result>
        <result name="error">error.jsp</result>
    </action>

    <action name="action2" class="someAction2">
        <interceptor-ref name="completeStack"/>

        <result name="success">success.jsp</result>
        <result name="error">error.jsp</result>
    </action>
    <action name="action3" class="someAction3">
        <result name="success">success.jsp</result>
        <result name="error">error.jsp</result>
    </action>

</package>
</struts>

```

The three action mappings are provided in the Listing 4.3. All three action mappings are grouped in a package named `mypackage`. The different set of Interceptors are defined for two action mappings and the last action mapping do not have any `<interceptor-ref>` element defining Interceptor for it.

The four Interceptors are defined for action `action1`. Now, you might ask, “Where are the definitions for these Interceptors and what classes will be executed?” The answer is in the `struts-default.xml` file and shown in Listing 4.4. The `struts-default.xml` file, which is contained in `struts2-core-2.0.6.jar` available with Struts 2 distribution, defines all framework Interceptors. To use framework Interceptors, like `action1`, make sure of the following things:

- `struts-default.xml` is included in your `struts.xml` file (see `<include>` element in Listing 4.3).
- action mappings are enclosed in a `<package>` element, which extends `struts-default` package (see `<package>` element in Listing 4.3).

Here's Listing 4.4 for some Interceptor definitions provided in `struts-default.xml` file:

Listing 4.4: Interceptors configured in struts-default.xml

```

<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="struts-default">
    <!--Result type declaration-->
    <result-types>
        . . .
        . . .

```

```
</result-types>

<!-- Interceptor and Interceptor Stack Declaration -->
<interceptors>
<interceptor name="alias"
class="com.opensymphony.xwork.interceptor.AliasInterceptor"/>
<interceptor name="autowiring"
class="com.opensymphony.xwork.spring.interceptor.ActionAutowiringInterceptor"/>
<interceptor name="chain"
class="com.opensymphony.xwork.interceptor.ChainingInterceptor"/>
<interceptor name="component"
class="com.opensymphony.xwork.interceptor.component.ComponentInterceptor"/>
<interceptor name="conversionError"
class="org.apache.struts2.interceptor.webwerkConversionErrorInterceptor"/>
<interceptor name="external-ref"
class="com.opensymphony.xwork.interceptor.ExternalReferencesInterceptor"/>
<interceptor name="execAndwait"
class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor"/>
<interceptor . . . . . />
<interceptor . . . . . />
. . .
. . .
<interceptor name="validation"
class="com.opensymphony.xwork.validator.ValidationInterceptor"/>
<interceptor name="workflow"
class="com.opensymphony.xwork.interceptor.DefaultWorkflowInterceptor"/>
<interceptor name="...", class="..."/>
<interceptor name="...", class="..."/>
. . .
. . .
</interceptors>
</package>
</struts>
```

These packages containing Interceptor classes are available in the Struts 2 JAR files. The Interceptor definition just maps an Interceptor class name with an alias name, which can further be used to refer that Interceptor.

Similar to defining Interceptors, different Interceptor stacks are also defined in `struts-default.xml`. These Interceptor stacks are a group of some commonly used and required Interceptors. These Interceptors stacks are defined using `<interceptor-stack name=" " >` element. An Interceptor stack can combine Interceptors as well as other Interceptor stacks. See the definition of `basicStack` and its use in other Interceptor stack definitions in Listing 4.5. The required conditions to use Interceptor stacks in our action mapping are similar to what is defined for using Interceptor definitions from `struts-default.xml` earlier. The second action mapping provided in Listing 4.3 uses `completeStack`, which is an Interceptor stack and defined in `struts-default.xml`.

Listing 4.5 defines all Interceptor stacks:

Listing 4.5: Interceptor Stacks configured in `struts-default.xml`

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
```

```
<package name="struts-default">

    <!--Result type declaration-->
    <result-types>
        . . .
        . . .
    </result-types>

    <!-- Interceptor and Interceptor Stack Declaration -->

    <interceptors>
        <interceptor . . . . . />
        <interceptor . . . . . />
        . . .
        . . .

        <!-- Basic stack -->
        <interceptor-stack name="basicStack">
            <interceptor-ref name="exception"/>
            <interceptor-ref name="servlet-config"/>
            <interceptor-ref name="prepare"/>
            <interceptor-ref name="static-params"/>
            <interceptor-ref name="params"/>
            <interceptor-ref name="conversionError"/>
        </interceptor-stack>

        <!-- Sample validation and workflow stack -->
        <interceptor-stack name="validationWorkflowStack">
            <interceptor-ref name="basicStack"/>
            <interceptor-ref name="validation"/>
            <interceptor-ref name="workflow"/>
        </interceptor-stack>

        <!-- Sample file upload stack -->
        <interceptor-stack name="fileUploadStack">
            <interceptor-ref name="fileUpload"/>
            <interceptor-ref name="basicStack"/>
        </interceptor-stack>

        <!-- Sample model-driven stack -->
        <interceptor-stack name="modelDrivenStack">
            <interceptor-ref name="model-driven"/>
            <interceptor-ref name="basicStack"/>
        </interceptor-stack>

        <!-- Sample action chaining stack -->
        <interceptor-stack name="chainStack">
            <interceptor-ref name="chain"/>
            <interceptor-ref name="basicStack"/>
        </interceptor-stack>

        <!-- Sample i18n stack -->
        <interceptor-stack name="chainStack">
            <interceptor-ref name="i18n"/>
            <interceptor-ref name="basicStack"/>
        </interceptor-stack>
```

```
<interceptor-stack name="executeAndwaitStack">
<interceptor-ref name="basicStack"/>
<interceptor-ref name="execAndwait"/>
</interceptor-stack>

<interceptor-stack name="defaultstack">
<interceptor-ref name="exception"/>
<interceptor-ref name="alias"/>
<interceptor-ref name="prepare"/>
<interceptor-ref name="servlet-config"/>
<interceptor-ref name="i18n"/>
<interceptor-ref name="chain"/>
<interceptor-ref name="model-driven"/>
<interceptor-ref name="fileUpload"/>
<interceptor-ref name="static-params"/>
<interceptor-ref name="params"/>
<interceptor-ref name="conversionError"/>
<interceptor-ref name="validation"/>
<interceptor-ref name="workflow"/>
</interceptor-stack>

<interceptor-stack name="completetestStack">
    <interceptor-ref name="defaultStack"/>
</interceptor-stack>
</interceptors>
<default-interceptor-ref name="defaultStack"/>
</package>
</struts>
```

For the last action mapping for action `action3` in Listing 4.3, there is no Interceptor or Interceptor stack defined. It does not mean that the request for this action is not going to be intercepted and pre-processed. When there is no Interceptor declared for an action mapping and our package extends the `struts-default` package, the default Interceptor stack is used. The default Interceptor stack is `defaultStack`, which is defined in `struts-default.xml` using `<default-interceptor-ref name="defaultStack"/>` element. We can override this definition by defining default Interceptor in our `struts.xml` file.

Implementing Interceptors in Struts 2

Struts 2 Framework has already defined its own set of Interceptors which can be used in the application to provide required processing before and after the action class is executed. We have a number of Interceptor classes bundled in the Struts 2 distribution and defined in `struts-default.xml` file. Each Interceptor is designed to implement logic for a specific function, which is common to almost all Web applications. The Interceptor class can be seen as a reusable component, which is commonly used in different application and different action invocations in an application. To use these Interceptors in our application, we can extend `struts-default` package or define these Interceptors in our package with name-class pair specified in `<interceptor>` element. All framework Interceptors are listed in Table 4.1 here.

Table 4.1: Struts 2 Framework Interceptors	
alias	prepare
autowiring	profiling
chain	roles
checkbox	scope
conversionError	scoped-model-driven
createSession	servlet-config
debugging	sessionAutowiring
exception	static-params
execAndWait	store
external-ref	timer
fileUpload	token
i18n	token-session
logger	validation
model-driven	Workflow
params	

Let's now discuss all the framework Interceptors in detail. We will discuss all the benefits of modularizing a large segment common to all application code into several reusable Interceptor classes with their implementation in the application. These Interceptors are discussed here for the better understanding of the basic functionality provided by them with their parameters and configuration examples.

Alias Interceptor

This Interceptor aliases a named parameter to a different parameter name. In action chaining, when two different action classes share a common parameter with a different name, this Interceptor is used to give an alias name to a parameter of first action class, which matches the parameter name in the second action class. The alias expression of the action should be in the form of `#{ 'name1': 'alias1', 'name2': 'alias2' }`. In other words, this Interceptor converts similar parameters that may be named differently between different requests. Table 4.2 lists class and different parameters for alias Interceptor.

Table 4.2: alias Interceptor

Interceptor name	Interceptor class
alias	com.opensymphony.xwork2.interceptor.AliasInterceptor
Parameters	
aliasesKey	It is an optional parameter which takes name of the action parameter that will be looked for alias map; the default value for this parameter is aliases

The class for alias Interceptor is `com.opensymphony.xwork2.interceptor.AliasInterceptor` which extends `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class. The abstract class `AbstractInterceptor` implements `com.opensymphony.xwork2.interceptor.Interceptor` interface. The basic method in all Interceptor classes is `String intercept (ActionInvocation invocation)` which is overridden to handle interception. The other method of alias Interceptor is `void setAliasesKey(String aliasesKey)`, which sets the value for field `aliasesKey` with the name of the action parameter to be looked for alias map.

Let's take an example here. Suppose, an action class has a parameter named `username` and another action has parameter `loginid`. To set parameter `loginid` with the value of parameter `username`, the alias Interceptor is applied and an action parameter, named `aliases`, is defined to describe the alias map containing name-alias pairs.

Here's Listing 4.6 showing the configuration using alias Interceptor:

Listing 4.6: Configuring alias Interceptor in struts.xml

```
<package name="default" extends="struts-default">

    <action name="action1" class="com.Action1">
        <result name="success" type="chain">action2</result>
        <result name="error"/>/error.jsp</result>
        <result name="input"/>/input.jsp</result>
    </action>

    <action name="action2" class="com.Action2">
        <!-- The value of parameter username will be applied as if it were
        named loginid -->
        <param name="aliases">#{'username': 'loginid' }</param>

        <interceptor-ref name="alias"/>
        <interceptor-ref name="basicStack"/>
        <interceptor-ref name="workflow"/>

        <result name="success"/>/success.jsp</result>
        <result name="error"/>/error.jsp</result>
        <result name="input"/>/input.jsp</result>
    </action>
</package>
```

When the `execute()` method of action `action1` returns ‘success’, another action `action2` is invoked. The parameter `loginid` in `action2` will be set with the value of parameter `username` of `action1` in this action chaining.

Chaining Interceptor

This Interceptor is used to copy all objects in the value stack of currently executing action class to the value stack of the next action class to be executed in action chaining. This copying of parameters from one action value stack to another action value stack is one of the basic requirements in action chaining where one action class needs to access parameters available in previously executing action class. The only objects which are not copied are those which implements `Unchainable`. We can also define a collection of includes and excludes to control how and which parameter is to be copied.

If the current value stack is empty, the Interceptor does nothing. Table 4.3 lists the class and different parameters for chain Interceptor.

Table 4.3: Chain Interceptor	
Interceptor name	Interceptor class
chain	com.opensymphony.xwork2.interceptor.ChainingInterceptor
Parameters	
excludes	This parameter provides a list of parameter names, which are to be excluded from copying. Remaining parameters will be included for copying
includes	This parameter provides a list of parameter names, which are to be included for copying. Remaining parameters will be excluded from copying

The class for this Interceptor `com.opensymphony.xwork2.interceptor.ChainingInterceptor`, which is similar to other Interceptor classes, extends `AbstractInterceptor` class. The various methods of this Interceptor are as follows:

- `Collection getExcludes()` – It returns a collection of excluded parameters
- `Collection getIncludes()` – It returns a collection of included parameters
- `String intercept (ActionInvocation invocation)` – It implements code to handle interception
- `void setExcludes(Collection excludes)` – It sets a collection of excluded parameters
- `void setIncludes (Collection includes)` – It sets a collection of included parameters

The action chaining feature provided by framework allows the chaining of multiple actions in a sequence, i.e. one action is executed after another. This feature works by applying chain result to one action and intercepting another action’s invocation with `ChainingInterceptor`. A chain result is a result type, which invokes an action with its own Interceptor stack and result. The request is forwarded to target action with propagation of the state of source action class.

Here’s Listing 4.7 showing the configuration using chain Interceptor:

Listing 4.7: Configuring Chain Interceptor in struts.xml

```
<action name="action1" class="com.Action1">
    <interceptor-ref name="basicStack"/>
    <result name="success" type="chain">action2</result>
</action>
<action name="action2" class="comAction2">
    <interceptor-ref name="chain"/>
    <interceptor-ref name="basicStack"/>
    <result name="success">success.jsp</result>
</action>
```

Checkbox Interceptor

The Checkbox Interceptor looks for the hidden identification fields, which are used to specify the original value of the check box. If the check box is left unchecked, the value added in the parameters for this checkbox is ‘false’.

The `org.apache.struts2.interceptor.CheckboxInterceptor` class implements the `Interceptor` interface. This Interceptor class has a field `String uncheckedValue` and a method `void setUncheckedValue(String uncheckedValue)` to set this field. Table 4.4 lists class and different parameters for Checkbox Interceptor.

Table 4.4: Checkbox Interceptor

Interceptor name	Interceptor class
checkbox	<code>org.apache.struts2.interceptor.CheckboxInterceptor</code>
Parameters	
<code>setUncheckedValue</code>	This parameter is used to override the default value of an unchecked box by setting <code>uncheckedValue</code> property

The Checkbox Interceptor is included in the default Interceptor stacks, like `basicStack`, `defaultStack`, `paramsPrepareParamsStack` defined in `struts-default.xml` file.

Conversion Error Interceptor

The Conversion Error Interceptor to be discussed here is `StrutsConversionErrorInterceptor` which extends the `ConversionErrorInterceptor` class. All errors found in `ActionContext`'s `conversionErrors` map is added by this Interceptor as a field error. This Interceptor works when the action implements `ValidationAware` interface or extends `ActionSupport` class (an abstract class implementing `ValidationAware`). For all fields, their original value is saved so that the subsequent requests return original value for the fields, instead of their values in action. For example, if a field value `somestring` cannot be converted into `int`, we'll always want to show original value `somestring`, while displaying the field error message for this field. We can get the map containing all conversion errors using the `ActionContext.getConversionErrors()` method.

This Interceptor is included in `basicStack` and `defaultStack` Interceptor stack defined in `struts-default.xml` file. Table 4.5 lists the class and different parameters for `conversionError` Interceptor.

Table 4.5: conversionError Interceptor

Interceptor name	Interceptor class
conversionError	org.apache.struts2.interceptor.StrutsConversionEr rorInterceptor
Parameters	
There is no parameter defined for this Interceptor	

Here's Listing 4.8 for the configuration using conversionError Interceptor:

Listing 4.8: Configuring conversionError Interceptor in struts.xml

```
<action name="action1" class="someAction1">
    <interceptor-ref name="params"/>
    <interceptor-ref name="conversionError"/>
    <result name="success">success.jsp</result>
</action>
```

Create Session Interceptor

This Interceptor creates an HttpSession. The class to be used for createSession Interceptor is org.apache.struts2.interceptor.CreateSessionInterceptor, which extends AbstractInterceptor Interceptor. The only method of this Interceptor is its intercept() method which is used to define what should be done when this Interceptor intercepts. There is no parameter defined for this Interceptor. Table 4.6 lists the class and different parameters for createSession Interceptor.

Table 4.6: create-session Interceptor

Interceptor name	Interceptor class
create-session	org.apache.struts2.interceptor.CreateSessionIntercep tor
Parameters	
There is no parameter defined for this Interceptor	

Here's Listing 4.9 for configuration showing the use of create-session Interceptor.

Listing 4.9: Configuring create-session Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="create-session"/>
    <interceptor-ref name="defaultStack"/>
    <result name="input">some_page</result>
    <result name="success">some_other_page</result>
</action>
```

Debugging Interceptor

This Interceptor provides different screens giving an inner look of the data behind the page. This Interceptor works when devMode is enabled in struts.properties file. A request parameter named, debug, is passed, which is removed from parameter list before the action is executed. The different values for debug parameter can be as follows:

- xml**—It dumps the parameters, context, session, and value stack as an XML document.
- console**—An OGNL Console is displayed to test OGNL expressions against the value stack.
- command**—It tests an OGNL expression and returns the string result. This can be used only with OGNL console.

Table 4.7 lists class for Debugging Interceptor.

Table 4.7: Debugging Interceptor

Interceptor name	Interceptor class
debugging	org.apache.struts2.interceptor.debugging.DebuggingInterceptor

The Interceptor class DebuggingInterceptor implements Interceptor interface and have methods, like String getParameter(String key), String intercept(ActionInvocation inv), void printContext(), and void setDevMode(String mode).

The screens provided by this Interceptor are useful when debugging a Struts 2 application. For debugging, set struts.devMode to true and use defaultStack as Interceptor stack, which includes the debugging Interceptor. A request string, like <http://localhost:8080/Test/Hello.action?debug=xml>, will display parameters, context, session and value stack in XML format.

Execute and Wait Interceptor

While running a long action, the user may get impatient in case of a long delay in response. To avoid this, the execAndWait Interceptor is used, which runs a long running action in the background and displays a page with a progress bar to the user. It also prevents the HTTP request from timing out. The class for this Interceptor is org.apache.struts2.interceptor.ExecuteAndWaitInterceptor, which extends an abstract class com.opensymphony.xwork2.interceptor.MethodFilterInterceptor. Table 4.8 lists the class and different parameters for execAndWait Interceptor.

Table 4.8: execAndWait Interceptor

Interceptor name	Interceptor class
execAndWait	org.apache.struts2.interceptor.ExecuteAndWaitInterceptor
Parameters	
threadPriority	This is the priority assigned to the thread and the default is Thread.NORM_PRIORITY
delay	It shows initial delay in milliseconds to wait before the wait

Table 4.8: execAndWait Interceptor

Interceptor name	Interceptor class
	page is shown
delaySleepInterval	It is a time interval after which the Interceptor constantly checks for the premature completion of running action in background; the default delay is 100 milliseconds

The `execAndWait` Interceptor is configured in `struts-default.xml`, but not included in any of the defined Interceptor stack. It is recommended that this Interceptor should be the last one in the Interceptor stack.

Here's Listing 4.10 providing the configuration of `execAndWait` Interceptor:

Listing 4.10: Configuring `execAndWait` Interceptor in `struts.xml`

```
<action name="anyLongAction" class="com.AnyLongAction">
    <interceptor-ref name="completeStack"/>
    <interceptor-ref name="execAndWait">
        <param name="delay">1000</param>
        <param name="delaySleepInterval">50</param>
    </interceptor-ref>
    <result name="wait">some_wait.jsp</result>
    <result name="success">some_success.jsp</result>
</action>
```

In a given session, the action class `com.AnyLongAction` cannot be run more than once at a time, as the `execAndWait` Interceptor works on a per-session basis. The action will be running in a separate thread and, hence, we cannot access `ActionContext`. The thread started by the Interceptor for an action is named as `actionNameBackgroundProcess`. For initial and subsequent requests before the completion of an action, a wait result is returned. The wait result is responsible for sending subsequent requests back to the action and gives an effect of a self updating progress bar, which does not let the user get bored on seeing a blank page, while an action is being executed. In case no wait result is found, a wait result is automatically created by the framework, which is written in `FreeMarker` and needs `FreeMarker` installed. Hence, providing our own wait result is always a better idea.

When a wait result is returned, the action, running in the background, is placed at the top of the stack allowing the display of progress data in the wait page.

We can also set the initial delay for the appearance of wait page. The initial delay is the time in milliseconds for which the server waits before the wait page is shown to the user. While waiting, in every 100 milliseconds, the running action is checked by the Interceptor for its premature completion. This time interval is known as `delaySleepInterval`, which can also be set as a parameter. Hence, in case the action does not take that long, the wait page is not shown to the user. This Interceptor is very useful for the action having a long execution time.

Exception Interceptor

The Struts 2 Framework provides the functionality of exception handling through this Interceptor. Instead of displaying stack trace for the exception to the user, it is always good to show a nicely designed page describing the real problem to the user. Exception handling with Exception Interceptor enables the mapping of an exception to a result code. In case of any exception, it works as if the action

has returned a result code, instead of throwing an exception. The encountered exception is wrapped with an `ExceptionHolder` and pushed on the stack. This makes exception accessible within your result code easily. Table 4.9 lists the class and different parameters for exception Interceptor.

Table 4.9: exception Interceptor	
Interceptor name	Interceptor class
exception	com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor
Parameters	
logEnabled	It sets to true if the exceptions should be logged otherwise sets to false
logLevel	It sets the log level like trace, debug, info, warn, error or fatal; the default is debug
logCategory	The category to be used; default is ExceptionMappingInterceptor

The class which is executed for Exception Interceptor is `com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor`.

`ExceptionMappingInterceptor` class extends `AbstractInterceptor` class and in addition of having an `intercept()` method, it has getter and setter methods for its fields, like `logCategory`, `logEnabled`, and `logLevel`, etc.

To handle exceptions in this way, we need to configure exceptions in `struts.xml` file and add Exception Interceptor in the Interceptor stack of your actions. The Interceptor should be the first Interceptor in the stack, so that it can access and catch any exception, which may be caused by the action class or the other Interceptors in the stack.

Here's Listing 4.11 showing the configuration of exception Interceptor:

Listing 4.11: Configuring Exception Interceptor in `struts.xml`

```
<package name="default" extends="struts-default ">

    <global-exception-mappings>
        <exception-mapping exception="java.lang.Exception" result="exception"/>
    </global-exception-mappings>

    <action name="modelAction" class="com.kogent.action.ModelAction">
        <interceptor-ref name="exception" />
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="debugging"/>
        <interceptor-ref name="model-driven"/>
        <interceptor-ref name="params"/>
        <interceptor-ref name="conversionError"/>
        <interceptor-ref name="workflow"/>

        <result name="success">. . . </result>
        <result name="error">. . . </result>
    </action>
</package>
```

```

<result name="exception">/exception.jsp</result>
<result name="input">. . . </result>
</action>
</package>

```

File Upload Interceptor

The fileUpload Interceptor is based on MultiPartRequestWrapper class, which parses a multipart request and provides a wrapper around a request. This wrapper is automatically applied for any request that includes a file. The fileUpload Interceptor can add several field errors, given the action implements ValidationAware interface. These error messages are based on key-value pairs stored in properties file. The keys used by Interceptor for different types of field errors added are as follows:

- struts.messages.error.uploading** – If the file could not be loaded
- struts.messages.error.file.too.large** – If the file size exceeds the maximum file size allowed
- struts.messages.error.content.type.not.allowed** – If the content type of the file mismatch all specified content types

Table 4.10 lists the class and different parameters for fileUpload Interceptor.

Table 4.10: fileUpload Interceptor	
Interceptor name	Interceptor class
fileUpload	org.apache.struts2.interceptor.FileUploadInterce ptor
Parameters	
maximumSize	It is the maximum size (in bytes) allowed by the Interceptor for the file reference
allowedTypes	It is a list of allowed content types

When we upload a file using an HTML form, the action class needs some description about the file which is being uploaded, like a File object to handle the file, the content type of the file and the name of the file. These properties are declared in the action class and the setter methods are provided with a specific name convention.

The parameters for these properties are added by fileUpload Interceptor. For example we are uploading a file using an HTML form shown here:

```

<a:form action="doUpload" method="post" enctype="multipart/form-data">
    <a:file name="upload" label="File"/>
    <a:submit/>
</a:form>

```

Now see the configuration of Interceptors and results for the action provided in struts.xml file:

```

<action name="doUpload" class="uploadAction">
    <interceptor-ref name="fileUpload"/>
    <interceptor-ref name="basicStack"/>

```

```
<result name="success">success_page.jsp</result>
</action>
```

For the file being loaded through the field name upload, the Interceptor adds the following parameters:

- ❑ upload: File – It is the actual File object
- ❑ uploadContentType–String – It is the content type of the file
- ❑ uploadFileName–String – It is the actual name of the file, not the HTML form field name for the file

For setting these parameters, the action class should have setter methods with the following names:

- ❑ setUpload(File file)
- ❑ setUploadContentType(String contentType)
- ❑ setUploadFileName (String filename)

Here's the sample code, given in Listing 4.12, for the action class `UploadAction`, which sets three attributes with the parameters added by Interceptor making a File object available in the action class before the execution of `execute()` method of the action class:

Listing 4.12: UploadAction.java

```
public class UploadAction implements Action {
    private File file;
    private String contentType;
    private String filename;

    public void setUpload(File file) {
        this.file = file;
    }

    public void setUploadContentType(String contentType) {
        this.contentType = contentType;
    }

    public void setUploadFileName(String filename) {
        this.filename = filename;
    }
    ...
}
```

I18n Interceptor

The i18n Interceptor sets the locale (an object defining particular geographical or cultural region) for the current action request with the locale specified in the session. This Interceptor can also set the locale in the session, according to the value of a specific HTTP request parameter. Hence, this Interceptor allows the dynamic changing of locale for the different user sessions by the application. The Interceptor is required to be implemented in the internationalized applications, which support multiple languages. The request parameter sent for setting locale is removed during the execution of i18n Interceptor and need not to be handled further by the action. Table 4.11 lists class and different parameters for i18n Interceptor.

Table 4.11: i18n Interceptor	
Interceptor name	Interceptor class
i18n	com.opensymphony.xwork2.interceptor.I18nInterceptor
Parameters	
parameterName	It is the name of HTTP request parameter which provides the locale to be saved in the user session; the default value is request_locale
attributeName	It is the name of the session key to store selected locale; the default is WW_TRANS_I18N_LOCALE

We can save the locale in the user session by sending a request parameter name `request_locale`. The i18n Interceptor separates this parameter and sets its value as the locale for the user session. For example, a request path similar to `http://localhost:8080/Test>Hello.action?request_locale=es` sets locale es in the user session, as shown in the following code lines:

```
<package name="hello-default" extends="struts-default">
    <action name="Hello" class="Hello">
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="basicStack"/>
        <result>/Hello.jsp</result>
    </action>
</package>
```

Logger Interceptor

The Logger Interceptor, when added to the Interceptor stack of an action mapping, logs the start and end point of the execution of action or the execution of whole stack defined for the action including all Interceptors and action itself. The starting point of the execution of stack (including action class itself) and its finishing point are logged with detail of date and time in the log file. Printing this sort of messages may be helpful in debugging of the application. Table 4.12 lists the class and different parameters for logger Interceptor.

Table 4.12: logger Interceptor	
Interceptor name	Interceptor class
logger	com.opensymphony.xwork2.interceptor.LoggingInterce ptor
Parameters	
There is no parameter defined for this Interceptor	

We can configure it anywhere in the Interceptor stack for the action. The two ways to configure this Interceptor is shown in Listing 4.13.

Listing 4.13: Configuring Logger Interceptor in struts.xml

```
<!-- prints message before and after the otherinterceptors starts and
finishes -->
<action name="action1" class="com.Action1">
    <interceptor-ref name="logger"/>
    <interceptor-ref name="completeStack"/>
    <result name="success">some_page.jsp</result>
</action>

<!-- prints a message immediately before and after the action is executed -->
<action name="action2" class="com.Action2">
    <interceptor-ref name="completeStack"/>
    <interceptor-ref name="logger"/>
    <result name="success">some_page.jsp</result>
</action>
```

On the execution of an action, say action1, having Logger Interceptor in its Interceptor stack prints following messages in the log file:

```
May 4, 2007 2:30:34 PM
com.opensymphony.xwork2.interceptor.LoggingInterceptor logMessage
INFO: Finishing execution stack for action //action1
May 4, 2007 2:30:34 PM
com.opensymphony.xwork2.interceptor.LoggingInterceptor logMessage
INFO: Starting execution stack for action //action1
```

Model-Driven Interceptor

The Model-Driven Interceptor is responsible for looking ModelDriven actions (discussed in chapter 3) and adds the model of the action into the actions value stack making it available in the action. The ModelDriven actions are those which implements com.opensymphony.xwork2.ModelDriven. Struts 2 Framework provides the flexibility of defining attributes and their setter methods in action class itself and in some other form bean like class (Model). To set request attributes to the fields of some other model object, we need to implement the following two things:

- ❑ The action class must implement ModelDriven interface
- ❑ The Model-Driven Interceptor must be applied to the action

Table 4.13 lists the class and different parameters for model-driven Interceptor.

Table 4.13: Model-Driven Interceptor

Interceptor name	Interceptor class
model-driven	com.opensymphony.xwork2.interceptor.ModelDrivenI nterceptor
Parameters	
There is no parameter defined for this Interceptor	

Instead of finding the setter methods for the request attributes in the action, the model is first retrieved through the `getModel()` method, which is overridden in action class to return the object of model class and searched for the setter matching the attribute. In case of no matching setter in the model class, the value can be set on action. The model class reference in action must be initialized before it is populated, and this can be done in the constructor of action class. For example, we have an action class `MyAction` and a model class `MyModel` with a field 'name'.

Here's Listing 4.14 showing the structure of `MyAction` and `MyModel` class:

Listing 4.14: Sample Model-driven action and Model class being used.

```
//Structure of a ModelDriven action class.
package com.kogent.action;
import com.opensymphony.xwork2.ModelDriven;
public class MyAction extends ActionSupport implements ModelDriven{

    private MyModel mymodel;

    public MyAction(){
        mymodel =new MyModel();
    }

    public Object getModel(){
        return mymodel;
    }

    public String execute() throws Exception {
        if(mymodel.getName().equals("scott"))
            return SUCCESS;
        else{
            this.addActionError(getText("app.invalid"));
            return ERROR;
        }
    }
    public void validate() {

        String name= mymodel.getName();
        if( (name == null ) || (name.length() == 0) ) {
            this.addFieldError("name", getText("app.username.blank"));
        }
    }
}

//Structure of model class which is simple bean class.
package com.kogent.action;
public class MyModel{

    private String name;
    public String getName() {
        return name;
    }
}
```

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

The Model-Driven Interceptor must be placed before the `StaticParametersInterceptor` and `ParametersInterceptor` in the Interceptor stack, if the parameters are to be applied to the model. The Model-Driven Interceptor pushes only the not null model into the value of stack of action. We can use a default Interceptor stack `modelDrivenStack` or `defaultStack`, which is defined in `struts-default.xml` file or alternately add Model-Driven Interceptor in the stack.

Here's Listing 4.15 showing the configuration of Model-Driven Interceptor:

Listing 4.15: Configuring Model-Driven Interceptor in struts.xml

```
<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">
    <action name="myaction" class="com.kogent.action.MyAction">
        <interceptor-ref name="model-driven"/>
        <interceptor-ref name="basicStack"/>
        <result name="success">/success.jsp</result>
        <result name="error">/input.jsp</result>
        <result name="input">/input.jsp</result>
    </action>
</package>
</struts>
```

Parameters Interceptor

The `ParametersInterceptor` sets all parameters on the value stack. All parameters can be obtained by the Interceptor using `ActionContext.getParameters()` method. The values are set in the value stack using `ValueStack.setValue(String, Object)`. In this way, the values submitted in a form request are applied to an action in value stack. Table 4.14 lists the class and different parameters for `params` Interceptor.

Table 4.14: params Interceptor

Interceptor name	Interceptor class
params	com.opensymphony.xwork2.interceptor.ParametersInterceptor
Parameters	
ordered	When set to true, action properties are set top-down which means top action's properties are set first

The parameter names are OGNL statements and any value in the parameter map is not allowed by the Interceptor, if the expression contains an assignment operator (=), multiple expressions () or references of the objects from the context (#). This is done with the help of the `acceptableName(String)` method. If the action implements `ParameterNameAware` interface, the boolean

`acceptableParameterName(String parameterName)` method is used to know whether the parameter with the given name is acceptable by the action or not.

In addition, three different flags are also set when this Interceptor is invoked. These flags are as follows:

- ❑ `XWorkMethodAccessor.DENY_METHOD_EXECUTION`—This flag is turned on to restrict the invocation of methods. The expressions invoking some methods, like `person.someMethod()`, will be prohibited.
- ❑ `InstantiatingNullHandler.CREATE_NULL_OBJECTS`—This flag is turned on to make sure that any null reference is automatically created.
- ❑ `XWorkConverter.REPORT_CONVERSION_ERRORS`—This flag, if set, indicates the occurrence of some errors while converting the values to their final data types. Setting of this flag indicates that the conversion errors are reported in the action context.

To configure `ParametersInterceptor` Interceptor we can use the default Interceptor stacks, like `basicStack` and `defaultStack`, or can add an Interceptor reference with name `param`. For instance, see the following code:

```
<action name="anyAction" class="com.AnyAction">
    <interceptor-ref name="params"/>
    <result name="success">success.jsp</result>
</action>
```

Prepare Interceptor

The `PrepareInterceptor` Interceptor invokes `prepare()` method on the actions, which implement `com.opensymphony.xwork2.Preparable` interface. Sometimes we need to ensure some processing before the `execute` method is invoked. In this scenario, the `Prepare Interceptor` helps by invoking the `prepare()` method of the action class containing some logic, like initializing some null objects, loading an object from database, etc.

For example, we have an object `user` with properties, like `userid` and `username`. Before the parameters are set on this object by the `param` Interceptor, the object should be initialized. This is ensured by the use of `Prepare Interceptor` which consequently calls the `prepare()` method where we can provide code for creating a new user object or loading it from database. Table 4.15 lists the class and different parameters for `Prepare Interceptor`.

Table 4.15: Prepare Interceptor	
Interceptor name	Interceptor class
prepare	<code>com.opensymphony.xwork2.interceptor.PrepareInterceptor</code>
Parameters	
<code>alwaysInvokePrepare</code>	If set to true, <code>prepare()</code> method will always be invoked; the default is true

We can directly invoke different methods of an action class having signature, like `String methodName()`, by providing the method name in the action mapping. We can create different `prepareMethodName()` methods for preparing action before a specific method is invoked. For

example, a method `prepareUpdate()` can be created to prepare action before the `String update()` method is invoked.

Hence, we can say that if the action class implements `Preparable` interface and the `Prepare` Interceptor is added in the Interceptor stack defined for action, the following statements are true:

1. If the action class has `prepare{MethodName}()`, it will be invoked.
2. If no `prepare[MethodName]()` is found, `prepareDo{MethodName}` is searched and executed.
3. If `alwaysInvokePrepare` property of the Interceptor is set to true, the `prepare()` method is invoked irrespective of the fact whether step 1 and step 2 has been taken.

Here's Listing 4.16 showing the configuration of `prepare` Interceptor:

Listing 4.16: Configuring Prepare Interceptor in struts.xml

```
<package name="default" extends="struts-default">
    <interceptors>
        <interceptor-stack name="myStack">
            <interceptor-ref name="prepare"/>
            <interceptor-ref name="model-driven"/>
            <interceptor-ref name="static-params"/>
            <interceptor-ref name="params"/>
            <interceptor-ref name="workflow"/>
        </interceptor-stack>
    </interceptors>
    <action name="someAction" class="com.someAction">
        <interceptor-ref name="mystack"/>
        <result name="success">/success_page.jsp</result>
    </action>
</package>
</struts>
```

Sometimes we may need to apply parameters directly to an object, which is to be loaded externally (from database). Here, we need the invocation of `params` Interceptor before the `Prepare` Interceptor. We can add `params` Interceptor twice in the action's Interceptor stack or can use `paramsPrepareParamsStack` Interceptor stack, which is defined in `struts-default.xml` file. This Interceptor stack is similar to `defaultStack`, except that it has an extra `params` Interceptor added before `Prepare` Interceptor.

Roles Interceptor

This Interceptor is used to make sure that the user having the correct role executes the action. We can set the allowed/disallowed list of roles for this Interceptor. It has not been defined in any default Interceptor stack, so you have to add it in action's Interceptor stack to use it. In addition to `intercept()` method, this Interceptor has methods like `void setAllowedRoles(String role)` and `void setDisallowedRoles(String role)` which are used to set the list of allowed and disallowed roles for the action. Another method, `boolean isAllowed(HttpServletRequest req, Object action)` is used to check whether the request is to be allowed for the action or not. To handle all rejected/unauthorized requests, `String handleRejection(ActionInvocation invocation, HttpServletResponse response)` method is used. Table 4.16 lists the class and different parameters for `roles` Interceptor.

Table 4.16: Roles Interceptor

Interceptor name	Interceptor class
roles	org.apache.struts2.interceptor.RolesInterceptor
Parameters	
allowedRoles	It is a list of allowed roles
disallowedRoles	It is a list of roles that are not allowed

Here's Listing 4.17 showing the configuration of Roles Interceptor:

Listing 4.17: Configuring Roles Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="completeStack"/>
    <interceptor-ref name="roles">
        <param name="allowedRoles">admin,member</param>
    </interceptor-ref>
    <result name="success">success_page.jsp</result>
</action>
```

Scope Interceptor

In any application, we may have a number of parameters, which are to be managed for their values in the application scope or session scope. Now, instead of checking each action for the required parameter, the Scope Interceptor is used to look for the specified list of parameters and pull these parameters from the given scope.

When action starts, all the listed properties are set with their current values from the given scope. Just after the action's completion, all these listed properties are taken back and put back in the respective scope. This Interceptor provides session level consistency by using session-level locking. Table 4.17 lists the class and different parameters for Scope Interceptor.

Table 4.17: Scope Interceptor

Interceptor name	Interceptor class
scope	org.apache.struts2.interceptor.ScopeInterceptor
Parameters	
session	It is a list of properties which are to be bound to session scope
application	It is a list of properties, which are to be bound to application scope
key	It is an unique attribute key prefix, the value for this parameter can be CLASS, ACTION or any other string. Using CLASS as value for this parameter creates a key prefix based on action namespace and action class and it is the default one too.

Table 4.17: Scope Interceptor

Interceptor name	Interceptor class
	Similarly, using ACTION as value for key parameter, creates a key prefix based on action namespace and action name. Any other value is taken literally as key prefix.
type	The values for these parameters are start and end. Setting type parameter with start indicates that the action is the first one in the sequence of actions, and properties in session scope are reset to their defaults. Similarly, using end indicates that after this action execution all properties from session are removed. Any other value or no value declares that the session properties are set before the action execution and put back into the session after the execution
sessionReset	It takes a Boolean value causing resetting of all session properties with default values of action or values in application scope

The following configuration, done in struts.xml file, shows a Scope Interceptor is configured for an action to be executed:

```

<action name="someAction" class="com.kogent.action.SomeAction">
    <interceptor-ref name="basicStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
    <interceptor-ref name="scope">
        <param name="session">field1, field2</param>
        <param name="key">ACTION</param>
        <param name="type">start</param>
        <param name="autoCreateSession">true</param>
    </interceptor-ref>
    <result name="success">/login_success.jsp</result>
    <result name="error">/login.jsp</result>
    <result name="input">/login.jsp</result>
</action>

```

The ScopeInterceptor class extends the AbstractInterceptor class and implements com.opensymphony.xwork2.interceptor.PreResultListener Interceptor. The PreResultListeners may be registered with an ActionInvocation to get a callback after the action is executed, but before the result is executed. On this callback, the Interceptor takes a snapshot of the action properties and sets them into session/application scope accordingly.

Scoped-Model-Driven Interceptor

The scoped-model-driven Interceptor enables the scoped model-driven actions. The action class, which implements com.opensymphony.xwork2.interceptor.ScopedModelDriven interface is known as scope model-driven actions. The ScopedModelDriven interface extends the ModelDriven interface. The scoped-model-driven Interceptor get activated for only the scope model-driven actions. This Interceptor is similar to the Model-Driven Interceptor, and to set the model object to the action

class, the `getModel()` method can be used. The `resolveModel()` of this Interceptor is used to obtain the model object and takes arguments, like `ObjectFactory` factory, `ActionContext` `actionContext`, `String modelClassName`, `String modelScope`, and `String modelName`. We have setter methods for fields, like ‘`className`’, ‘`name`’, and ‘`scope`’.

Hence, the main objective of this Interceptor is to obtain the model class from the configured scope and provide it to the action. Table 4.18 lists the class and different parameters for Scoped-Model-Driven Interceptor.

Table 4.18: scoped-model-driven Interceptor	
Interceptor Name	Interceptor Class
scoped-model-driven	<code>com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor</code>
Parameters	
<code>className</code>	It is name of model class, the default is the name of the object returned by <code>getModel()</code> method
<code>name</code>	It is a key used when storing or retrieving instance in a scope
<code>scope</code>	It is the scope to store the model; default is <code>request</code>

Here’s Listing 4.18 showing the configuration of Scoped-Model-Driven Interceptor:

Listing 4.18: Configuring scoped-Model-Driven Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="scoped-model-driven">
        <param name="scope">session</param>
        <param name="name">model</param>
        <param name="className">com.MyActionForm</param>
    </interceptor-ref>
    <interceptor-ref name="basicStack"/>
    <result name="success">success_page.jsp</result>
</action>
```

Servlet-Config Interceptor

The Servlet-Config Interceptor sets the action properties, based on the different interfaces implemented by the action class. For example, if the action implements `org.apache.struts2.interceptor.ParameterAware` interface, a map containing all the input parameters as name/value pairs is set on the action. This Interceptor is designed to set all the properties on the action, given that the action is aware of servlet parameters, servlet context, session, request, application, etc. To make the action aware of these things, the action needs to implement the following interfaces:

- `ServletContextAware`
- `RequestAware`
- `ServletRequestAware`
- `SessionAware`

- ServletResponseAware
- ApplicationAware
- ParameterAware
- PrincipalAware

The implementation of these interfaces by the action class gives different setter methods like:

- void setServletContext(ServletContext context)–ServletContextAware
- void setServletRequest(HttpServletRequest request)–ServletRequestAware
- void setServletResponse(HttpServletRequest response)–Servlet Response-Aware
- void setParameters(Map parameters)–ParameterAware
- void setRequest(Map request)–RequestAware
- void setSession(Map session)–SessionAware
- void setApplication(Map application)–ApplicationAware
- void setPrincipalProxy(PrincipalProxy principalProxy)–PrincipalAware

Table 4.19: Servlet-Config Interceptor

Interceptor name	Interceptor class
servlet-config	org.apache.struts2.interceptor.ServletConfigInt erceptor
Parameters	
There is no parameter defined for this Interceptor	

This Interceptor is part of all Interceptor stacks defined in struts-default.xml file. Hence, when we are using any of these Interceptor stacks, we are automatically using servlet-config Interceptor.

Static Parameters Interceptor

All action classes implementing com.opensymphony.xwork2.config.entities.Parameterizable interface can receive a Map of all the static parameters, which are defined in action configuration.

This needs the addition of static-params Interceptor in the Interceptor stack of the action. The implementation of Parameterizable interface brings methods in scenes like void addParam(String name, Object value), Map getParams(), and void setParams(Map<String, Object> params) which are called by the static-params Interceptor to set static parameters on to the action.

The class for this Interceptor is com.opensymphony.xwork2.interceptor.StaticParametersInterceptor which extends Abstract-Interceptor. There is no parameter defined for this Interceptor.

Table 4.20: static-params Interceptor

Interceptor name	Interceptor class
static-params	com.opensymphony.xwork2.interceptor.StaticPar ametersInterceptor

Parameters
There is no parameter defined for this Interceptor

Message Store Interceptor

The error messages and field errors of some ValidationAware action are, by default, not available longer than an HTTP request. If we want to store these error messages and field error messages into HTTP Session making them available at later stages, we can use store Interceptor (Table 4.21).

Table 4.21: store Interceptor	
Interceptor Name	Interceptor Class
store	org.apache.struts2.interceptor.MessageStoreInterceptor
Parameters	
allowRequestParameterSwitch	It sets to true to enable request parameter that can change the operation mode of this Interceptor
requestParameterSwitch	It is the request parameter used to indicate the mode of Interceptor
operationMode	It is the operation mode of the Interceptor (STORE, RETRIEVE, NONE); default is NONE

In STORE mode, this Interceptor stores the messages (error messages, field error messages) into HTTP session. While in RETRIEVE mode, the stored messages are retrieved and put back into the ValidationAware action. In default mode, which is NONE, the Interceptor does nothing.

There are basically two different methods to switch the working mode of this Interceptor. In the first method, the Interceptor parameter `operationMode` can be set to STORE or RETRIEVE as shown here:

```
<action name="..." class="...">
    <interceptor-ref name="store">
        <param name="operationMode">STORE</param>
    </interceptor-ref>
    <interceptor-ref name="defaultStack" />
    ...
</action>
```

The second method is using a request parameter named `operationMode`. Changing the mode of this Interceptor in this way requires that the `allowRequestParameterSwitch` must be set to true which is also the default. So the following URL can change the mode of the Interceptor to STORE:

<http://localhost:8080/Test/someAction.action?operationMode=STORE>

Here's Listing 4.19 showing the configuration of store Interceptor with different values for its `operationMode` parameter:

Listing 4.19: Configuring store Interceptor in struts.xml

```
<action name="action1" class="...">
    <interceptor-ref name="store">
        <param name="operationMode">STORE</param>
    </interceptor-ref>
    <interceptor-ref name="defaultStack" />
    <result name="input" type="redirect">action2.action</result>
    <result name="success" type="dispatcher">success_page.jsp</result>
</action>

<action name="action2" class="...">
    <interceptor-ref name="store">
        <param name="operationMode">RETRIEVE</param>
    </interceptor-ref>
    <result>fail_page.jsp</result>
</action>
```

Timer Interceptor

This Interceptor logs the total amount of time in milliseconds elapsed during the execution of action including/excluding the execution time of Interceptors in the Interceptor stack of the action. The working of this Interceptor relies on the Commons Logging API to report the execution time value in the log file. Table 4.22 lists the class and different parameters for Timer Interceptor.

Table 4.22: Timer Interceptor

Interceptor name	Interceptor class
timer	com.opensymphony.xwork2.interceptor.TimerInterceptor
Parameters	
logLevel	It defines the log level to be used (trace, debug, info, warn, error, fatal); default value for this parameter is info
logCategory	The category to be used; default is TimerInterceptor

The parameters `logLevel` and `logCategory` enables the logging of execution time in the logfile. We can have custom message format by extending this Interceptor and overriding its `invokeUnderTiming()` method. This Interceptor has not been defined in any default Interceptor stack and is added to the action's Interceptor stack, as shown in Listing 4.20.

Listing 4.20: Configuring timer Interceptor in struts.xml

```
<!-- Logs execution time of action only -->
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="completeStack"/>
    <interceptor-ref name="timer">
        <param name="logLevel">warn</param>
    </interceptor-ref>
    <result name="success">success_page.jsp</result>
</action>
```

```
<!-- Logs execution time of action and its Interceptors-->
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="timer"/>
    <interceptor-ref name="completeStack"/>
    <result name="success">success_page.jsp</result>
</action>
```

The message displayed in logfile showing the execution time of action someAction is similar to what is shown here:

```
//if the logLevel parameter is set to 'info'
INFO: Executed action [/someAction!execute] took 78 ms.
Or
//if the logLevel parameter is set to 'warn'
WARNING: Executed action [/someAction!execute] took 78 ms.
```

Token Interceptor

Sometimes user may double-click on the submit button or the back button, which can cause some side-effects on the execution of action class. The Token Interceptor makes sure that there is no such side-effects caused by any action of careless users. This Interceptor achieves this by allowing the processing of one request per token. When an invalid token is found, the Interceptor returns the `invalid.token` result, which is mapped to the action configuration. Table 4.23 lists class and different parameters for token Interceptor.

Table 4.23: token Interceptor	
Interceptor name	Interceptor class
token	org.apache.struts2.interceptor.TokenInterceptor
Parameters	
There is no parameter defined for this Interceptor	

The `TokenInterceptor` class extends `com.opensymphony.xwork2.interceptor.MethodFilterInterceptor` class and, hence, inherits methods like `setExcludeMethods()`, `setIncludeMethods()`. The parameters, like `excludeMethods` and `includeMethods`, can be provided to this Interceptor which is the comma-separated list of methods.

A token can be set in the form by using a token tag. All forms, which are submitted to the actions having this Interceptor in their Interceptor stack, require this tag. A request without this token are processed as request with invalid token. The key `struts.messages.invalid.token` is used for the action errors generated by Token Interceptor.

Another Interceptor class `TokenSessionStoreInterceptor` extends `TokenInterceptor` class and handles invalid tokens in a better way. The protected methods `handleInvalidToken()` and `handleValidToken()` are available in the subclass for some better logic implementation as done by `TokenSessionStoreInterceptor` class.

Here's Listing 4.21 showing the configuration of Token Interceptor:

Listing 4.21: Configuring Token Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="token"/>
    <interceptor-ref name="basicStack"/>
    <result name="success">success_page.jsp</result>
</action>

<!-- In this case, someMethod of the action class is not checked
     for invalid token -->
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="token">
        <param name="excludeMethods">someMethod</param>
    </interceptor-ref>
    <interceptor-ref name="basicStack"/>
    <result name="success">success_page.jsp</result>
</action>
```

Token Session Interceptor

The `TokenSessionStoreInterceptor` extends `TokenInterceptor` class and provides some advance logic for handling invalid tokens. In case of multiple requests using the same session, this Interceptor is capable of handling the situation more intelligently. All the subsequent requests are blocked by the Interceptor, while the first request is being processed. Instead of returning `invalid.token` code for the subsequent requests, the Interceptor attempts to display the same response as the response would, in case of no multiple requests. Table 4.24 lists the class and different parameters for token-session Interceptor.

Table 4.24: token-session Interceptor	
Interceptor name	Interceptor class
token-session	org.apache.struts2.interceptor.TokenSessionStoreInterceptor
Parameters	
There is no parameter defined for this Interceptor	

The `TokenSessionStoreInterceptor` extends `MethodFilterInterceptor`, and hence, is capable of deciding its applicability for the selective methods in the action class, which can be listed by using `excludeMethods` and `includeMethods` parameters.

Here's Listing 4.22 showing the configuration of token-session Interceptor:

Listing 4.22: Configuring token-session Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="token-session"/>
    <interceptor-ref name="basicStack"/>
    <result name="success">success_page.jsp</result>
</action>
```

```
<!-- In this case, someMethod of the action class is not checked
   for invalid token -->
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="token-session">
        <param name="excludeMethods">someMethod</param>
    </interceptor-ref name="token-session">
    <interceptor-ref name="basicStack"/>
    <result name="success">success_page.jsp</result>
</action>
```

Validation Interceptor

The Validation Interceptor checks the action against all validation rules declared in Validation Framework configuration files, like SomeAction-validation.xml. The Interceptor adds field-level and action-level error messages into the action context. The action class must implement the com.opensymphony.xwork2.ValidationAware interface. In other words, the Validation Interceptor enables the execution of action class through standard Validation Framework. Table 4.25 lists the class and different parameters for validation Interceptor.

Table 4.25: Validation Interceptor

Interceptor name	Interceptor class
validation	org.apache.struts2.interceptor.validation.AnnotationValidationInterceptor
Parameters	
There is no parameter defined for this Interceptor	

If any of the default Interceptor stacks, like validationWorkflowStack, paramsPrepareParamsStack, defaultStack, and executeAndWaitStack is being used for the action, the Validation Interceptor is indirectly being used as this Interceptor is part of all these default Interceptors. This is the last or second last Interceptor in the stack.

The org.apache.struts2.interceptor.validation.AnnotationValidationInterceptor class extends com.opensymphony.xwork2.validator.ValidationInterceptor class, which is further a subclass of MethodFilterInterceptor class. Hence, this Interceptor class extends all the protected and public methods of ValidationInterceptor and MethodFilterInterceptor class. We can define the list of methods to be excluded and included. The Interceptor does nothing if the name of the action method to be invoked is in the list of excludeMethods parameter. The two methods extended from ValidationInterceptor class, boolean validateAnnotatedMethodOnly() and void setValidateAnnotatedMethodOnly(String str), are used to check and set the field validateAnnotatedMethodOnly, which determines whether the validate() method is to be called always or only per annotated method. The Interceptor do not validate action method with @SkipValidation annotation.

The use of Validation Interceptor does not change the workflow of the action request. The Validation Interceptor is often used in conjunction with the Workflow Interceptor.

Here's Listing 4.23 showing the configuration of Validation Interceptor:

Listing 4.23: Configuring Validation Interceptor in struts.xml

```
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
    <result name="success">success_page.jsp</result>
</action>

<!-- in this case someMethod of the action class is not validated -->
<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation">
        <param name="excludeMethods">someMethod</param>
    </interceptor-ref>
    <interceptor-ref name="workflow"/>
    <result name="success">success_page.jsp</result>
</action>

<!-- in this case only the annotated methods of the action
     class are validated -->

<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation">
        <param name="validateAnnotatedMethodOnly">true</param>
    </interceptor-ref>
    <interceptor-ref name="workflow"/>
    <result name="success">success_page.jsp</result>
</action>
```

Workflow Interceptor

The Workflow Interceptor provides some basic validation workflow before the rest of the Interceptor chain is allowed to continue. As the name suggests, this Interceptor ensures a specific flow of execution. The execution in the workflow is dependent on the interface implemented by the action class to be executed. For instance, let's look at a few cases:

- ❑ **Case 1**—The action implements `Validateable` interface
 - **Result**—Action's `validate()` method is invoked.
- ❑ **Case 2**—The action implements `ValidationAware` interface
 - **Result**—Action's `hasErrors` method is called. If this method returns true indicating some error, the Interceptor stops the execution and immediately returns `Action.INPUT`.
- ❑ **Case 3**—The action does not implement either of interfaces.
 - **Result**—The Interceptor does nothing.

Table 4.26 lists the class and different parameters for `workflow` Interceptor.

Table 4.26: Workflow Interceptor

Interceptor name	Interceptor class
workflow	com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor
Parameters	
alwaysInvokeValidate	If true, the validate() method is always invoked; default is true
inputResultName	The result name to be returned when action or field error is found; default is input

The Workflow Interceptor should be used with the Validation Interceptor, if all the validation rules are defined in the XML file, instead of defining them in action's validate() method. The DefaultWorkflowInterceptor class extends MethodFilterInterceptor and, hence, we can define a list of methods in the excludeMethods parameter. The invocation of methods listed in excludeMethods parameter is not affected by this Interceptor.

We can add some extra logic implementation for different methods by creating separate methods having names similar to validate{MethodName}, which is executed before the validate() method. This enables us in executing some validation logic specific to the method name specified. For example, the validateInput() method is invoked before the String input(void) method is called.

Here's Listing 4.24 showing the configuration of workflow Interceptor:

Listing 4.24: Configuring Workflow Interceptor in struts.xml

```

<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
    <result name="success">success_page.jsp</result>
</action>

<!-- In this case someMethod of the action is not passed through
     the workflow process -->

<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow">
        <param name="excludeMethods">someMethod</param>
    </interceptor-ref name="workflow">
    <result name="success">success_page.jsp</result>
</action>

<!-- In this case, the result named "error" will be used if any
     action/field error is found -->

<action name="someAction" class="com.SomeAction">
    <interceptor-ref name="params"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow">

```

```
<param name="inputResultName">error</param>
</interceptor-ref>
<result name="success">success_page.jsp </result>
</action>
```

Writing Custom Interceptors

This framework provides the flexibility to create our own Interceptor classes to enable additional logic which can be separated and reused in the Interceptor stack of different action classes. The custom Interceptor class needs to be defined in `struts.xml` file of the application. Further, we can define a default Interceptor stack, including the newly designed Interceptor to use it with the actions.

All Interceptors in a stack, and hence the custom Interceptors too, can interact with the values provided by the user as input. The Interceptors can also interact with the action after they have completed the processing, they are created for. The Interceptor can intercept the processing in between and can return some result value also.

The main method in every Interceptor is its `String intercept(ActionInvocation invocation)` method, which takes a reference of `com.opensymphony.xwork2.ActionInvocation` interface. The `ActionInvocation` represents the execution state of the action. The instances of all Interceptors in the Interceptor stack of the action and instance of action class itself are held by `ActionInvocation`. The processing of `ActionInvocation` is divided in steps and every step is invoked by the `ActionInvocation.invoke()` method. The first `invoke()` on `ActionInvocation` is called by `ActionProxy` and, further, each Interceptor calls the `invoke()` method to execute next Interceptor. The action class is executed only after the execution of all the Interceptors defined in the Interceptor stack for that action.

The Interceptor class can access various resources by invoking methods on `ActionInvocation` object, which is passed as argument to its `intercept()` method. These methods are shown in Table 4.27.

Table 4.27: Methods of ActionInvocation interface

Method	Description
<code>Object getAction()</code>	It returns the action associated with the <code>ActionInvocation</code>
<code>ActionContext getInvocationContext()</code>	It returns <code>ActionContext</code> associated with the <code>ActionInvocation</code>
<code>ActionProxy getProxy()</code>	It returns the <code>ActionProxy</code> holding this <code>ActionInvocation</code>
<code>ValueStack getStack()</code>	It returns an object of <code>ValueStack</code> , which represents the value stack of the action
<code>String invoke()</code>	Invokes the next step in the processing of <code>ActionInvocation</code> . This may be the execution of next Interceptor or the action itself
<code>Result getResult()</code>	If the <code>ActionInvocation</code> is executed before and the <code>Result</code> is an instance of <code>ActionChainResult</code> , this method will walk down the chain of

Table 4.27: Methods of ActionInvocation interface

Method	Description
	ActionChainResults, until it finds a non-chain result, which will be returned.
String getResultCode()	It gets the result code returned from this ActionInvocation

The ActionContext returned by getInvocationContext() method can further provide other things, like HttpServletRequest and HttpSession, etc.

Here's Listing 4.25 showing the sample intercept() method of an Interceptor:

Listing 4.25: A sample Interceptor and its intercept() method.

```

package com.kogent.interceptors;
public class SimpleInterceptor extends AbstractInterceptor {
    public String intercept (ActionInvocation invocation) throws Exception {
        final ActionContext context = invocation.getInvocationContext();
        HttpServletRequest request = (HttpServletRequest) context.get(HTTP_REQUEST);
        HttpSession session = request.getSession (true);
        Object user = session.getAttribute (USER_HANDLE);
        if (user == null) {
            String loginAttempt = request.getParameter(LOGIN_ATTEMPT);
            if (! StringUtil.isNotBlank (loginAttempt) ) {

                if (some condition ) {
                    return "success";
                } else {
                    Object action = invocation.getAction ();
                    if (action instanceof validationAware) {
                        ((validationAware) action).addActionError("Username
                            or password incorrect.");
                    }
                }
            }
        // Either the login attempt failed or the user hasn't tried to login yet,
        // and we need to send the login form.
        return "login";
        } else {
        return invocation.invoke ();
    }
}
}

```

After creating your own Interceptor, this needs declaration in struts.xml to configure this custom Interceptor so that it can be used in the Interceptor stacks defined in action mappings provided in struts configuration file. The sample configuration of a custom Interceptor is shown here:

```

<struts>
<package name="my-default" extends="struts-default">
<interceptors>
<interceptor name="login" class=" com.kogent.interceptors.SimpleInterceptor " />

```

```
</interceptors>
</package>
</struts>
```

The Interceptor pattern is a very powerful one, well worth looking at. Struts 2 uses it to implement a great deal of its functionality.

Now that we have discussed about the different Struts 2 framework Interceptors, it is high time to implement these Interceptors in our applications according to the different features provided by them. Though we created a number of applications, using Interceptors, in the previous chapters, we never emphasized on the use of them earlier. The “Immediate Solutions” section will now provide an application with different action mappings, giving us the real implementation of different Interceptors.

Immediate Solutions

Implementing Interceptors

The Interceptors provide different kinds of preprocessing, before and after the action class is executed. We can use the default Interceptor stacks or can define our own, according to the need of the application and the particular action. Let's create an application, by writing code for various JSP pages, action classes, and providing action mapping for all action classes declaring all Interceptors for the action's Interceptor stack.

Before implementing actions and Interceptors, first of all create a directory structures similar to what is shown in Figure 4.2. All .java files discussed in this application is to be compiled and placed at the location, as shown in Figure 4.2. All JSP pages are to be saved in a project folder struts2_i directly. The standard files, like web.xml, struts.xml, struts.properties, and other resource bundles (ApplicationResources.properties) files should be placed as per the Struts 2 standards as shown in Figure 4.2.

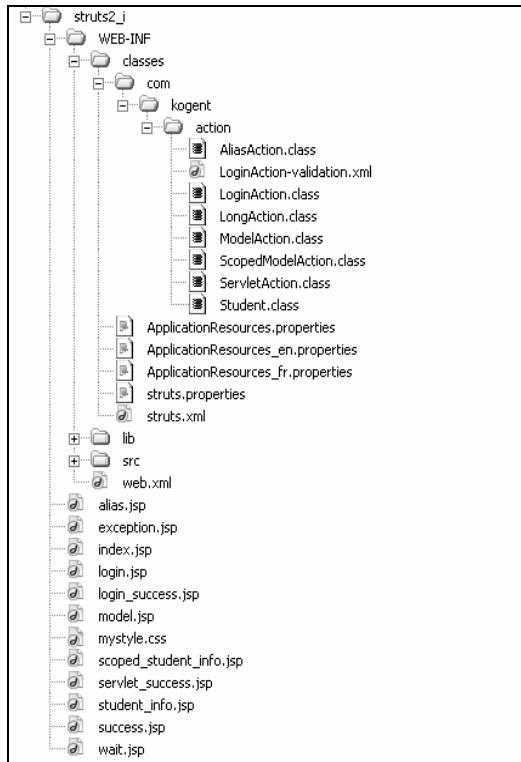


Figure 4.2: Directory structure of the Struts 2 application.

We have already developed Struts 2 applications with the similar directory structure; hence, here we are not describing further where the things, like classes, JSP pages, JAR files, configuration files like web.xml, struts.xml, and other properties files containing global resources are saved. You need to compile class files, create WAR file and deploy it on Web server as discussed in Chapter 2. We are directly jumping over the writing of various action classes and providing action mapping in struts.xml file declaring Interceptors to be used for each action. The JSP pages are created to invoke the action and to represent the final result to the user. The first JSP page that is to be designed here is index.jsp, which provides different hyperlinks to run different examples implementing different set of Interceptors.

Here's Listing 4.26 showing the code for index.jsp page (you can find index.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.26: index.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <table align="center" width=400>
        <tr><td><h2>Sruts 2 Interceptors</h2></td></tr>

        <tr><td><s:a href="alias.jsp">Interceptor Example 1</s:a></td></tr>
        <tr><td>alias, basicStack </td></tr>
        <tr><td>&nbsp;</td></tr>
        <tr><td><s:a href="model.jsp">Interceptor Example 2</s:a></td></tr>
        <tr><td>
            exception, prepare, debugging, model-driven, params,
            conversionError, workflow
        </td></tr>
        <tr><td>&nbsp;</td></tr>
        <tr><td>
            <s:a href="servletAction.action">Interceptor Example 3</s:a>
        </td></tr>
        <tr><td>servlet-config, scoped-model-driven </td></tr>
        <tr><td>&nbsp;</td></tr>
        <tr><td>
            <s:a href="longAction.action">Interceptor Example 4</s:a>
        </td></tr>
        <tr><td>completeteststack, execAndwait</td></tr>
        <tr><td>&nbsp;</td></tr>
        <tr><td><s:a href="login.jsp">Interceptor Example 5</s:a> </td></tr>
        <tr><td>basicStack, validation, workflow, scope </td></tr>
    </table>
</body>
</html>
```

Save this page and other JSP pages in the project folder directly, i.e. struts_i. This JSP page simply provides a hyperlink to other JSP pages, which requests an action or sometimes these hyperlinks itself invoke the action (Figure 4.3).



Figure 4.3: The index.jsp providing links to other JSP pages or actions.

You can copy the structure of the following files from CD and place them into your project folder, according to the directory structure before running the application (remove action mappings from struts.xml for which action classes are yet to be created):

- struts.xml
- web.xml
- struts.properties
- ApplicationResources.properties
- ApplicationResources_en.properties
- ApplicationResources_fr.properties

Now we'll be providing some Interceptor examples implementing different Interceptors according to the functionality required. Every example needs the creation of some JSP pages, action classes, and provides action mappings in struts.xml with declaration of Interceptor stacks for the action. Each hyperlink provided in index.jsp is for a different Interceptor examples. In Figure 4.3, you can see the list of interceptors and interceptor stacks implemented in the different examples to be created now.

Interceptor Example 1

Now, start writing code to make the first hyperlink to work, i.e. Interceptor Example 1. This example basically implements alias Interceptor and default Interceptor stack basicStack. The whole process of making this example functional involves the following:

- Creating of JSP pages - alias.jsp, and success.jsp
- Creating action class - AliasAction.java
- Configuring newly created action AliasAction in struts.xml file.

JSP Pages

The alias.jsp page provides an HTML form with a textfield name and a password field pwd. The form submission invokes an action referred as aliasing which is provided here as the action attribute of

<s:form> tag. The rest of processing is done by the associated action class AliasAction, which is the next component to be created.

Here's the code, given in Listing 4.27, for alias.jsp (you can find alias.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.27: alias.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html><head>
    <title><s:text name="app.title"/></title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<table align="center" width="300">
<tr>
<td colspan="2"><s:actionerror/></td>
</tr>
<tr><td align="center" colspan="2">Enter Login ID and Password</td></tr>
<tr><td align="center">
        <s:form action="aliasing" method="post">
            <s:textfield name="uname" key="app.loginid"/>
            <s:password name="pwd" key="app.password"/>
            <s:submit value="Enter"/>
        </s:form>
    </td>
</tr>
<tr><td align="center" colspan="2">
<a href="index.jsp">B a c k</a></td>
</tr>
</table>
</body>
</html>
```

The output is shown in Figure 4.4.

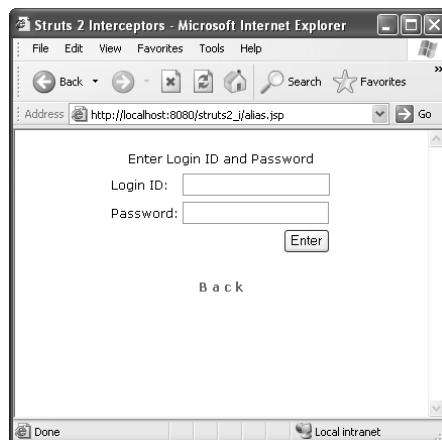


Figure 4.4: The alias.jsp with two input fields.

When the action AliasAction (yet to be created) returns SUCCESS as result code, the JSP page, success.jsp, is shown to the user, which simply displays a message and link to index.jsp page.

Here's Listing 4.28 providing the code for success.jsp (you can find success.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.28: success.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head><title><s:text name="app.title"/></title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <div align="center">
            <h1><s:text name="app.success"/></h1>
            The action has returned SUCCESS as result code.<br><br>
            <a href="index.jsp">Back to Index</a>
        </div>
    </body>
</html>
```

The output is shown in Figure 4.5.

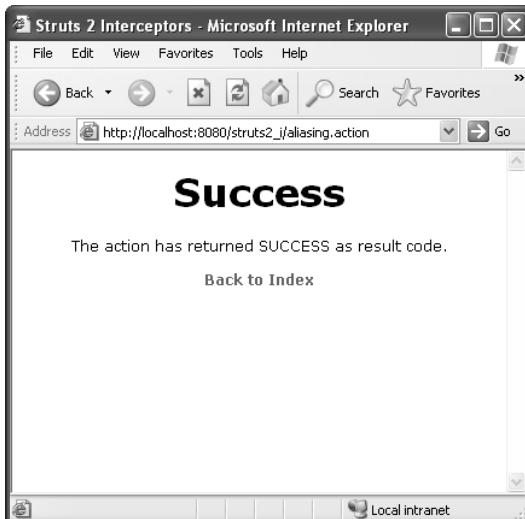


Figure 4.5: Showing the output of success.jsp.

For other result codes, like ERROR and INPUT, the result page declares the alias.jsp page itself.

Action—AliasAction

After creating these JSP pages (alias.jsp and success.jsp), write the code for AliasAction.java which is an action class and will be used here to process the data entered through alias.jsp page. This action class has two fields, username, and password with setter and getter method for them. In addition, like all action classes, an String execute() method is defined here,

which returns SUCCESS or ERROR as result code. A mismatch in loginid and password results in addition of action error using addActionError() method and returns the ERROR as result code.

Here's the code, given in Listing 4.29, for AliasAction action class (you can find AliasAction.java file in Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action folder in CD):

Listing 4.29: AliasAction.java

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class AliasAction extends ActionSupport {
    private String loginid;
    private String password;

    public String getLoginid() {
        return loginid;
    }
    public void setLoginid(String loginid) {
        this.loginid = loginid;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    public String execute() throws Exception {
        System.out.println("Login Id: "+getLoginid());
        System.out.println("Password: "+getPassword());
        if(loginid.equals(password))
            return SUCCESS;
        else{
            this.addActionError(getText("app.invalid"));
            return ERROR;
        }
    }
}
```

Before the execute() method of action is executed, the setter methods for its fields are invoked. The request is searched for the parameter with the same name as that of the field. For example, the field username is set using the getUsername() method, which takes the value of parameter with name username as its argument. In case, there is no matching parameter, the field is set to null.

Compile the AliasAction.java to AliasAction.class and get it into WEB-INF/classes/com/kogent/action folder.

Configuring Action and Interceptors

The HTML form designed in alias.jsp page has fields, 'uname' and 'pwd', while the action class AliasAction has fields, 'username' and 'password'. To set fields, username and password, with the values of parameter, uname and pwd, respectively, we'll have to implement the alias Interceptor. This Interceptor looks for the action parameter for the alias map. The alias map helps in finding the alias

name for some action fields and the field is set with the value of the parameter with the matching alias name. The default action parameter for alias map is aliases.

Here's Listing 4.30 showing the action mapping provided in struts.xml for this action (you can find struts.xml file in Code\Chapter 4\struts2_i\WEB-INF\classes folder in CD):

Listing 4.30: struts.xml with action mapping for action aliasing

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">

<action name="aliasing" class="com.kogent.action.AliasAction">
    <param name="aliases">#{ 'uname': 'loginid', 'pwd' : 'password' }</param>

    <interceptor-ref name="alias"/>
    <interceptor-ref name="basicStack"/>

    <result name="success">/success.jsp</result>
    <result name="error">/alias.jsp</result>
    <result name="input">/alias.jsp</result>
</action>

</package>
</struts>

```

The action parameter, aliases, defines two name:alias pairs. The alias Interceptor accesses the alias map, and it is used before the param Interceptor in the stack.

The class property, loginid, is treated as if it is named uname, and similarly, password is treated as pwd. Any reference to alias name in the action class takes the value of the corresponding parameter. That is how the alias Interceptor helps in setting a class property value with the value of an input parameter, even if the name of action class property and input parameter are different. This case of providing alias map by setting action parameter aliases helps in setting action class properties by the value of another action class property, specially in action chaining. For example, assume that action1 has a getter method `getFirstname()` and action2 has a setter method `setName()`. In action chaining from action1 to action2, we can set the name property of action2 with the value of `firstname` property of action1. The only formality is to add alias Interceptor in the Interceptor stack of action2 and define an action parameter for action2 as shown here:

```

<param name="aliases">#{ 'firstname' : 'name' }</param>

```

Working of Example

After designing JSP pages, compiling action class, and providing action mapping in struts.xml, follow these steps to execute the example:

- ❑ Click on 'Interceptor Example 1' hyperlink in index.jsp page shown Figure 4.3. This brings you to the alias.jsp page shown in Figure 4.4.

- ❑ Enter your ‘Login Id’ and ‘Password’ followed by clicking on the ‘Enter’ button. If the string entered in Login id and Password field are same, you will see success.jsp page (shown in Figure 4.5) otherwise alias.jsp page is shown with an error message.

In the path of action execution, the alias Interceptor intercepts the request and its `setAliasKey()` method is called to set the value of field `aliasesKey` (default is `aliases`). This alias map is further accessed to know the alias name when the parameters are being set by param Interceptor.

Interceptor Example 2

This example implements Interceptors, like exception, prepare, debugging, model-driven, params, conversionError, and workflow. Write the following code files and place them in your application folder according to the directory structure, shown in Figure 4.2:

- ❑ model.jsp
- ❑ student_info.jsp
- ❑ exception.jsp
- ❑ ModelAction.java
- ❑ Student.java

These JSP and Java classes are discussed along with their code in the following headings.

JSP Pages

This example uses three JSP pages—model.jsp to provide input to the action, student_info.jsp to display the student information from the current value stack, and exception.jsp page to display a message intimating occurrence of some exception during the execution of action.

The model.jsp page uses Struts 2 tags to design a form with five input fields. These fields are ‘Roll No.’, ‘Name’, ‘Password’, ‘Course’, and ‘City’.

Here’s the code, given in Listing 4.31, showing the `action` attribute of the `<s:form>` tag and providing the code of model.jsp page (the action with this name is to be added to `struts.xml`) (you can find model.jsp file in `Code\Chapter 4\struts2_i` folder in CD):

Listing 4.31: model.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title><s:text name="app.title"/></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <table align="center" width="300">
            <tr><td colspan="2"><s:actionerror/></td></tr>
            <tr><td align="center" colspan="2">Enter Student Details</td></tr>
            <tr><td align="center">
                <s:form action="modelAction" method="post">
                    <s:textfield name="rollno" key="app.rollno"/>
                    <s:textfield name="name" key="app.name"/>
                    <s:password name="password" key="app.password"/>
                    <s:textfield name="course" key="app.course"/>
                    <s:textfield name="city" key="app.city"/>
                </s:form>
            </td></tr>
        </table>
    </body>
</html>
```

```

<s:submit value="Submit" align="center"/>
</s:form>
</td>
</tr>
<tr><td align="center" colspan="2">
    <a href="index.jsp">B a c k</a>
</td></tr>
</table>
</body>
</html>

```

The keys used here to create the labels of each text box is picked up for the ApplicationResources.properties file. The output of the model.jsp page is shown in Figure 4.6.

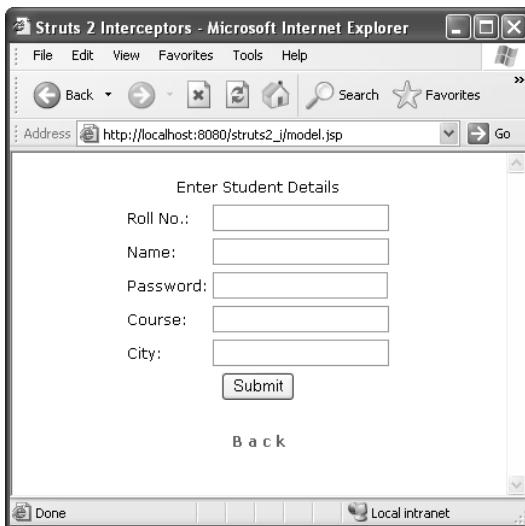


Figure 4.6: The model.jsp showing a student form.

The student_info.jsp page will be displayed when the action processing data, submitted from model.jsp, returns a SUCCESS. The student_info.jsp page simply displays different field values set in the value stack using <s:property> tag.

Here's the code, given in Listing 4.32, for student_info.jsp page (you can find student_info.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.32: student_info.jsp

```

<%@ page language="java" pageEncoding="ISO-8859-1" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
    <h2>Student Information</h2>
    Enrollment No. : <s:property value="rollno"/><br><br>

```

```
Student's Name : <s:property value="name"/><br><br>
Student's Password : <s:property value="password"/><br><br>
Course : <s:property value="course"/><br><br>
City : <s:property value="city"/><br><br>
<a href="index.jsp">Back to Index</a>
</body>
</html>
```

The next page to be designed for this example is exception.jsp. As its name suggests, this page is to be displayed by the framework when some exception occurs. This needs some additional configuration yet to be described. The exception.jsp tells the user about the exception and some hyperlinks to navigate to another location. This approach enables us to avoid our user to see stack trace of the exception, which is irritating to the user. Instead of getting a stack trace describing the exception occurred, the user now gets a JSP page describing this exception.

Here's the code, given in Listing 4.33, for exception.jsp page (you can find exception.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.33: exception.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head><title>Exception. . . .</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <div align="center">
            <h2>Exception</h2><br>
            Some Exception has caused you to see this page.<br><br>
            <a href="index.jsp">Back to Index</a>
        </div>
    </body>
</html>
```

Model-Driven Action—**ModelAction**

This example mainly focuses on the implementation of the model-driven and prepares Interceptors. This Interceptor works only when the action implements com.opensymphony.xwork2.ModelDriven and com.opensymphony.xwork2.Preparable interfaces, respectively. The action class created here is ModelAction.

This class implements ModelDriven and Preparable interfaces. The implementation of ModelDriven interface exposes Object getModel() method in the action class which returns the object of the separate model class and makes it available in the action for processing. This approach enables the implementation of form bean like approach in which an ActionForm object is separately created to hold and validate the input data. In this way, we can use a POJO as a form bean class, which can be injection for our action using getModel() method.

So, let's create our POJO form first which is a JavaBean having a set of fields and getter/setter methods. The fields are 'rollno', 'name', 'password', 'course', and 'city'.

Here's the code, given in Listing 4.34, for Student class (you can find Student.java file in Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action folder in CD):

Listing 4.34: Student.java

```
package com.kogent.action;
public class Student{
    private int rollno;
    private String name;
    private String password;
    private String course;
    private String city;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCourse() {
        return course;
    }
    public void setCourse(String course) {
        this.course = course;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public int getRollno() {
        return rollno;
    }
    public void setRollno(int rollno) {
        this.rollno = rollno;
    }
}
```

The ModelAction action class declares a member object of Student type, which represents the model for this action here. The getModel() method of ModelDriven interface is defined to return the model object, which happens to be student here.

However, before the model object is returned and populated with data, the object must be created and initialized first. For this purpose, we have implemented the Preparable interface that provides the prepare() method. We can use this method to prepare our model object before the invocation of getModel() method.

Here's the code, given in Listing 4.35, for ModelAction action class (you can find ModelAction.java file in Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action folder in CD):

Listing 4.35: ModelAction.java

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.opensymphony.xwork2.Preparable;

public class ModelAction extends ActionSupport
    implements ModelDriven, Preparable{

    private Student student;

    public Object getModel(){
        return student;
    }

    public void prepare(){
        student=new Student();
    }

    public String execute() throws Exception {
        if(student.getPassword().length()>=6)
            return SUCCESS;
        else{
            this.addActionError(getText("app.invalid.password.length"));
            return ERROR;
        }
    }

    public void validate() {
        if ((student.getRollno()== 0) ) {
            this.addFieldError("rollno", getText("app.rollno.blank"));
        }
        if ( (student.getName() == null) || (student.getName().length() == 0) ) {
            this.addFieldError("name", getText("app.name.blank"));
        }
        if ( (student.getPassword() == null) ||
            (student.getPassword().length() == 0) ) {
            this.addFieldError("password", getText("app.password.blank"));
        }
    }
}
```

The Student class does not have any validate method, so the data is still validated in the action class itself. But the real logic of executing different methods of the action lies in the implementation of various Interceptors configured for the action, while providing action mapping.

Configuring Action and Interceptors

In the Interceptor Example 2 being created here, we are implementing the following Interceptors describing their objective and way of working:

- exception
- prepare
- debugging
- model-driven
- params
- conversionError
- workflow

Add a new action mapping in `struts.xml` file to enable the invocation of `ModelAction` action class. The name of the action used here is `modelAction` which matches with the value action attribute of `<s:form>` tag used in Listing 4.31 for `model.jsp`. In addition, an `<exception-mapping>` has also been provided under `<global-exception-mappings>`.

Here's Listing 4.36 showing the new action mapping and exception mapping added:

Listing 4.36: Action mapping for `ModelAction` action class in `struts.xml`

```

<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">

    <global-exception-mappings>
        <exception-mapping exception="java.lang.Exception" result="exception"/>
    </global-exception-mappings>

    <action name="aliasing" class="com.kogent.action.AliasAction">
        .
        .
        .
        .
    </action>

    <action name="modelAction" class="com.kogent.action.ModelAction">

        <interceptor-ref name="exception"/>
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="debugging"/>
        <interceptor-ref name="model-driven"/>
        <interceptor-ref name="params"/>
        <interceptor-ref name="conversionError"/>
        <interceptor-ref name="workflow"/>

        <result name="success">/student_info.jsp</result>
        <result name="error">/model.jsp</result>
        <result name="exception">/exception.jsp</result>
        <result name="input">/model.jsp</result>
    </action>

</package>
</struts>

```

We have declared a set of results for this action mapping having names, success, error, exception, and input. The Interceptors configured here are executed in the order they are placed here (top to bottom) in the Interceptor stack. Now let's discuss what is being done by each Interceptor.

The *exception* Interceptor

This Interceptor is made the first Interceptor in the stack ensuring that it has full access to any exception caused by action class and other Interceptors also. With the use of this Interceptor, the exception mapping provided enables action to return a result code exception, instead of throwing unexpected exception. The exception mapping provided here maps all exceptions of type `Exception` (their subclasses too) with the result code exception. The action mapping also includes a result with the name exception, which causes rendering of `exception.jsp` page.

The *prepare* Interceptor

This Interceptor invokes the `prepare()` method of the action, if it implements `Preparable` interface. Placing this Interceptor before Model-Driven Interceptor makes sure that the `prepare()` method will be executed earlier to `getModel()` method.

The *debugging* Interceptor

This Interceptor provides different debugging screens according to the value of the `debug` parameter passed with the request to the action. This Interceptor is activated only when the `devMode` is enabled in `struts.properties` file. This can be done by adding the text, `struts.devMode=true` in `struts.properties`. The parameter `debug=xml` causes the dumping of parameters, context, session and value stack as an XML document.

The *model-driven* Interceptor

The model-driven Interceptor adds the model of action on the value stack given that the action implements `ModelDriven` interface. This invokes the `getModel()` method of action. If the parameters are to be applied to the model, the model-driven Interceptor must come before params and static-params Interceptors in the Interceptor stack.

The *params* Interceptor

The params Interceptor gets all parameters from `ActionContext` using its `getParameters()` methods and sets them on the value stack using the `ValueStack.setValue(String, Object)` method.

The *conversionError* Interceptor

This Interceptor adds `conversionErrors` map as the field errors. The condition for this Interceptor to be functional is that the action should be implementing the `ValidationAware` interface. The original value of the fields are saved for the subsequent requests.

The *workflow* Interceptor

This Interceptor invokes the `validate()` method, prior to `execute()` method, and prohibits its execution, if some error is added in `validate()` method. This Interceptor works with the following two functions:

- ❑ If the action implements `Validateable` interface, it invokes the `validate()` method of the action.
- ❑ If the action implements `ValidationAware` interface, and if the `hasError()` method returns true, the Interceptor stops the execution of action and returns `Action.INPUT`.

Working of Example

After this discussion about the action and Interceptor configuration for this example, we can run this example by clicking over ‘Interceptor Example 2’ shown in Figure 4.3 and created in Listing 4.26. The page, model.jsp, will appear, as shown in Figure 4.6. If we click on the ‘Submit’ button after leaving fields blank, the workflow Interceptors invokes the validate() method of ModelAction class. The validate() method adds different field errors and the Interceptor returns INPUT as result code. This results in showing of model.jsp with error displayed with the corresponding fields, as shown in Figure 4.7.

The screenshot shows a Microsoft Internet Explorer window titled 'Struts 2 Interceptors - Microsoft Internet Explorer'. The address bar displays 'http://localhost:8080/struts2_j/modelAction.action'. The main content area contains a form titled 'Enter Student Details'. The form has five input fields: 'Roll No.' (containing '0'), 'Name' (empty), 'Password' (empty), 'Course' (empty), and 'City' (empty). Above each field, an error message is displayed: 'Roll No. is Required' for 'Roll No.', 'Name is Required' for 'Name', 'Password is Required' for 'Password', and 'Course is Required' for 'Course'. A 'Submit' button is at the bottom of the form. The status bar at the bottom of the browser window shows 'Done' and 'Local intranet'.

Figure 4.7: The model.jsp with field errors set by validate() method.

The data entered into input fields are converted according to the types of fields defined in model class, which has to populate with this data. For example, the ‘Roll No.’ field of Student class is of type int. When some data is entered in ‘Roll No.’ field, which cannot be converted into int,, an entry is added into conversionErrors map of ActionContext. These errors are added as field errors by the conversionError Interceptor. You can see the field errors set for ‘Roll No.’ field in Figure 4.8.

The screenshot shows a Microsoft Internet Explorer window titled 'Struts 2 Interceptors - Microsoft Internet Explorer'. The address bar displays 'http://localhost:8080/struts2_j/modelAction.action'. The main content area contains a form titled 'Enter Student Details'. The form has five input fields: 'Roll No.' (containing '101ad'), 'Name' (containing 'Prakash'), 'Password' (empty), 'Course' (containing 'MCA'), and 'City' (containing 'Delhi'). Above the 'Roll No.' field, an error message is displayed: 'Invalid field value for field "rollno". Roll No. is Required'. The 'Submit' button is at the bottom of the form. The status bar at the bottom of the browser window shows 'B a c k' and 'Local intranet'.

Figure 4.8: The model.jsp showing conversion error for Roll No. field.

If every thing goes fine and the `execute()` method returns `SUCCESS`, the `student_success.jsp` page is shown with data entered in `model.jsp` page. Figure 4.9 shows the output of `student_info.jsp` page.

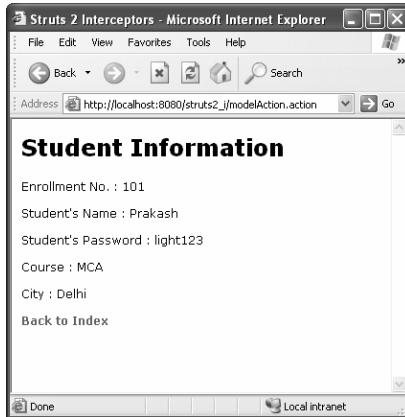


Figure 4.9: The student_info.jsp showing student data.

The only page, which has not yet been executed, is `exception.jsp`. This page will be shown only when some exception occurs during the processing of Interceptors and action. Let's generate an exception in this example.

When the form from `model.jsp` is submitted, the sequence of method invocation is `prepare()`, `getModel()`, `validate()` and then `execute()`. Assume that the `prepare()` method is never executed. This will result in the setting of model student as null. Any method invocation over this object will cause a `NullPointerException` as the object is referencing to null. We can get this exception by removing the Prepare Interceptor from the Interceptor stack defined for this action, making sure that the `prepare()` method is not executed and the model remains null. This Exception Interceptor checks for any exception occurred during the processing, which is pushed on the stack. A `<exception-mapping>` is searched for the exception caused and the corresponding result code is returned. The result code returned here is `exception`, which results in the display of the `exception.jsp` page, as shown in Figure 4.10.



Figure 4.10: The exception.jsp showing message.

Finally to check how the debugging Interceptor behaves, change the action attribute of `<s:form>` tag in `model.jsp` to `/modelAction.action?debug=xml` and again submit the form from `model.jsp` filling it with some data, say '101', 'Prakash', 'kogentindia', 'MCA' and 'Delhi'. You'll get an XML output displayed in your browser, as shown in Listing 4.37.

Listing 4.37: parameters, session, request, context and value stack dumped in XML format

```

- <debug>
- <parameters>
- <course>
<arrayitem>MCA</arrayitem>
</course>
- <rollno>
<arrayitem>101</arrayitem>
</rollno>
- <name>
<arrayitem>Prakash</arrayitem>
</name>
- <password>
<arrayitem>kogentindia</arrayitem>
</password>
- <city>
<arrayitem>Delhi</arrayitem>
</city>
</parameters>
- <context>
<attr />
<report.conversion.errors>false</report.conversion.errors>
- <struts.actionMapping>
<class>
class org.apache.struts2.dispatcher.mapper.ActionMapping</class>
<name>modelAction</name>
<namespace>/</namespace>
</struts.actionMapping>
</context>
<request />
<session />
- <valueStack>
- <value>
<city>Delhi</city>
<class>class com.kogent.action.Student</class>
<course>MCA</course>
<name>Prakash</name>
<password>kogentindia</password>
<rollno>101</rollno>
</value>
- <value>
<actionErrors />
<actionMessages />
<class>class com.kogent.action.ModelAction</class>
<errorMessages />
<errors />
<fieldErrors />
- <locale>
<ISO3Country>USA</ISO3Country>

```

```
<ISO3Language>eng</ISO3Language>
<class>class java.util.Locale</class>
<country>US</country>
<displayCountry>United States</displayCountry>
<displayLanguage>English</displayLanguage>
<displayName>English (United States)</displayName>
<displayVariant />
<language>en</language>
<variant />
</locale>
- <model>
  <city>Delhi</city>
  <class>class com.kogent.action.Student</class>
  <course>MCA</course>
  <name>Prakash</name>
  <password>kogentindia</password>
  <rollno>101</rollno>
</model>
</value>
- <value>
  <class>class com.opensymphony.xwork2.DefaultTextProvider</class>
</value>
</valueStack>
</debug>
```

This helps in debugging, as it provides information about parameters, context, request, session, and value stack.

Interceptor Example 3

The third example is to describe the working of servlet-config and scoped-model-driven Interceptor. The servlet-config Interceptor sets action properties according to the different interfaces implemented by the action. Different interfaces, which are supported by this Interceptor, are as follows:

- ServletContextAware
- ServletRequestAware
- ServletResponseAware
- ParameterAware
- RequestAware
- SessionAware
- ApplicationAware

The Scoped-Model-Driven Interceptor works similar to Model-Driven, but the model is taken from the specified scope and again set into the same scope after being processed by the action.

The code files created for this example to work with are listed here:

- servlet_success.jsp
- scoped_student_info.jsp
- ServletAction.java
- ScopedModelAction.java

JSP Pages

The `servlet_success.jsp` page displays data from different scopes, like request and session. The details to be displayed are to be set by some action class, which is aware of Servlet parameters like request and session. We can access application, request, and session scoped attributes by using `<s:property />` tag using the following syntax:

```
<s:property value="#someScope.attributeName"/>
or
<s:property value="#someScope['attributeName']"/>
```

Here's the code, given in Listing 4.38 for `servlet_success.jsp` page (you can find `servlet_success.jsp` file in `Code\Chapter 4\struts2_i` folder in CD):

Listing 4.38: `servlet_success.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2>Request and Session Attributes..</h2>
Printing <b>Request</b> Attributes..
<br><br>The action class used : <b><s:property value="#request.action"/></b>
<br><br>The interceptor used: <b><s:property value="#request.interceptor"/> </b>

<br><br><br>Printing <b>Session</b> Attributes..
<br><br>User Name : <b><s:property value="#session.username"/></b>
<br><br>Company : <b><s:property value="#session.company"/></b>

<br><br><br>Printing Student Details from <b>Session</b> scope. <br><br>
Enrollment No. : <s:property value="#session.student.rollno"/><br><br>
Student's Name : <s:property value="#session.student.name"/><br><br>
Student's          Password           : <s:property
value="#session.student.password"/><br><br>
Course : <s:property value="#session.student.course"/><br><br>
City : <s:property value="#session.student.city"/><br><br>

<br><a href="scopedmodelAction.action">Invoke a ScopedModelDriven Action</a>
<br><br>
<a href="index.jsp">Back to Index</a>
</body>
</html>
```

The page also displays values of different properties of a `Student` class object stored in session scope. This is obtained again by using the `<s:property />` tag and setting the value to `#scope.attributeName.property`.

The `scoped_student_info.jsp` displays the same `Student` detail as displayed in the `servlet_success.jsp`. But the page will be displayed through another action, which is capable of

processing a session scoped Student object. The process logic of different actions used in this example is described next.

Here's the code, given in Listing 4.39, for scoped_student_info.jsp page (you can find scoped_student_info.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.39: scoped_student_info.jsp

```
<%@ page language="java" import="com.kogent.action.Student"
    pageEncoding="ISO-8859-1" %>

<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
    <link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
<br>Printing Student Details from <b>Session</b> scope. <br><br>
Enrollment No. : <s:property value="#session.student.rollno"/><br><br>
Student's Name : <s:property value="#session.student.name"/><br><br>
Student's Password : <s:property value="#session.student.password"/><br><br>
Course : <s:property value="#session.student.course"/><br><br>
City : <s:property value="#session.student.city"/><br><br>
<a href="index.jsp">Back to Index</a>
</body>
</html>
```

Actions – ServletAction and ScopedModelAction

One of the two actions created for this example is `ServletAction`. This action implements the `ServletRequestAware`, and `SessionAware` interface giving this action access to an object of `HttpServletRequest` and a session Map, which can store session scoped objects. The methods exposed by these interfaces are executed by the `Servlet-Config` Interceptor.

Here's the code, given in Listing 4.40, for `ServletAction.java` (you can find `ServletAction.java` file in Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action folder in CD):

Listing 4.40: ServletAction.java

```
package com.kogent.action;

import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts2.interceptor.ServletRequestAware;
import org.apache.struts2.interceptor.SessionAware;

import com.opensymphony.xwork2.ActionSupport;

public class ServletAction extends ActionSupport
    implements ServletRequestAware, SessionAware{

    HttpServletRequest request;
    Map<String, Object> session_map;
```

```

public void setServletRequest(HttpServletRequest request) {
    this.request=request;
}
public void setSession(Map map) {
    this.session_map=map;
}
public String execute() throws Exception {

    //Setting request attribute.
    request.setAttribute("interceptor", "ServletConfigInterceptor");
    request.setAttribute("action", "ServletAction");

    Student student=new Student();
    student.setRollno(101);
    student.setName("Steve");
    student.setPassword("steve123");
    student.setCourse("B.Tech");
    student.setCity("London");

    //Setting session attributes.
    session_map.put("username", "Prakash");
    session_map.put("company", "Kogent Solutions Inc.");
    session_map.put("student", student);

    return SUCCESS;
}

}

```

Another action used in this example is `ScopedModelAction`, which implements the `com.opensymphony.xwork2.interceptor.ScopedModelDriven` interface. The different methods of this interface, which needs to be implemented by this action class, are `Object getModel()`, `String getScopeKey()`, `void setModel(Object obj)`, and `void setScopeKey(String key)`. This action injects the model to be processed by the action, but this time the model is not populated by the input parameters, if any. Rather the model is picked from the scope defined, like request and session. The Interceptor Scoped-Model-Driven is responsible for the execution of different methods implemented from `ScopedModelDriven` interface. The scope, key, and class of the scoped model is defined where the Interceptors are configured.

Here's the code, given in Listing 4.41, for `ScopedModelAction` class (you can find `ScopedModelAction.java` file in `Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action` folder in CD):

Listing 4.41: `ScopedModelAction.java`

```

package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.interceptor.ScopedModelDriven;

public class ScopedModelAction extends ActionSupport
    implements ScopedModelDriven{

    private Student student;

```

```
private String key;

public Object getModel(){
    System.out.println("getting model.");
    return student;
}

public String getScopeKey() {
    System.out.println("getting key.");
    return key;
}
public void setModel(Object obj) {
    System.out.println("setting model: "+obj.toString());
    this.student=(Student)obj;
}
public void setScopeKey(String key) {
    System.out.println("setting key: "+key);
    this.key=key;
}
public String execute() throws Exception {
    System.out.println("executing");
    student.setName(student.getName().toUpperCase());
    student.setPassword(student.getPassword().toUpperCase());
    student.setCourse(student.getCourse().toUpperCase());
    student.setCity(student.getCity().toUpperCase());
    return SUCCESS;
}
}
```

This action class obtains the model from the scope defined and just changes its different properties to upper case, showing how the scoped model can be accessed and processed from the action.

Configuring Actions and Interceptors

The invocation of these two actions needs addition to two new action mappings in your struts.xml file. The first action mapping provided is with the name `servletAction` and class `ServletAction`. The only Interceptor required here to set the Servlet parameter in the action is `servlet-config` and the only result declared here takes you to `servlet_success.jsp`.

The next action mapping is for our second action. The action name this time is `scopedmodelAction` and the class defined is `ScopedModelAction`. We have configured different Interceptors here, but the most important one is `scoped-model-driven`.

A set of three parameters is defined here for scoped-model-driven Interceptor. These are `scope`, `name`, and `className`. The scoped-Model-Driven Interceptor, here, will search for an object of `Student` class stored in the session with the name `student` and inject it in the action to be processed.

Here's Listing 4.42 showing the action mapping and adds to your copy of `struts.xml` file:

Listing 4.42: Action mapping for `ServletAction` and `ScopedModelAction` actions

```
<action name="servletAction" class="com.kogent.action.ServletAction">
    <interceptor-ref name="servlet-config"/>
    <result name="success">/servlet_success.jsp</result>
</action>
```

```

<action name="scopedmodelAction"
    class="com.kogent.action.ScopedModelAction">
    <interceptor-ref name="prepare"/>
    <interceptor-ref name="debugging"/>
    <interceptor-ref name="scoped-model-driven">
        <param name="scope">session</param>
        <param name="name">student</param>
        <param name="className">com.kogent.action.Student</param>
    </interceptor-ref>
    <interceptor-ref name="params"/>
    <interceptor-ref name="conversionError"/>
    <interceptor-ref name="workflow"/>
    <result name="success">/scoped_student_info.jsp</result>
</action>

```

Working of Example

Now, we are ready to invoke the `ServletAction` action class. This can be invoked by clicking over the hyperlink ‘Interceptor Example 3’, shown in Figure 4.3 and created in Listing 4.26. This executes the `ServletAction` action class, which is configured with `Servlet-Config` Interceptor and set with data in request and session scope.

All request and session scoped data set by this action will be displayed by `servlet_success.jsp` page. The output of `servlet_success.jsp` page is shown in Figure 4.11.

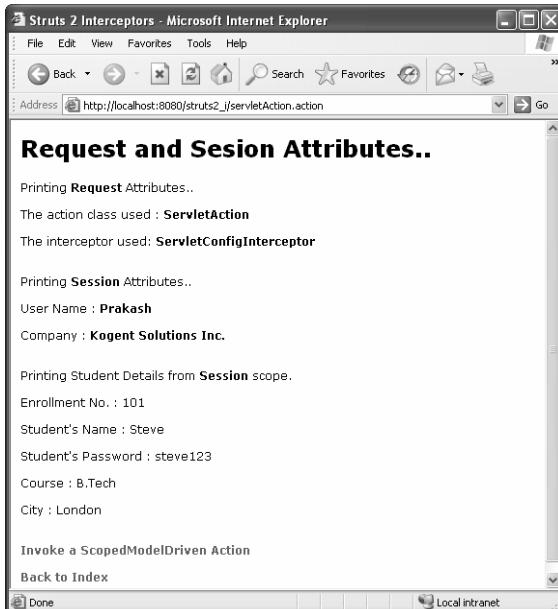


Figure 4.11: The `servlet_success.jsp` showing request and session scoped attributes.

The two request scoped attributes are action and Interceptor, and the two session scoped attributes are username and company. A Student object is put into the session scope by `ServletAction` action. The different property values of this object is also displayed in this page.

The `servlet_success.jsp` page provides a link to invoke the `ScopedModelAction` class. We can click on this hyperlink to get the output, as shown in Figure 4.12.

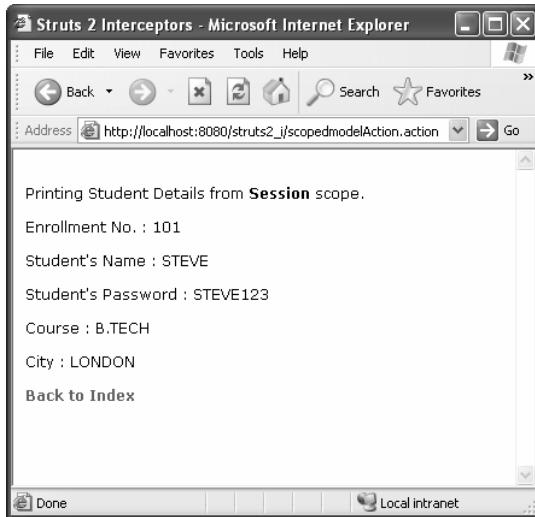


Figure 4.12: The `scoped_student_info.jsp` page showing processed session scoped Student object.

You can see the strings updated to uppercase here. The `ScopedModelAction` class has updated the model after obtaining it from the session. After processing it by the action, the scoped model is again set in the same scope, i.e. session.

Interceptor Example 4

This example shows how `execAndWait` Interceptor is implemented. This is very useful for scenarios where the processing time of some action is somewhat large. This example applies a different approach where a user gets a waiting page, while the action is being processed in the background. The `execAndWait` makes sure that the wait is returned as result code, if the action is not completed. The user is automatically taken to the next view after the execution of action is completed. The following code files are required here:

- `wait.jsp`
- `success.jsp`
- `LongAction.java`

JSP Pages

The `wait.jsp` page is a simple JSP page which is to be displayed to the user, while the action is being processed. This page can show some progress bar or some waiting message to the user. Observe the use of `<META HTTP-EQUIV="Refresh" CONTENT= "4" >` which automatically refreshes the page.

Here's the code, given in Listing 4.43, for `wait.jsp` (you can find `wait.jsp` file in `Code\Chapter 4\struts2_i` folder in CD):

Listing 4.43: wait.jsp

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<META HTTP-EQUIV="Refresh" CONTENT="4">
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<table width="300" align="center">
<tr>
    <td width="150">&nbsp;</td>
    <td bgcolor="#9f1704" height="20">
        <font color="white"><strong>L o a d i n g . . .</strong></font>
    </td>
</tr>
<tr><td>&nbsp;</td></tr><tr>
    <td colspan="2">
        The action is being processed.<br><br>
        If the page did not appear automatically, click
        <s:a href="longAction.action">here</s:a>
    </td></tr>
</table>
</body>
</html>

```

Another JSP page used in this example is `success.jsp`, which has already been created and used in our first example.

Action – LongAction

The action created here is just to make the user wait for some time so that the `execAndWait` Interceptor can show its capabilities. The action class created here is `LongAction`. The thread running here is made to sleep for 5000 milli seconds.

Here's the code, given in Listing 4.44, for `LongAction` class (you can find `LongAction.java` file in `Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action` folder in CD):

Listing 4.44 : LongAction.java

```

package com.kogent.action;
import com.opensymphony.xwork2.Action;

public class LongAction implements Action {
    public String execute() throws Exception
    {
        Thread.sleep(5000);
        return SUCCESS;
    }
}

```

Configuring Action and Interceptors

A new action mapping is to be added in struts.xml to execute this example. The action mapping provided here for LongAction includes a completeStack Interceptor stack and execAndWait Interceptor. Because of its nature, execAndWait Interceptor is placed as the last Interceptor in the stack. This Interceptor works on per session basis and, hence, we cannot run two instances of LongAction action class in a single session. Now add the action mapping, shown in Listing 4.45, in your copy of struts.xml:

Listing 4.45: Action mapping for LongAction

```
<action name="longAction" class="com.kogent.action.LongAction">
    <interceptor-ref name="completeStack"/>
    <interceptor-ref name="execAndWait">
        <param name="delay">2000</param>
        <param name="delaySleepInterval">50</param>
    </interceptor-ref>
    <result name="wait">/wait.jsp</result>
    <result name="success">/success.jsp</result>
</action>
```

The delay parameter is set to 2000 milli seconds, which is the initial delay to wait before the wait page is shown. The delaySleepInterval parameter, set to 50, is the time to wake up and check whether the background process is completed or not.

Working of Example

The LongAction can be invoked by clicking on the ‘Interceptor Example 4’ hyperlink, as shown in Figure 4.3 and created in Listing 4.26. The execution time of the LongAction action is set to 5000 milli seconds by using Thread.sleep(5000) method in its execute() method. The wait page will appear after the delay time set (2000 milli seconds). The output of wait page (wait.jsp), is shown in Figure 4.13.

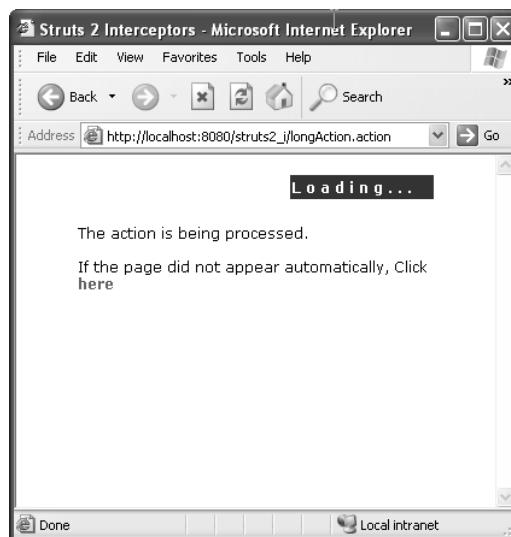


Figure 4.13: The wait.jsp page showing wait message.

The page automatically refreshes and the user is automatically taken to the success.jsp, given that the execution of LongAction action is completed. The output of success.jsp is shown in Figure 4.14.

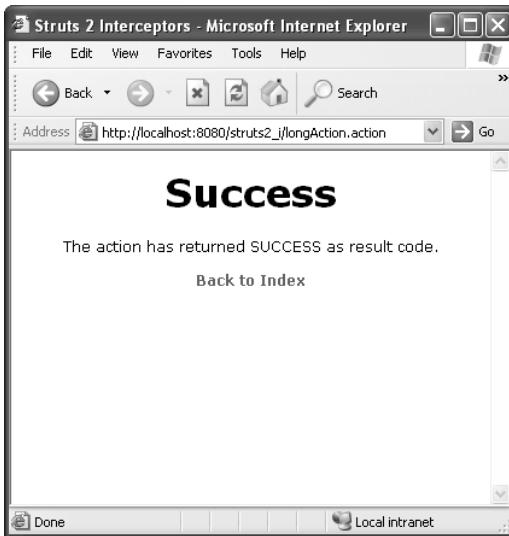


Figure 4.14: The success.jsp shown after the execution of action completed.

Interceptor Example 5

The last example created in this application gives the implementation of validation and Scope Interceptor in the execution of an action. The Validation Interceptor applies standard Validation Framework over the action and validate action against the validation rules defined in some ActionClass_validation.xml file. The following code files are created here in this example:

- login.jsp
- login_success.jsp
- LoginAction.java

JSP Pages

The login.jsp page provides a login form with two input fields to enter username and password. The form is submitted to some action to be processed.

Here's the code, given in Listing 4.46, for login.jsp page (you can find login.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.46: login.jsp

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title><s:text name="app.title"/></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <table align="center" width="300">

```

```
<tr><td colspan="2"><div class="boldred"><s:actionerror/></div></td></tr>
<tr><td align="center" colspan="2">Login!</td></tr>
<tr><td align="center">
<s:form action="login" method="post">
<s:textfield name="username" key="app.username"/>
<s:password name="password" key="app.password"/>
<s:submit value="Login"/>
</s:form>
</td>
</tr>
<tr><td align="center" colspan="2"><a href="index.jsp">
    B a c k</a></td></tr>
</table>
</body>
</html>
```

The output of login.jsp page is shown in Figure 4.15.

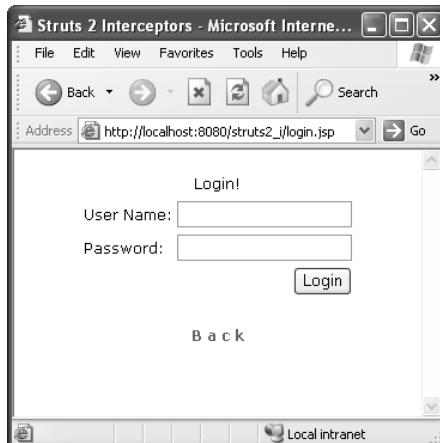


Figure 4.15: The login.jsp showing login form.

Another JSP page, login_success.jsp, created renders a welcome message to the user after its successful login. The page also displays the username and password entered from the session scope.

Here's the code, given in Listing 4.47, for login_success.jsp page (you can find login_success.jsp file in Code\Chapter 4\struts2_i folder in CD):

Listing 4.47: login_success.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1" %>
<%@ taglib prefix="s" uri="/struts-tags" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head><title><s:text name="app.title"/></title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <div align="center">
```

```

<h2>Successfully Logged In!</h2>
<br><br>
User name set in Session : <s:property value="#session['struts.ScopeInterceptor:/:loginusername']"/>
<br>Password set in Session :
<s:property value="#session['struts.ScopeInterceptor:/:loginpassword']"/>

<br><br><br>Welcome
<s:property value="#session['struts.ScopeInterceptor:/:loginusername']"/> !
<br><br><a href="index.jsp">Back to Index</a></div>
</body>
</html>

```

Action - LoginAction

The action class created here is `LoginAction`. The `LoginAction` action class has two input properties, i.e. `username` and `password`, with setter/getter methods. The `LoginAction` action class does not have any validate method implemented here as its properties are to be validated using the Validation Framework, taken care by the validate Interceptor.

Here's the code, given in Listing 4.48, for `LoginAction` action class (you can find `LoginAction.java` file in `Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action` folder in CD):

Listing 4.48: `LoginAction.java`

```

package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport {

    private String username;
    private String password;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    public String execute() throws Exception {
        if(username.equals(password))
            return SUCCESS;
        else{
            this.addActionError(getText("app.invalid"));
            return ERROR;
        }
    }
}

```

The LoginAction returns SUCCESS if the username matches with password, otherwise an action error is added and an ERROR is returned. The input properties, like username and password, are validated for their values through the Validation Framework. This means the validation rules for these fields are to be defined in ActionClass_validation.xml file where ActionClass is the name of the action class to be validated.

Here's the content of LoginAction-validation.xml, given in Listing 4.49, which defines the validation rules for the input fields of LoginAction action class (you can find LoginAction-validation.xml file in Code\Chapter 4\struts2_i\WEB-INF\src\com\kogent\action folder in CD):

Listing 4.49: LoginAction-validation.xml file

```
<!DOCTYPE validators PUBLIC "-//openSymphony Group//xwork validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="username">
        <field-validator type="requiredstring">
            <message>Enter User Name</message>
        </field-validator>
    </field>
    <field name="password">
        <field-validator type="requiredstring">
            <message>Enter Password</message>
        </field-validator></field>
</validators>
```

The action class and associated ActionClass_validation.xml file must be at the same location.

Configuring Action and Interceptors

Now, we need to configure this action class with a set of Interceptors and results. The basic Interceptors, which need discussion here, are validation and scope. The Interceptor stack, defined here for this action, includes these Interceptors. The Validation Interceptor works in conjunction with the Workflow Interceptor. Add a new action mapping for LoginAction, which is shown in Listing 4.50, with all Interceptors and results:

Listing 4.50: Configuring Interceptors for LoginAction

```
<action name="login" class="com.kogent.action.LoginAction">
    <interceptor-ref name="basicStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
    <interceptor-ref name="scope">
        <param name="session">username, password</param>
        <param name="key">ACTION</param>
        <param name="type">start</param>
        <param name="autoCreatesession">true</param>
    </interceptor-ref>
    <result name="success">/login_success.jsp</result>
    <result name="error">/login.jsp</result>
    <result name="input">/login.jsp</result>
</action>
```

The Scope Interceptor is used to set some action properties in application or session scope to make it consistent throughout the application. The session and application attributes are keyed after the action's class, action's name or any other given key. The `LoginAction` class properties to be set in session is provided using `<param name="session">username, password</param>`.

Similarly, parameter key is set to ACTION, which creates a unique key prefix for the attribute to be set in session scope here with actions namespace and action name.

Working of Example

We can access the `login.jsp` page by clicking over the 'Interceptor Example 5' hyperlink, as shown in Figure 4.3 and created in Listing 4.26. The output of the `login.jsp` page is already shown in Figure 4.15. Click over 'Login' button, as shown in Figure 4.15, leaving both fields blank to see the field errors as shown in Figure 4.16. These field errors have been added by the Validation Interceptor according to the validation rules defined in `LoginAction_validation.xml` for different properties of `LoginAction`.



Figure 4.16: The login.jsp page showing field errors set by validation Interceptor.

The error text displayed here is defined in `LoginAction_validation.xml` file. Enter a same string as username and password here to see the output of `login_success.jsp`, which displays a message for successful login and prints the username and password from the session scope. We have not used any session object throughout this example, but the two properties of the `LoginAction` action class are put into session scope by the Scope Interceptor. The output of the `login_success.jsp` is shown in Figure 4.17.



Figure 4.17: The login_success.jsp showing data from session scope.

This chapter described the important and frequently-used Interceptors through five different examples. Each example emphasized a different set of Interceptors in their working. The configuration and execution flow of different Interceptors are fully discussed in these examples. Now, the different Interceptors can be seen for their importance. The chapter fully describes where these Interceptors are required and how they should be implemented. Different modifications in the action class in conjunction have also been described, like implementing different interfaces and defining some methods to make Interceptors active.

In the next chapter, we'll discuss the OGNL expression language and its different capabilities and how it is supported in Struts 2 Framework for interaction with ValueStack.



5

Manipulating Data with OGNL in Struts 2

If you need an immediate solution to:

Creating an Struts 2 application with OGNL

See page:

184

In Depth

We can handle Java objects in our page embedding corresponding Java code with HTML text. However, using Java code while designing web pages, is always irritating for page designers. The two options can be using tags or using some expression language to access different properties of various types of Java objects, like JavaBeans, ArrayList, and Map. An Expression language (EL) is basically developed to help in writing an expression in a very simple and smart way to perform some common tasks. Usually, EL support is included with a particular framework with the intent to make a developer's life easier. The use of any EL language reduces the Java code used in a view page. They also help in eliminating the repetitive code that might have been used. Expression language will always be preferred over using Java code as they provide easy access to Java components. Different frameworks add support for various expression languages into their APIs. Similarly, Struts 2 supports an expression language, which is known as Object Graph Navigation Language (OGNL). Struts 2 uses OGNL expressions for mapping request parameters to different value object in action instance and to read value objects for displaying them in View layer. Struts 2 implements a new concept of value stack through which the different data objects are available in View. Value stack can be defined as a set of value object which are stored in stack and different action instances are pushed in the value stack. In this chapter, we are going to understand OGNL for all its syntax with different operations and expressions. All the basic features of OGNL have been described with examples. Further, the Struts 2 Framework support for OGNL has been explained in the chapter.

Object Graph Navigation Language (OGNL) can be defined as an expression language to interact with Java objects. This is a very useful binding language for manipulating and retrieving different properties of Java objects. The OGNL has its own syntax which is very simple in structure and, hence, easy to learn and use. It also makes code more readable. The OGNL syntax provides a high-level abstraction for navigating object graphs, which means specifying paths through and to JavaBeans properties, collection indices, etc. instead of accessing them directly using Java code. You can see the following code segment to find the difference between the two Java codes and the corresponding OGNL expression to access a session scoped object (say, student) and obtaining value for property (say name). In Web Frameworks, expression languages have similar goals. They exist to eliminate the repetitive code that you might otherwise write, if you didn't have an EL. For example, without an EL, getting an Student object from the session and then displaying its name on the web page requires a few lines of Java code in a JSP, as can be seen in the following code lines:

```
//Using Java Code - First way
<%
Student student = (Student) session.get("student");
String name = student.getName();
%>
<%= name %>

//Using Java Code - Second way
<%= ((Student) session.get("student")).getName() %>
```

```
//OGNL expression  
#session.student.name
```

Here, we have used three ways to get the name of a student from session. First two are using Java code in JSP, while the last one is using OGNL expression. Though the second way has reduced the code to single line, the code now looks ugly and is hard to read. In addition, the same parts of the original statements are still required, such as casting to Student, and calling getName() method, although you have moved three statements to a single statement, yet the complexity remains the same. The OGNL expression has all keys, variable, and getters same as Java code. But what it lacks is all the Java language overheads.

An expression language for a Web Framework saves you from this kind of complexity. Rather than using the Servlet APIs, cast an object, and then call a getter method, most ELs simplify this down to a much more readable expression similar to #session.list.id. The expression #session.list.id has all the methods and casting. Because these kinds of operations are so common, using an EL makes perfect sense. You can use a much more loosely typed and dynamic language to act as a buffer between you and that complex Java code you would have to write otherwise.

Understanding Object Graph Navigation Language (OGNL)

Object Graph Navigation language (OGNL) is an expression and binding language used for getting and setting the properties of the Java objects. Usually, for setting and getting the value for a particular property, the same expression is used. OGNL acts as a binding language between GUI elements to model objects. One can utilize the OGNL for multiple uses. Here are a few uses of OGNL:

- ❑ OGNL provides a TypeConverter mechanism, which makes transformations easier. TypeConverter mechanism is used to convert values from one type to another. For example, it converts String type to numeric type.
- ❑ OGNL is a data source language, which is used to create a map between table columns and a Swing TableModel.
- ❑ It works as a binding language between Web components and the underlying model objects.
- ❑ OGNL is a replacement to other property getting languages used by Jakarta Commons BeanUtils package or JSTL's expression languages, as it is more expressive in terms of getting and setting the objects properties.

OGNL helps in creating applications similar to Java. In addition, OGNL also supports tasks, such as projection, selection, and lambda expressions. Lambda expression is one, which is used for writing short anonymous expression.

OGNL is not the only kid in the block. In fact, there are more expression languages and scripting languages than we care to count. We will quickly outline a few of the popular ones of these languages that could potentially work with Struts 2:

- ❑ **JSTL's EL**—JSP Standard Tag Library (JSTL) has since been integrated into JSP 2.0 with all support for standard scripting and expression language for JSP. All JSTL tags support Expression Language (EL) with them. For this purpose, we have different versions of JSTL tag library, which support runtime expression. Some Struts 2 users prefer JSTL because it is a standard and commonly in use in the industry. Using JSTL with Struts 2 involves almost no work.
- ❑ **Groovy**—Recently introduced in the IT sector, Groovy uses the Java syntax. The result is a loosely-typed language that offers a lot of functionality with minimal typing. Groovy is an EL that is

integrated into the Web form called Groovlet. In future, Groovy will be the language preferable for Struts 2 applications.

- ❑ **Velocity**—Although Velocity is not exactly an expression language. The syntax used for writing Velocity templates is very similar to OGNL and other ELs. Programmers rarely use the OGNL for creating Velocity-based views.

NOTE

Velocity have been discussed in Appendix B.

Everything in OGNL is centralized around a context that contains one or more JavaBeans from which you evaluate your expressions. One of those JavaBeans is special because it is considered the *root* of the context. An object that is the root is assumed to be the object your expressions are concerned with, if no other object from the context is specified. In the meantime, know that when we give examples in this chapter, we'll indicate what the context contains as well as what object in the context is configured to be the root.

Now, let's explore some key concepts of OGNL which includes its syntax, literals and operators, indexing, method call, variable reference and chaining of OGNL sub-expressions.

Syntax of OGNL

Expressions based on OGNL are very simple and easier to learn and use. This language has become quite rich during the course of its development. But you don't need to worry about the complicated parts of the language because the simple ones have remained simple. For example, to work with the name property, you simply need to use the expression name in OGNL. Sometimes, you need to work with text property; for this you can use the text OGNL expression. An OGNL property is same as a bean property, i.e. this property also has a pair of getter and setter methods or alternatively, a field that defines the property.

The most commonly used unit in OGNL expression language is the navigation chain, which is also termed as chain. The chain consists of the following parts:

- ❑ **Property names**—Property names, such as name and text.
- ❑ **Method calls**—Method calls, such as the hashCode(), which returns the current object's hashcode.
- ❑ **Array indices**—For example, listeners [0], which return the first of the current object's list of listeners.

All OGNL expressions are evaluated in the context of the current object. Chains work by inheriting the output of the last link and then present the same as the current object for the next link. A chain can be extended upto any limit, no chain can be limited. For example, consider the following chain:

```
name.toCharArray() [0].numericvalue.toString()
```

The preceding expression is evaluated in the following ways:

- ❑ First, it extracts the name property of the initial, or root object. This root object is provided to the OGNL through the OGNL context.
- ❑ Then it calls the toCharArray() method of the resulting string.
- ❑ After calling the toCharArray()method, the first character at 0 index is extracted from the resulting array.

- ❑ It then gets the numericValue property from that character. The character is represented as a Character object, and the Character class has a method called getNumericValue().
- ❑ Finally the String is returned after calling the `toString()` on the resulting Integer object.

For extracting the value from the object you can follow the preceding syntax. This example cannot set a value, however, you can perform multiple tasks by using this syntax.

Literals and Operators

In this section, we will study the OGNL expressions in detail. The first element of the expression that we are going to study is Literals and Operates. Accessing data isn't useful if you can't do things to it. Fortunately, OGNL supports the same literals and mathematical operations as found in Java. The following literals are supported by OGNL:

- ❑ **String literals**—These literals are delimited by single or double quotes with full set of character escapes. These are same as in Java, e.g. 'Hello' and "Hello".
- ❑ **Character literals**—Character literals are delimited by single quotes with a full set of character escapes. These are also same as in Java, e.g. 'H'.
- ❑ **Numeric literals**—In addition to all the already described types, the types available in Java, such as int, double, OGNL also supports BigDecimals (specified with 'b' or 'B' suffix) and BigIntegers (specified with 'h' or 'H'), e.g. BigDecimals is shown as 120b and BigIntegers as 120h.
- ❑ **Boolean literals**—It contains either true or false
- ❑ **The null literal**—It contains Null values

String literals in OGNL are enclosed in single or double quotes. These are designed to make the embedding OGNL easier in languages, like XML and JSP. The characters are identified by single quotes and more than one characters enclosed in single quotes are identified as String. To make a String literal that is one character long, you must use double quotes to enclose it. In addition to literals, OGNL lets you use all Java operations, such as addition (+) and division (/). Table 5.1 contains all the expressions that OGNL supports as well as an example of each one in use.

Table 5.1: OGNL operators

Operator	Example
add(+)	1+2 , 'hello' + 'World'
subtract(-)	3-2
multiply(*)	4*5
divide(/)	6/2
modulus(mod)	8 mod 3
increment(++)	++count , count++
decrement(--)	--count , count--
Equality(==)	count ==5
Inequality (! =)	flag != true

Table 5.1: OGNL operators	
Operator	Example
In	orange in {'apple', 'banana', 'orange'}
not in	orange not in {'apple', 'banana', 'orange'}
assignment(=)	count = 5

Indexing

The indexing notation is just a property reference, which acts as computed form of referenced property, instead of a constant one. For example, OGNL takes the `array.length` expression as mentioned here:

```
array ["length"]
```

OGNL expands the concept of indexed properties to include the indexing with arbitrary objects, not just integers. When properties need to be found as the candidates for indexing an object, OGNL searches for methods pattern with some specific signature. These signatures are as follows:

- ❑ `public PropertyType getPropertyName(IndexType index)`—This signature is used to retrieve the property name with the value of `IndexType`.
- ❑ `public void setPropertyName(IndexType index, PropertyType value)`—This signature is used to set the property name with the value of `IndexType`.

You need to make sure that the `PropertyType` and `IndexType` are matching with each other in the set and get methods. If you are asked to give an example of Object Indexed Properties, give with the Servlet API which is commonly used for object indexed properties; the Servlet API Session object uses two methods for getting and setting random attributes:

```
public Object getAttribute(String name)
public void setAttribute(String name, Object value)
```

A sample OGNL expression, which helps you to get and set one of the mentioned attributes is as follows:

```
session.getAttribute("foo")
```

Method Call

In OGNL a method is called little differently from the way it is called in Java. This difference exists because OGNL needs to choose the right method at runtime, even when you need not specify the extra information, except for the required arguments. OGNL always chooses the most appropriate method and whose type matches with the type of arguments supplied. In case, when arguments of two or more methods are matching and they all are equally suitable for the logic, then one of these methods can be randomly picked.

Sometimes, all non-primitive types are matched with a null argument thus resulting in the call of an unexpected method. As methods arguments are separated by commas, you need to ensure that the comma operator is enclosed within parentheses. For example:

```
Method (ensureLoaded (), name)
```

The preceding example is used to call a 2-argument method, while the following syntax is used to call a 1-argument method:

```
Method ((ensureLoaded (), name))
```

Variable References

OGNL-based variables are simple to use and allow you to store the intermediary output for use in future transitions. OGNL-based variables are always global for the complete expressions used in the program. The syntax for writing an OGNL variable is as follows:

```
#var
```

OGNL helps you to store the value of the current object for evaluation which can be used in future. This value is stored in the “this” variable. You can write this variable in the same way as other variables. For example, the following OGNL expression works on the number size listeners:

```
listeners.size() . (#this > 300? 2*this: 3*this)
```

The current value returned by `size()` method here can be accessed using `this` here. This expression gives twice the number if it is more than 300 or thrice of the number otherwise. You can invoke the OGNL with a map that is used for defining the initial value for the variables. You need to define the variable root and context for invoking the variables. Variable root is used for defining the initial value and context is used for defining the map. For defining a value explicitly, you need to follow the following syntax:

```
#var = 89
```

Chaining OGNL Sub-Expressions

Chaining OGNL sub-expression involves using the parenthetical expression after the dot defined in the expression. Due to this, the object at the dot position is termed as the current object. For example, the following expression is evaluated by first navigating through the `headline` and `parent` properties. This evaluation ensures that the `parent` is loaded and `parent` ‘s name is either `get` or `set`:

```
headline.parent.(ensureLoaded(), name)
```

You can use the preceding expression to evaluate the top-level expressions. In this expression, the final output will be the right most element of the expression. In this expression, first the `ensureLoaded()` method is called on the root object, after this `name` property of the root object is returned as an output:

```
ensureLoaded(), name
```

Understanding Basic Expression Language Features

Struts 2 lets you write powerful expressions, but its greatest advantage is in the simplicity of basic features, such as accessing bean properties and calling methods. Let's look at some examples of these basic features so that you can get accustomed to OGNL as a language. After that, we'll examine the intermediate and advanced features offered by OGNL and Struts 2, all of which will help you write cleaner, simpler, and more focused Web applications.

Accessing Bean Properties

By far, the most common expression, which will be used in Struts 2 to access bean properties is when you want to get data from an action or set data on an action. According to the JavaBeans specification, bean properties are a getter method and/or a setter method using a standard format, such as `getXxx()`, `setXxx()`, `isXxx()`, or `hasXxx()`. The `isXxx()` and `hasXxx()` formats are used for Boolean properties and are only provided to make the code more readable. In Struts 2, accessing those properties (either setting or getting data) involves referencing the property as `xxx`. This probably isn't new to you, considering that we are hinting about it throughout the book.

Let's look at a few simple examples in which a property from the root object is accessed. Suppose the context contains one object, `John`, which is also set as the root.

Here's a code snippet where `John` is a class with a few properties:

```
import java.util.List;
import java.util.Set;
import java.util.Map;
public class John {
    public static final String OG_JOHN = "Alen";

    public static John getOgJohn() {
        John og = new John();
        og.setName(OG_JOHN);
        return og;
    }
    private String name;
    private int age;
    private John father;
    private John mother;
    private List children;
    private Map favorites;

    public int avgParentsAge() {
        return ((father.getAge() + mother.getAge()) / 2);
    }
}
```

```
// getters and setters methods for each property field
}
```

Let's start with the simplest properties—`name` and `age`. The expression for accessing the values of these properties is as simple as the property names themselves. The expressions `name` and `age` retrieve the name and age of John, respectively. It is possible (and common) to chain properties together in order to navigate deep into the object graph. For example, the expression `father.age` retrieves John's father's age. Likewise, `father.mother.age` gets the John's grandmother's age (on the father's side). This is equivalent to calling `john.getFather().getMother().getAge()` on the `john` object. Table 5.2 contains some sample Java code snippets and compares them to their OGNL equivalents.

Table 5.2: Java syntax and their equivalent OGNL equivalent

Java code	Equivalent OGNL expressions (John is the root)
<code>john.getName()</code>	<code>name</code>
<code>john.getMother().getName()</code>	<code>mother.name</code>

Accessing the OGNL Context and ActionContext

So far, you have only worked with the root object, John. However, OGNL lets you access any element in its context map, called the `OgnlContext`. Struts 2 also has its own context called the `ActionContext`. Struts 2 and OGNL are integrated with each other in many ways. One of these ways is taking `OgnlContext` and `ActionContext` as the same thing. Many objects are often found in the `ActionContext`. Because these two contexts are the same, you can access those objects using OGNL's standard contextual access notation, the hash sign (#). Let's see how you might get access to some of those objects stored in the context.

Table 5.3 provides a few examples using features of OGNL that you have learned so far.

Table 5.3: Java syntax and their OGNL equivalent

Java code	OGNL expression
<code>ActionContext().getContext().getParameters()</code>	<code>#parameters</code>
<code>ActionContext().getContext().getParameters().size()</code>	<code>#parameters.size</code>
<code>((John) ActionContext.getContext().get("john")).getAge()</code>	<code>#john.age</code>

Many of the items in the `ActionContext` are identified by long and unique keys, e.g. the key that is used to store the `HttpServletRequest` is `org.apache.struts2.interceptor.ServletRequestAware`. Obviously, you don't want to type

that when you're using an otherwise simple expression language. To help you, Struts 2 identifies a few of the items in the context and aliases them with shorter identifiers:

- ❑ **parameters**—A Map that contains all the HttpServletRequest parameters for the current request
- ❑ **request**—A map that contains all the HttpServletRequest attributes for the current request
- ❑ **session**—A map that contains all the HttpSession attributes for the current session
- ❑ **application**—A map that contains all the ServletConfig attributes for the current application
- ❑ **attr**—A special map that searches for attributes from the request, session, and application maps (in that order)

In the next section, you'll learn how to access collections, including `java.util.Map`, so that you can begin to get data from these five elements if you wish to.

Working with Collections

OGNL API provides the support for Java language, which is not provided by the other expression languages. OGNL handles property references, while treating the different objects differently involved in an expression. For example, maps involved in the expression treat all the referenced properties as element lookups or storage. The key used here is the name of the property. Even the lists and arrays treat numeric properties as element lookups and storage. Here, index is replaced by property name. But, the string properties are handled by the lists and arrays in the same way they are handled by ordinary objects. All the other objects are called ordinary objects and use the `get` or `set` method to handle the string properties.

Collections in OGNL

OGNL provides support for three types of collections:

- ❑ **Lists**
- ❑ **Native arrays**
- ❑ **Maps**

Lists

List of objects in the OGNL can be created by enclosing the list of expression in curly braces. Similar to method arguments, list of expressions also avoid the use of the comma operator unless it is enclosed in the parentheses. For example, the expression syntax mentioned below will create an instance of the interface for managing the list of objects. The following syntax mentioned is without the sub-class:

```
name in {null, "untitled"}
```

In this expression, the `name` property will be checked for null or untitled value.

Native Arrays

OGNL provide the support for native arrays similar to Java, such as `int[]` or `Integer[]`. Here's the sample syntax for declaring native arrays:

```
new int[] { 1, 2, 3, 4 }
```

The preceding expression creates an array, which consists of four integers 1, 2, 3, and 4.

Sometimes, you need to create an array with null values and the same can be created by using the following syntax:

```
new int[10]
```

In this expression, an integer array will be created with ten slots and all are initialized to zero.

Maps

Maps are the contexts, which are used for initializing the root values, and they are created using the following syntax:

```
#{"fun" : "fun value", "barbarian" : "barbarian value" }
```

In this syntax, a map will be created and initialized with fun and barbarian values.

Sometimes, you need to select the specific map class. This class can be specified before opening the curly brace like this:

```
#@java.util.LinkedHashMap@{"fun" : "fun value", "barbarian" : "barbarian value" }
```

This expression will create an instance of the class `LinkedHashMap`, and this will ensure that an insertion order of the elements is maintained.

Working with Lists and Arrays

In OGNL, lists and arrays are generally treated as the same. Thus, offset notation that is normally reserved for arrays is also used to access list elements. Table 5.4 demonstrates some simple examples of using array notation to work with lists and arrays.

Table 5.4: Accessing lists and arrays using OGNL expression

Java code	OGNL expression
<code>list.get(0)</code>	<code>list[0]</code>
<code>Array[0]</code>	<code>Array[0]</code>
<code>((John) list.get(0)).getName();</code>	<code>list[0].name</code>
<code>array.length</code>	<code>array.length</code>
<code>list.size()</code>	<code>list.size</code>
<code>list.isEmpty()</code>	<code>list.isEmpty</code>

In addition to accessing the values of lists, we can implement looping over a known set of items. Table 5.5 shows constructing lists in simplified notation.

Table 5.5: Interacting with lists using OGNL expression

Java code	OGNL expression
<pre>List list = new ArrayList(3); list.add(new Integer(1)); list.add(new Integer(3)); list.add(new Integer(5)); return list;</pre>	{1, 3, 5}
<pre>List list = new ArrayList(2); list.add("foo"); list.add("bar"); return list.get(1);</pre>	{"foo", "bar"}[1]

Working with Maps

Maps are similar to lists, but instead of only being able to access elements by a numbered index, you can access elements with any object key. Their syntax for element access, however, is virtually identical. In the situation where primitives are used as keys, OGNL does autoboxing for you—this means it automatically converts a primitive int into an Integer object or a Boolean into a Boolean object.

Table 5.6 demonstrates how you can access maps as well as how OGNL automatically takes care of converting types for you, such as 1 (int) to 1 (Integer).

Table 5.6: Interacting with Map using OGNL expression

Java code	OGNL expression
map.get("foo")	map['foo']
map.get(new Integer(1))	map[1]
John John = (John) map.get("Alen"); return John.getAge();	map['Alen'].age
map.put("foo", "bar");	map['foo'] = 'bar'
map.size()	map.size
map.isEmpty()	map.isEmpty

As with lists, you can create maps dynamically. The syntax for doing so is slightly different—you must place a hash sign (#) before the opening curly brace.

Dynamic maps are especially useful for radio groups and select tags. For example, if you wanted to offer a true/false selection that displays as a Yes/No choice, #{true : 'Yes', false : 'No'} would be

the value for the list attribute. The value for value attribute would evaluate to either true or false. Table 5.7 shows a few examples of creating maps dynamically.

Table 5.7: More interaction with Maps using OGNL expressions

Java code	OGNL expression
<pre>Map map = new HashMap(2); map.put("foo", "bar"); map.put("baz", "whazzit"); return map;</pre>	<code>#{ "foo" : "bar", "baz" : "whazzit" }</code>
<pre>Map map = new HashMap(3); map.put(new Integer(1), "one"); map.put(new Integer(2), "two"); map.put(new Integer(3), "three"); return map;</pre>	<code>#{ 1 : "one", 2 : "two", 3 : "three" }</code>
<pre>Map map = new HashMap(2); map.put(Alen.getName(), Alen.getMother().getName()); map.put(steve.getName(), steve.getMother().getName()); return map;</pre>	<code>#{@Alen.name: Alen.mother.name, #steve.name : #steve.mother.name }</code>
<code>ActionContext.getContext().getParameters().get("id")</code>	<code>#parameters['id']</code>
<pre>String name = John.getName(); Map map = ActionContext.getContext().getSession(); return map.get("John- + name);</pre>	<code>#session["John- " + name]</code>
<code>session.put("steve- Alen", John);</code>	<code>#session['steve-Alen'] = John</code>

Projection in Collections

OGNL has provided a simple way to call the same method or extract the same property from each element in a collection and store that result in a new collection. This whole procedure is called *projection*. For example, the following expression will create a list of all the listener's delegates:

```
listeners. {delegate}
```

During a projection, the `#this` variable refers to the current element of the iteration:

```
objects.{ #this instanceof String ? #this: #this.toString()}
```

The preceding code would produce a new list of elements from the objects list as string values.

Selection in Collections

The selection in OGNL is a simple way of using an expression for choosing some elements from the collection and saving the results in a new collection. For example, the following expression will return a list of all those listeners, which are instances of the `ActionListener` class:

```
listeners.{? #this instanceof ActionListener}
```

We can use indexing to get the first match from the list of matches, like `listeners.{? true} [0]`. This may throw a `ArrayIndexOutOfBoundsException`, if no match is found. OGNL provides a facility of selection here. This will select only the first match and return it as a list. If the match does not succeed, then it will return an empty list as a result:

```
objects.{^ #this instanceof String }
```

This code will return the first element contained in the objects that is an instance of the `String` class. Similarly, we can obtain the last element that matched using the following code:

```
objects.{$ #this instanceof String }
```

This code will return the last element contained in the objects that is an instance of the `String` class.

Evaluation of Expression

In case of OGNL parenthesized expressions, if there is no dot in front of the parentheses, then the OGNL will treat the result of the first expression as another expression to evaluate. The parenthesized expression result is treated as root object for the evaluation. The result of the first expression may be any object; if it is an AST (Abstract Syntax Tree), OGNL assumes it in the parsed form of an expression and just interprets it. Otherwise, OGNL takes the string value of the object and parses that string to get the AST to interpret it. For example, the expression:

```
#fact(30H)
```

looks up the `fact` variable, and interprets the value of that variable as an OGNL expression using the `BigInteger` representation of 30 as the root object.

Lambda Expressions

Lambda expressions are used for writing simple functions. OGNL provides a simplified lambda syntax, which will help you to write simple functions. This is not a full lambda calculus, because there are no closures. All variables in OGNL have global scope and extent. For example, the following OGNL expression declares a recursive factorial function, and then calls it:

```
#fact = :[#this<=1? 1 : #this*#fact(#this-1)], #fact(30H)
```

The lambda expression is everything inside the brackets. The `#this` expression holds the argument to the expression. It is initialized to `30H` and is then one less in each successive call to the expression.

Lambda expressions are treated as constants in the OGNL. The value of a lambda expression is the AST (Abstract Syntax Tree) that is used by OGNL as the parsed form of the contained expression.

The Multiple Uses of `#`

You have now seen three different uses of the `#` operator. For the sake of clarity, we'll outline these uses here. You may see the `#` character in various OGNL expressions, when:

- Referring to values in the `ActionContext`
- Constructing Maps on the fly
- Selecting or projecting a collection

Remember, a simple expression that includes something like `#foo`, uses the `#` character to refer to a value in the `ActionContext`. This is different than an expression that is creating a `Map` on the fly, because the `#` character isn't followed by curly braces `({})`. Whereas the `#foo` refers to an `ActionContext`, the variable, `#{1 : 'one', 2 : 'two'}` doesn't. Instead, the expression creates a new `Map`.

In the next section, we'll look at advanced features of OGNL that also use the `#this` keyword, so it's important to understand now how hash sign (`#`) work differently in an OGNL expression.

Properties for Collections

There are some special properties for collections that are made available by the OGNL. The reason for providing these properties is that the collections do not follow the JavaBeans patterns for method naming, therefore the `size()`, `length()`, etc. methods must be called, instead of more intuitively referring to these as properties. OGNL solves this problem by providing certain pseudo-properties as if they are already built-in. These properties are described in the Table 5.8.

Table 5.8: Properties for collections

Collection	Property name	Description
Collection (inherited by Map, List & Set)	size	It evaluates the size of the collection
	isEmpty	It evaluates to true, if the collection is empty
List	iterator	It evaluates to an Iterator over the List

Table 5.8: Properties for collections		
Collection	Property name	Description
Map	keys	It evaluates to the Set of all the keys in the Map
	values	It evaluates to a Collection of all values in the Map
Set	iterator	It evaluates to an Iterator over the Set
Iterator	next	It evaluates to the next object from the Iterator
	hasNext	It evaluates to true, if there is a next object available from the Iterator
Enumeration	next	It evaluates to the next object from the Enumeration
	hasNext	It evaluates to true, if there is a next object available from the Enumeration
	nextElement	It similar to next property
	hasMoreElements	It is similar to hasNext property

Setting Values vs. Getting Values

Sometimes some gettable values are not settable because of the nature of the expression. For example, the expression `names[0].location` is a settable expression and the final component of the expression resolves to a settable property.

However, some expressions, such as `names[0].length + 1`, cannot be set because they do not resolve to a settable property in an object. It is simply a computed value. If you try to evaluate such an expression using any of the `Ognl.setValue()` methods, it will fail with an `InappropriateExpressionException`. It is also possible to set the variables using get expressions that include the `=` operator. This is useful when a get expression needs to set a variable as a side effect of execution.

Interpreting Objects as Collections

The projection and selection operators (`e1.{e2}` and `e1.{?e2}`), and the `in` operator, all treat one of their arguments as a collection. This is done differently depending on the following class of the arguments:

- Java arrays are walked from front to back
- Members of `java.util.Collection` are walked by walking their iterators

- Members of `java.util.Map` are walked by walking iterators over their values
- Members of `java.util.Iterator` and `java.util Enumeration` are walked by iterating them
- Members of `java.lang.Number` are *walked* by returning integers less than the given number starting with zero
- All other objects are treated as singleton collections containing only themselves

Support for Value Stack

The biggest addition that XWork2 provides on top of OGNL is the support for the `ValueStack`. While OGNL operates under the assumption that there is only one *root*, XWork2's `ValueStack` concept requires that there be many *roots*.

NOTE

The relation among WebWork2, XWork2 and Struts 2 has been discussed in Chapter 1.

Suppose we are using the standard OGNL (not using XWork2) and there are two objects in the `OgnlContext` map—`"foo" -> foo` and `"bar" -> bar`—and that the `foo` object is also configured to be the single *root* object. The following code illustrates how the OGNL deals with these three situations:

```
#foo.blah // returns foo.getBlah()  
#bar.blah // returns bar.getBlah()  
blah // returns foo.getBlah() because foo is the root
```

What this means is that the OGNL allows many objects in the context, but unless the object you are trying to access is the *root*, it must be prepended with a namespaces, such as `@bar`. Now let's talk about how XWork2 is a little different.

Suppose the stack contains two objects, `Animal` and `Person`. Objects have a `name` property, `Animal` has a `species` property, and `Person` has a `salary` property. '`Animal`' is on the top of the stack, and '`Person`' is just below it. The following code fragment helps you get an idea of what is going on here:

```
species      // call to animal.getSpecies()  
salary       // call to person.getSalary()  
name        // call to animal.getName() because animal is on the top
```

In the last example, there was a tie and the `animal`'s name was returned. Usually this is the desired effect, but sometimes you may require the property of a lower-level object. To access property of lower-level object in `ValueStack`, XWork2 has added support for indexes on the `ValueStack`. All you have to do is as follows:

```
[0].name    // call to animal.getName()  
[1].name    // call to person.getName()
```

With expressions, like `[0] ... [3]`, etc. Struts 2 will cut the Stack and still return a `CompoundRoot` object. To get the top of that particular stack cut use `[0].top`. This (`[0].top`) would get the top of the stack cut starting from element 0 in the stack.

Accessing Static Property and Methods

OGNL supports accessing static properties as well as static methods. As the OGNL docs point out, you can explicitly call static by the following:

```
@some.package.ClassName@FOO_PROPERTY  
@some.package.ClassName@someMethod()
```

However, XWork2 allows you to avoid having to specify the full package name and call static properties and methods of your action classes using the `vs` prefix:

```
@vs@FOO_PROPERTY  
@vs@someMethod()  
  
@vs1@FOO_PROPERTY  
@vs1@someMethod()  
  
@vs2@BAR_PROPERTY  
@vs2@someOtherMethod()
```

Here, `vs` stands for *value stack*. The important thing to note here is that if the class name you specify is just `vs`, the class for the object on the top of the stack is used. If you specify a number after the `vs` string, an object's class deeper in the stack is used, instead.

Using OGNL in Struts 2

The Struts 2 Framework uses a standard naming context for evaluating an OGNL expression. The topmost object dealing with OGNL is a Map, usually called as a context map or context. OGNL has a notion that there should be a root object within the context. In OGNL expressions, the properties of the root object can be referenced without any special marker notion.

The framework sets the OGNL context to be the `ActionContext`, and the value stack to be the OGNL root object. In addition to the value stack, the Struts 2 Framework places other objects in the `ActionContext`. They include maps, representing the application, session, and request contexts. These objects coexist in the `ActionContext`, alongside the value stack.

The `Action` instance is always pushed onto the value stack. Because the `Action` is on the stack, and the stack is the OGNL root, references to `Action` properties can omit the `#` marker. But, to access other objects in the `ActionContext`, we must use the `#` notation so that the OGNL knows not to look in the root object, but for some other object in the `ActionContext`. The following code shows how to reference an `Action` property:

```
<s:property value="postalCode"/>
```

Other non-root objects in the `ActionContext` can be referenced using the `#` notation as follows:

```
<s:property value="#session.mySessionPropKey"/> or  
<s:property value="#session['mySessionPropKey']"/> or  
<s:property value="#request['mySessionPropKey']"/>
```

Since the Collections (Maps, Lists, and Sets) in the framework are referred very often, here are a few examples using the select tag.

- ❑ **Syntax for list** ({e1, e2, e3})—This idiom creates a list containing the String name1, name2 and name3. It also selects name2 as the default value:

```
<s:select label="label" name="name"
list="{'name1','name2','name3'}" value="%{'name2'}" />
```

- ❑ **Syntax for map**: (#{}key1:value1, key2:value2)—This idiom creates a map that maps the string foo to the string foovalue and bar to the string barvalue:

```
<s:select label="label" name="name" list="#{
'foo':'foovalue', 'bar':'barvalue'}" />
```

To determine if an element exists in a Collection, use the operations in and not in:

```
<s:if test="'foo' in {'foo','bar'}">
    muhahaha
</s:if>
<s:else>
    boo
</s:else>

<s:if test="'foo' not in {'foo','bar'}">
    muhahaha
</s:if>
<s:else>
    boo
</s:else>
```

To select a subset of a collection (called projection), use from the following wildcard within the collection:

- ❑ ?—All elements matching the selection logic
- ❑ ^—Only the first element matching the selection logic
- ❑ \$—Only the last element matching the selection logic

To obtain a subset of just male relatives from the object person:

```
person.relatives.{? #this.gender == 'male'}
```

OGNL supports basic lambda expression syntax enabling you to write simple functions which returns the nth element in Fibonacci series:

```
fib: if n==0 return 0; elseif n==1 return 1; else return fib(n-2)+fib(n-1);
//Output of different method calls with argument 0, 1 and 11
fib(0)= 0
fib(1)= 1
fib(11)= 89
```

The following code shows the usage of lambda expression for the same fib method in Struts 2 Framework:

```
<s:property  
value="#fib =:[#this==0 ? 0 : #this==1 ? 1 : #fib(#this-2)+#fib(#this-1)],  
#fib(11)" />
```

The lambda expression is everything inside the brackets. The #this variable holds the argument to the expression, which is initially starts at 11.

Immediate Solutions

Creating an Struts 2 application with OGNL

In this section we are going to develop an application, which uses OGNL expressions in generating contents to be displayed on the JSP by interacting different Java objects. This application accesses bean's property, and requests object and session object using OGNL. The directory structure of the application here, as shown in Figure 5.1, is similar to what we are using in other applications developed in the previous chapters.

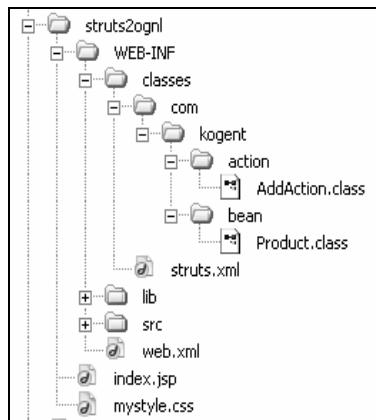


Figure 5.1 : Directory structure of struts2ognl application.

The application is designed with a simple JSP page (`index.jsp`) and a single action class `AddAction`, which processes the data submitted through the form designed in `index.jsp` page. The application gives a console to select an item of user's choice and required quantity. This order detail is represented by a JavaBean named `Product`. All these code files are described here for you.

JavaBean—The *Product* Class

This application uses a JavaBean, which is used to collectively represent currently selected product name and quantity through `index.jsp` page. The JavaBean `Product` has two properties—`productName` and `quantity`. The class also defines getter/setter methods for these properties.

Here's the code, given in Listing 5.1, that describes the code for `Product` JavaBean (you can find `Product.java` file in `Code\Chapter 5\struts2ognl\WEB-INF\src\com\kogent\bean` folder in CD):

Listing 5.1: Product.java

```

package com.kogent.bean;

public class Product {

    private String productname;
    private int quantity;

    public String getProductname() {
        return productname;
    }

    public void setProductname(String productname) {
        this.productname = productname;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}

```

AddAction Class

Now we move towards our action class, i.e. AddAction. This is a simple action class, which extends `com.opensymphony.xwork2.ActionSupport` class and implements two interfaces—`ServletRequestAware` and `SessionAware`. The implementation of different methods of `ServletRequestAware` and `SessionAware` interfaces makes request and session object accessible in our action class, respectively. The action class itself has two input properties, i.e. `productname` and `quantity`. The action class defines setters and getters for these input properties and implements `setServletRequest()`, `getServletRequest()`, `setSession()` and `getSession()` methods. The most important method of the class, which is its `execute()` method, implements the logic of binding different objects with request and session objects, so that they can be accessed from different JSP pages using OGNL expressions.

Here's the code, given in Listing 5.2, for `AddAction.java` (you can find `AddAction.java` file in `Code\Chapter 5\struts2ognl\WEB-INF\src\com\kogent\action` folder in CD):

Listing 5.2: AddAction.java

```

package com.kogent.action;

import java.util.ArrayList;
import java.util.Map;
import javax.servlet.http.*;
import org.apache.struts2.interceptor.*;
import com.opensymphony.xwork2.ActionSupport;

```

```
import com.kogent.bean.Product;

public class AddAction extends ActionSupport implements ServletRequestAware,
    SessionAware {
    private HttpServletRequest request;

    private String productname;

    private Map session;

    private int quantity;

    public String getProductname() {
        return productname;
    }

    public void setProductname(String productname) {
        this.productname = productname;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void setServletRequest(HttpServletRequest request) {
        this.request = request;
    }

    public void setSession(Map session) {
        this.session = session;
    }

    public Map getSession() {
        return session;
    }

    public HttpServletRequest getServletRequest() {
        return request;
    }

    public String execute() throws Exception {
        if(productname.equals("None")){
            this.addActionError("Select an item!");
            return ERROR;
        }

        ArrayList beans = null;
        Product bean = new Product();
        bean.setProductname(productname);
        bean.setQuantity(quantity);

        beans = (ArrayList) session.get("beans");
    }
}
```

```

        if (beans == null)
            beans = new ArrayList();
        beans.add(bean);
        session.put("beans", beans);
        request.setAttribute("bean", bean);
        return SUCCESS;
    }
}

```

Compile Product.java and AddAction.java and place the .class file in WEB-INF\classes folder within the corresponding package, as shown in Figure 5.1.

Creating index.jsp Page

This JSP page prompts the user to select an item from the list of items and inputs the quantity for the selected item. The form created in index.jsp page contains two input fields. In addition to this form, the page also displays different content, like currently selected product and the list of products selected throughout the session. The main point to be emphasized here is that this data access is implemented using OGNL expression.

Here's the code, given in Listing 5.3, for index.jsp (you can find index.jsp file in Code\Chapter 5\struts2ognl folder in CD):

Listing 5.3: index.jsp page

```

<%@ page language="java" session="true" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Select An Item</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h2>Using OGNL Expression</h2>
    <table cellspacing="5" >
        <tr>
            <td align="center" valign="top" bgcolor="#ffe8dd">
                <br>Select an Item and Quantity
                <s:actionerror/>
                <s:form action="addProduct">
                    <s:select
                        label="Items"
                        list="{'Sandwich', 'Pastry', 'Patties', 'Pizza'}"
                        name="productname"
                        headerKey="None"
                        headerValue="None"/>
                    <s:textfield
                        label="Quantity"
                        name="quantity"/>
                    <s:submit/>
                </s:form>
            </td>
            <td align="center" valign="top">

```

```

| Accessing Bean Properties |<br><br>
<table cellpadding="2" cellspacing="0" width="200">
<tr bgcolor="#cdf3f5"><td>Item</td><td>Quantity</td></tr>
<tr>
<td><s:property value="productname"/></td>
<td><s:property value="quantity"/></td>
</tr>
</table>
</td>

<td align="center" valign="top">
| Accessing Request |<br><br>
<table cellpadding="2" cellspacing="0" width="200">
<tr bgcolor="#cdf3f5"><td>Item</td><td>Quantity</td></tr>
<tr>
<td><s:property value="#request['bean'].productname"/></td>
<td><s:property value="#request['bean'].quantity"/></td>
</tr>
</table>
</td>

<td align="center" valign="top">
| Accessing Session |<br><br>
<table cellpadding="2" cellspacing="0" width="200">
<tr bgcolor="#cdf3f5"><td>Item</td><td>Quantity</td></tr>
<s:iterator id="beans" value="#session['beans']">
<tr>
<td><s:property value="productname"/></td>
<td><s:property value="quantity"/></td>
</tr>
</s:iterator>
</table>
</td>

</tr>
</table>
</body>
</html>

```

The OGNL expression `index.jsp` page is used for three different objectives here. The first one is the access bean properties from the current value stack. This is obtained by using the following syntax, used in Listing 5.3:

```

<s:property value="productname"/>
<s:property value="quantity"/>

```

Here, the accessor methods from the current value stack is invoked. These methods are `getProductname()` and `getQuantity()`, which must be defined in the last action class executed, i.e. our `AddAction` class.

Similarly, we can access different objects and their properties from the request and session scopes. The OGNL expression has helped in reducing lines of Java code into single line OGNL expression. The OGNL expression `#request['bean']` puts the request attribute named `bean` on the value stack and from there it is accessed using the `.`(dot) notation. The following code lines, taken from Listing 5.3,

access a Product type of object saved as request attribute with the name bean and two properties of this object—productname and quantity.

```
<s:property value="#request['bean'].productname"/>
<s:property value="#request['bean'].quantity"/>
```

The collection can also be accessed in a similar way and can be iterated over to access different properties of different objects stored in the collection object (ArrayList, Map, etc.). Observe the code lines taken from Listing 5.3, which show how the session scoped ArrayList is being accessed and iterated over:

```
<s:iterator id="beans" value="#session['beans']">
<br>
    <s:property value="productname"/>
    <s:property value="quantity"/>
</s:iterator>
```

With this code the JSP accesses the session attribute, beans, which is set in the Action class and stored in an ArrayList. The OGNL expression #session['beans'] puts the session attribute on the value stack and from there it is accessed by iterating over this value using <s:iterator>. You can also access the session value stored in the session attribute using the . (dot) notation. In that case the expression will be as follows:

```
#session['student'].name
#session['student'].city
```

Configuring the Application

The two basic configuration files used similar to other Struts 2 application are web.xml and struts.xml file. The index.jsp page is defined in <welcome-file-list> here.

Here's the code, given in Listing 5.4, for creating web.xml and making it Struts 2 enabled by providing mapping for FilterDispatcher class. You can find web.xml file in Code\Chapter 5\struts2ognl\WEB-INF folder in CD.

Listing 5.4: web.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"

    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>struts2ognl application</display-name>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
```

```
</filter>
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Here's Listing 5.5 showing action mapping provided in struts.xml (you can find struts.xml file in Code\Chapter 5\struts2ognl\WEB-INF\classes folder in CD):

Listing 5.5: struts.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>

<package name="default" extends="struts-default">
    <action name="addProduct" class="com.kogent.action.AddAction">
        <result name="success">/index.jsp</result>
        <result name="error">/index.jsp</result>
    </action>
</package>
</struts>
```

Create your copy of struts.xml file according to Listing 5.5 and save it in WEB-INF\classes folder. The action AddAction may return 'success' or 'error' as result code and, in both cases, index.jsp page is rendered for the user

Running the Application

Now, we are ready to execute the application. Start your Web server and enter the following URL in your web browser:

```
http://localhost:8080/struts2ognl
```

The output of the application gets displayed in Figure 5.2.

Figure 5.2: The index.jsp showing two input fields

The first page to appear here is `index.jsp`. Figure 5.2 shows a form with two input fields. You can select an item of your choice and enter the required quantity in the ‘Quantity’ field before clicking over the ‘Submit’ button.

Suppose that the first time user selects the item ‘Sandwitch’ and its quantity as ‘2’. Now clicking on the ‘Submit’ button invokes the `AddAction` action class with the output shown again in `index.jsp`, but this time with some data displayed using OGNL expression. You can see the affect in Figure 5.3.

Figure 5.3: Output of index.jsp after single submission of form.

Observe that the current item and the quantity submitted are present at three locations, but they all are accessed using different syntax as described in Listing 5.3.

On submitting different items and quantity, the output of `index.jsp` page changes to the one shown in Figure 5.4.

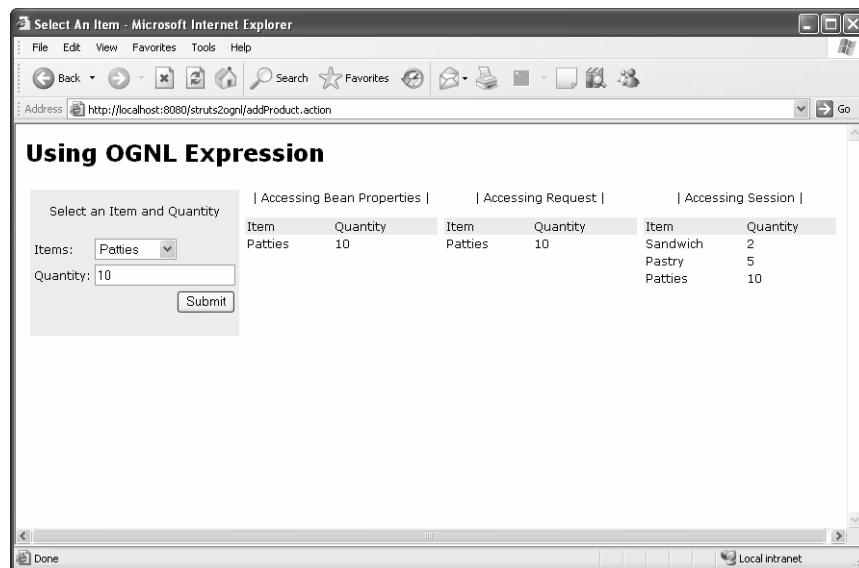


Figure 5.4: Output of index.jsp after few submissions of form.

By now we have known that Struts 2 provide a rich language for accessing and updating data. The most important features, however, are the simplest ones. Complex expressions and accessing collections are by far the most common features of the expression language you'll be using. That isn't to say that advanced features, like lambda expressions, won't be used, but they shouldn't be over-emphasized. The most important thing that you should remember in this chapter is how the expression language interacts with the value stack. As we mentioned in the beginning, we decided to use `John` objects for these examples. And yet, because we're going through an expression language (rather than writing native Java code to get and set various data elements), all that matters is that they share common properties and are available in the value stack. Take advantage of this loose coupling by not being afraid to utilize the value stack. The next chapter will throw light on the use of different Generics Tags in controlling execution flow in a JSP page.



6

Controlling Execution Flow with Tags in Struts 2

If you need an immediate solution to:

Using Control Tags with Data Tags

See page:

215

In Depth

In previous chapters, we used many JSP pages in which so many tags with prefix ‘s’ were used. These tags are known as Struts 2 tags. In this chapter, we’ll explore the rich library of different types of tags that Struts 2 provides. The tags in Struts 2 can be divided into two categories—Generic tags and UI tags. While this chapter will completely focus on Generic tags, the next chapter will examine UI tags. Generic tags are used for controlling the flow of data and data extraction from the value stack or other locations. In other words, the Generic tags are developed to control the flow of execution in the page and to obtain data and display it.

Implementing Generic Tags

Similar to other tags, Generic tags are also contained in a tag library which is a collection of predefined tags. We have been using Struts 2 tag library in our previous applications and designed number of JSP pages using Struts 2. Each tag has a specific work and has many attributes. You can use these tags in your JSP page by simply importing tag library in your JSP page like this:

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

The directive identifies the URI defined in the previously listed `<taglib>` element and states that all tags should be prefixed with string ‘s’. In Struts 1 there are many tag libraries but in Struts 2 only one tag library is specified, i.e. struts-tags. The Struts 2 tags are grouped into different parts. The Generic tags are the important part of these tag library. Struts Generic tags control the execution flow as pages render. The Generic tags are specified in `/struts-tags` tag library. There are two types of Generic tags:

- Control tags
- Data tags

Control Tags

Like every other language, the Struts 2 tags have a set of tags that make it easy to control the flow of page execution. This can be done with the help of Control tags. Control tags are used to control the behavior of data in the page. Using the `iterator` tag with `append` and `merge` tag to loop over data and the `if\else\elseif` tag to make decisions, you can develop more sophisticated dynamic web pages. Control tags, as the name suggests, are designed to control the path of execution with the help of some decision making constructs, like `if`, `else` and `elseif` tags. Another controlling structure is looping constructs, which is supported by the `iterator` tag. These tags allow easy customization of content displayed to the user, according to the data in the context. There are many Control tags available in Struts 2. Table 6.1 lists the Control tags.

Table 6.1: Control tags in Struts 2

Tag	Description
if	It is used to implement the conditional flow of execution
elseif	It is used with if tag and extends the decision making to a number of conditions
else	It is used to provide alternate execution path if all conditions given in if tag and elseif tag(s) fails.
append	It appends iterable lists together to form an appended iterator
generator	It generates an iterator on the basis of a given value.
iterator	It is used to iterate over the given value
merge	It is used to merge iterable lists together to form a merged iterator
sort	It is used to sort a given List
subset	It takes an iterator and outputs its subset

All of these tags are discussed in the text that follows with their functionality, syntax, and set of required and optional attributes for them.

The **if** Tag

The if tag is used to check the condition and execute some codes defined in its body, when the condition is tested to be true. You can use this tag alone or with else tag or elseif tag. The if tag has two attributes – id and test. The condition to be tested is given as test attribute. If the condition given under the test attribute is true, then the corresponding body is evaluated. The following code snippet shows how an if tag is used in your JSP pages:

```
<s:if test="#">  
    This line will be displayed.  
</s:if>  
  
<s:if test="#">  
    This line will never be displayed.  
</s:if>
```

As already mentioned, the if tag has two attributes, namely id and test. The id attribute is optional but you have to mention the test attribute, whenever you use if tag. The types of id and test attribute are String and Boolean, respectively. The test attribute must be a Boolean expression. Table 6.2 describes these attributes.

Table 6.2: Attributes of if tag

Attribute name	Description
id	It is used for referencing elements
test	This expression determines if body of tag is to be displayed (required)

The **else** Tag

The **if** tag body is executed when the test expression returns true. Sometimes, we want to execute some other logic when the test expression given in **if** tag returns false. In such a situation, we'll implement **if-else** constructs using **if** tag and **else** tag. For a single **if** tag there can be single **else** tag, which can contain the logic to be executed when the **if** tag is not executed. Using the **else** tag, without the **if** tag, is totally illogical and results in no execution. First, the condition is checked by the **if** tag. If this condition is false then the control goes to the **elseif** tag. If both the **if** and the **elseif** tag has false conditions, then the **else** tag is evaluated. The **else** tag has only one attribute, i.e. the **id** attribute, which is optional. The following code lines show the usage:

```
<s:if test="type=='manager'">
    You are a Manager.
</s:if>
<s:else>
    You are a Clerk.
</s:else>
```

The **elseif** Tag

The **if** tag is used to check a single condition, but sometimes we need to test a number of possible conditions which can be implemented using **if-elseif-else** construct using **if**, **elseif**, and **else** tag.

The **elseif** tag is not used alone, but with **if** tag. The condition given with **if** tag is checked first. If the evaluation of this condition is false, then the control goes to the **elseif** tag. An **elseif** tag may be followed by an **else** tag, which will be executed only when all test expressions given in the **if** tag and **elseif** tags return false. There may be more than one **elseif** tags used with a single **if** tag. Here's the syntax for using the **elseif** tag:

```
<s:if test="%{false}">
    This will not be executed.
</s:if>
<s:elseif test="%{false}">
    This will not be executed.
</s:elseif>
<s:else>
    This line will be executed.
</s:else>
```

The `elseif` tag has two attributes. These attributes are `id` and `test`. The `id` attribute is optional, but you have to mention the `test` attribute whenever you use the `elseif` tag. The `id` and `test` attributes are of `String` and `Boolean` type, respectively. The `test` attribute must be a Boolean expression.

The **iterator** Tag

The `iterator` tag allows you to loop over collections of objects easily. It's designed to loop over any `Collection`, `Map`, `Enumeration`, `Iterator`, or `array`. It also provides the ability to define a variable in the action context that lets you determine certain basic information about the current loop state, such as whether you are looping over an odd or even row. One of the nicest features about the iterator tag is that it can iterate over just about any data type having a concept of iteration. It has three attributes, namely `id`, `status` and `value`. An iterable value can be an object reference of either the `java.util.Collection` interface or the `java.util.Iterator` interface. The `iterator` tag uses the `property` tag to display the value from the current object in the collection.

Sometimes it is desirable to know the status information about the iteration that's taking place. This is where the `status` attribute steps in. The `status` attribute, when defined, provides an `IteratorStatus` object available in the action context that can provide simple information, such as the size, current index, and whether the current object is in the even or odd index in the list. The most common use for the iterator status feature is to render a table of values and shade even-numbered rows one color and odd-numbered rows another color.

Here's a simple example showing this:

```
<s:iterator status="stat" value="{23, 12, 34, 12, 15, 89}" >
    [ <s:property value="#stat.index" /> | <s:property value="top" /> ]

    <s:if test="#stat.last==false">, </s:if>
</s:iterator>



---


//The output of this syntax
[ 0 | 23 ] , [ 1 | 12 ] , [ 2 | 34 ] , [ 3 | 12 ] , [ 4 | 15 ] , [ 5 | 89 ]
```

The `last()` method in the preceding example indicates if the current object is the last available in the iteration, and if not, we need to separate the members using a comma. The `iterator` tag has three attributes, namely the `id`, the `status` and the `value`. Table 6.3 describes these attributes.

Table 6.3: The iterator tag attributes

Attribute name	Description
<code>id</code>	It is used for referencing element.
<code>status</code>	It can be set to true or false. If it is set to true, then an instance of <code>IteratorStatus</code> will be pushed into stack on each iteration. Its use is optional.
<code>value</code>	This attribute is set with the iterable source to iterate over. If not mentioned then an object itself will be put into the newly created List. Its use is optional.

The **append** Tag

This tag is used to append collection objects (iterators) to make a single collection object (appended iterator) which can be iterated using the `iterator` tag. The value of appended iterators is accessed by using `iterator` tag. The `append` tag has only one attribute, i.e. `id`. Entries go from one iterator to another iterator after each respective iterator is exhausted of entries, i.e. when iterating over the appended iterator all the elements of the first iterable object appended will be accessed. If all entries from an iterator are finished then the `append` tag stops processing that iterator and moves forward with other iterators. The value of `id` attribute of `append` tag will be used as a value of `value` attribute of `iterator` tag to iterate over the appended iterator.

Here's syntax which creates an appended iterator and finally we iterate over this appended iterator using `iterator` tag.

```
<s:append id="myIterator">
    <s:param value="% {fruits}" />
    <s:param value="% {cities}" />
    <s:param value="% {colors}" />
</s:append>
```

The preceding syntax will search for `getFruits()`, `getCities()`, and `getColors()` to get some collection object to be returned from the current value stack. Suppose that we have three `ArrayList` objects 'fruits', 'cities' and 'colors' and these objects have three entries each. For example, the `ArrayList` 'fruits' contains 'Apple,' 'Mango,' and 'Orange'; the second `ArrayList` 'cities' contains 'Delhi,' 'Mumbai,' 'Pune'; and the third `ArrayList` 'colors' contains 'Red,' 'Green,' 'Blue.' In this case, the `iterator` tag and its output will be as shown here:

```
<s:iterator value="#{#myIterator}">
    <s:property /><br>
</s:iterator>

//Output
Apple
Mango
Orange
Delhi
Mumbai
Pune
Red
Green
Blue
```

The contents of the second `ArrayList` appended ('cities') are displayed only after the contents of the first `ArrayList` appended ('fruits') are finished. Note that the `append` tag only appends the iterators, which are accessed by the `iterator` tag.

The `append` tag has only one attribute which is `id`. It must be a `String` type and optional too. The `id` attribute represents the appended iterators. You can access the appended iterators using this `id`.

The **merge** Tag

This tag is used to merge the iterator. It is generally used with the `iterator` tag. This tag is quite different from the `append` tag. This tag gives chance to each iterator available for merging to expose its

elements; the next call will allow the next iterator to expose its element. Once the last iterator has exposed all its elements, the first iterator is allowed to do so again, unless its entries are not exhausted. It has only one attribute `id`, which has the same name as `value` attribute of `iterator` tag. We are using the same example used for `append` tag earlier. Observe the output shown in the following code snipped to find how the `merge` tag is different from `append` tag.

```
<s:merge id="myIterator">
    <s:param value="% {fruits}" />
    <s:param value="% {cities}" />
    <s:param value="% {colors}" />
</s:merge>

<s:iterator value="#{myIterator}">
    <s:property/><br>
</s:iterator>
```

```
//Output
Apple
Delhi
Red
Mango
Mumbai
Green
Orange
Pune
Blue
```

The output shows that the first three elements are first elements of three different merged iterators. The `merge` tag has only one attribute, which is `id`. It is `String` and optional too. The `id` attribute represents the merged iterators collectively. You can access the merged iterators using this attribute.

The **generator** Tag

This tag is used to generate the iterator based on the `val` attribute supplied to this tag. The generated iterator is always pushed to the top of the stack, and popped at the end of the tag. The following code snippet will generate an iterator based on the values supplied in the `val` attribute and then print these values using `iterator` tag:

```
<s:generator
    val="#{'Mon,Tue,wed,Thu,Fri,sat,Sun'}"
    count="6"
    separator=""
    id="someid">
    <s:iterator>
        <s:property />
    </s:iterator>
</s:generator>

//Output
Mon Tue wed Thu Fri Sat
```

The generator tag has five attributes, namely converter, count, id, separator, and val. The converter attribute is of type `org.apache.struts2.util.IteratorGenerator.Converter`, and the count attribute is of `Integer` type and the remaining three are of `String` type. Table 6.4 shows the descriptions of these attributes.

Table 6.4: The generator tag attributes.	
Attribute name	Description
converter	It converts the String values passed in val attribute into an object (optional)
count	It is the maximum number of entries in the iterator (optional)
id	This id is used to store the resultant iterator into page contexts (optional)
separator	The separator is used to separate the values into entries of iterator (required)
val	The source to be parsed into an iterator (required)

The **sort** Tag

This tag is used to sort a List by using the `java.util.Comparator`. Both the List and Comparator's instance are passed as an attribute to this tag. The sorted list is placed into the `PageContext` attribute using the key specified by the id. It has only two attributes—comparator and source. The source attribute specifies the source to be sorted. The following code lines shows the use of `<s:sort>` tag:

```
<s:sort comparator="myComparator" source="myList">
    <s:iterator>
        <s:property value=" . . " />
    </s:iterator>
</s:sort>
```

Of the two attributes, the comparator attribute is required and must be of type `java.util.Comparator`. The source attribute gives the name of the source, which is to be sorted.

The **subset** Tag

Subset tag is a tag that takes an iterator and outputs a subset of it. It delegates to `org.apache.struts2.util.SubsetIteratorFilter` internally to perform the subset functionality. Suppose an `ArrayList` myList contains eight entries and you want to take only four entries from it, then you can use the subset tag. The extracted entries can be accessed by using the iterator tag. The syntax of using subset tag is as follows:

```
<s:subset source="myList" count="13" start="3">
    <s:iterator>
        <s:property />
    </s:iterator>
</s:subset>
```

The subset tag has four attributes—count, decider, source and start. The count and start attributes must be of Integer type and the decider attribute is of type org.apache.struts2.util.SubsetIteratorFilter. The source attribute takes a String giving the name of a List to get its subset. Table 6.5 shows the descriptions of these attributes.

Table 6.5: The subset tag attributes	
Attribute name	Description
count	It is the number of entries in the resulting subset iterator (optional)
decider	It determines that a particular entry is to be included in the resulting subset iterator (optional)
source	It indicates the source from which the resulting subset iterator is to be derived
start	It indicates the starting index of entries in the source to be available as the first entry in the resulting subset iterator (optional)

Data Tags

Data tags are used for creating and manipulating data. These tags let you either get data out of the value stack or place variables and objects in the value stack. Before discussing Data tags, we'll explain the meaning of value stack. The value stack is just a stack of objects. Initially, the stack contains input properties of the action executed. This is why when we write value="name", it results in calling of getName() method of the action class. New objects are placed onto the stack when the <s:iterator> or <s:property> tags are used. Returning to Data tags, there are many Data tags available in Struts 2. Table 6.6 shows the Data tags.

Table 6.6: Data tags	
Attribute name	Description
a	It is used to create an hyperlink similar to using HTML tag.
action	It is used to call actions directly from a page.
bean	It is used to create an instance of class, which is in agreement with JavaBean specification.
date	It is used to format and display date object in different ways.
debug	It creates a link, which can be clicked to view all value stack contents with different items available in the stack context. This helps in debugging.
i18n	It is used to get a resource bundle and place it on value

Table 6.6: Data tags

Attribute name	Description
	stack, so that in addition to the bundle associated with action, other resource bundles could be used.
include	It includes output of some Servlet or JSP page.
param	It is an important tag to define parameters to other tags.
push	It pushes value stack.
set	It is used to assign a value to a variable. It is something like setting a variable with the given value.
text	It is used to render an internationalized message from the resource bundle.
url	It is used to create a URL.
property	It is used to get the property of a value and returns the object from the top of the stack, by default.

a Tag

This tag is similar to HTML anchor tag. For the best use of a tag, first create your URL, then leverage this URL into you `<s:a/>` tag, and then pass it as an additional parameter in `s:a` tag. For this, give the reference of `id` attribute of `<s:url/>` tag in `href` attribute of `<s:a/>` tag. The following snippet shows the use of `<s:a/>` tag to create a hyperlink which invokes `addAction` action:

```
<s:url id="url" action="addAction"></s:url>
<s:a href="#">%{url}
```

The `a` tag has many attributes. Table 6.7 describes these attributes.

Table 6.7: The a tag attributes

Attribute name	Description
accesskey	It sets the html <code>accesskey</code> attribute on rendered html element
cssClass	It is used to set CSS class for the element created using this tag.
cssStyle	It is used to provide <code>cssStyle</code> definitions for element
disabled	It sets the <code>disabled</code> attribute of HTML on the element
errorText	The text to display the user at the time when an error is found during fetching the content.

Table 6.7: The a tag attributes

Attribute name	Description
executeScripts	If it is set to true, the JavaScript code will be executed in the fetched content
formFilter	It is the function name used to filter the fields of the form
formId	This is the id of HTML form whose fields are serialized and passed with the request.
id	It is used as HTML id attribute
key	It sets the key for this element and its value is stored in resource bundle
label	It gives a specific level to the element
labelposition	It defines label position of form element
handler	It is a Javascript function name that will make the request
href	It specifies the URL
indicator	It is used to set id of element that will be shown while making a request
loadingText	It is the text to be shown while the content is being fetched
notifyTopics	It displays the topics after the remote call completes
onLoadJS	It is the Javascript code executed after reload
name	It specifies the name of an element
onblur	It sets the onblur attribute of HTML on the element
onchange	It sets the onchange attribute of HTML on the element
onclick	It sets the onclick attribute of HTML on the element
ondblclick	It sets the ondblclick attribute of HTML on the element
onfocus	It sets the onfocus attribute of HTML on the element
onkeydown	It sets the onkeydown attribute of HTML on the element
onkeypress	It sets the onkeypress attribute of HTML on the element
onkeyup	It sets the onkeyup attribute of HTML on the element
onmousedown	It sets the onmousedown attribute of HTML on the element
onmousemove	It sets the onmouseover attribute of HTML on the element
onmouseout	It sets the onmouseout attribute of HTML on the element

Table 6.7: The a tag attributes	
Attribute name	Description
onmouseover	It sets the onmouseover attribute of HTML on the element
onmouseup	It sets the onmouseup attribute of HTML on the element
onselect	It sets the onselect attribute of HTML on the element
openTemplate	It sets template to use for opening the rendered HTML.
tabindex	It sets the tabindex attribute of HTML on the element
template	It sets the template used for the element
templateDir	It sets the template directory.
theme	It sets the theme to be used for rendering the element.
title	It sets the title attribute of HTML on the element
tooltip	It sets the tooltip of this particular element
tooltipConfig	It sets the tooltip configuration
showErrorTransportText	It can be set to true/false depending on whether errors will be shown

The **action** Tag

This tag is used to call actions directly from a JSP page. It can be done by specifying the action name and an optional namespace. Mapping for this action class in struts.xml will be ignored. The body content of the tag is used to render the results from the action.

You can use the `action` tag to add more advanced behavior to your pages. Rather than just putting JavaBeans into your action context, you can execute actions and then access the data in your JSP.

Only the `name` attribute in the `action` tag is required. By default, the `action` tag won't execute the result of the action, making it safe to execute actions that might, otherwise, normally cause a different page to be rendered. Also, the `namespace` parameter is required only if the namespace of the action you are executing is different than the current namespace of the original action. The `action` tag is great for creating simple reusable components without having to add scriptlets to your JSP pages.

Suppose you have an action class with the name `RegistrationAction`, then here's how the action definition looks like in `struts.xml`:

```
<action name="registrationForm" class=" kogent.com.RegistrationAction">
    <result name="success"/>/registeredUser.jsp</result>
</action>
```

To display the registered user listing, invoke the `action` tag and tell it to execute its result. This will render the output of `registerUser.jsp` directly inline in any of your pages.

The following snippet for a sample JSP page displays the use of <s:action/> tag:

```
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
<head>
<title>New User</title>
</head>
<body>
<br/>
<s:action name="registrationForm" executeResult="true"/>
</body>
</html>
```

Note that here the `executeResult` is set to true. This is necessary if you wish to tell Struts 2 to invoke the typical output you would expect if you made an HTTP request directly to `registrationForm.action`. If `executeResult` is not set, it would be set to false by default; in this case, although the action would execute, nothing would be rendered.

The action tag has 6 attributes, namely `executeResult`, `flush`, `id`, `ignoreContextParams`, `name`, `namespace`. All attributes, except the `name` attribute, are optional. The attributes, `executeResult`, `flush`, and `ignoreContextParams`, are of type Boolean, while `id`, `name`, and `namespace` are of String type. Table 6.8 describes these attributes.

Table 6.8: The action tag attributes

Attribute name	Description
<code>executeResult</code>	It can be set to true or false to assure whether the result of this action should be executed. If it is set to true then the result will be executed; default value is false.
<code>flush</code>	It can be set to true or false and represents whether the writer should be flushed upon end of action component tag. Its use is optional.
<code>id</code>	It is used as HTML id attribute (optional)
<code>ignoreContextParams</code>	This tag determines where the request parameter are included when the action is invoked (Optional)
<code>name</code>	It defines the name of the action to be executed. It is a required attribute.
<code>namespace</code>	It defined the namespace for action to be call. Its use is optional.

The **bean** Tag

Sometimes you need to provide logic that is more complex or data processing than the basic Struts 2 JSP tags can provide. The `bean` tag lets you create a simple JavaBean and push it on to the value stack. In addition to pushing the bean onto the stack between the opening and closing tags, you can also, optionally, assign a variable name for the bean to be accessible in the action context. This tag is used to instantiate a class that conforms to the JavaBean specification. It has only two attributes – `id` and `name`.

The name attribute specifies the name of class, which will be instantiated, and id attribute is set on the bean tag.

The bean tag is the one of a few tags we'll discuss that are *parameterized*, i.e. the tag is designed to surround the param tag, allowing you to customize the behavior of the tag by providing parameters. In the case of the bean tag, the parameters are used to set values on the properties of the bean.

Suppose you have a bean name BeanAction with a property name message, then the following code snippet shows how you can set and get that property using the bean tag:

```
<s:bean name="BeanAction" id="counter">
    <s:param name="message" value="Kogent"/>
    Hello ! : <s:property value="message"/>.
</s:bean>
```

Note that this example not only sets parameters but also renders output within the body of the bean tag. This is possible because the bean tag pushes the JavaBean onto the stack, allowing you to access the properties on the bean without having to prefix the bean's ID.

The date Tag

This tag is used to format the Date object in different ways. The date tag allows you to format a date in a quick and easy way. You can either use a custom format (dd/MM/yyyy) or predefined format with the key struts.date.format in your properties file.

In the following code snippet, we access the current date and then display it in the format 'dd/MMM/yyyy'. You can provide any custom format in the format attribute:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@page import="java.util.Date"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<body>
    <s:date name="new java.util.Date()" format="dd/MMM/yyyy" />
    <s:date name="new java.util.Date()" format="%{getText('app.date.format')}" />
</body>
</html>
```

The date tag has 4 attributes. Table 6.9 describes these attributes. All attributes are optional, except the name attribute.

Table 6.9: The date tag attributes

Attribute name	Description
format	It specifies the date format pattern
id	It is used for referencing element
name	The date value to format (required)

The **i18n** Tag

This tag is used to access the message from the resource bundle (.properties files). It has two attributes—`id` and `name`. The `name` attribute specifies the name of resource bundle used in your application.

Let's suppose that the name of the resource bundle used in your application is `Resources.properties`. It has a key `app.message` with value `WELCOME`. The following code snippet shows how this key value is accessed and prints it in your JSP page:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<body>
<s:i18n name="Resources">
    <s:text name="app.message"/>
</s:i18n>
<br>
</body>
</html>
```

The value of `id` attribute is used as HTML id and the `name` attribute specifies the name of resource bundle.

The **include** Tag

This tag is used to include the Servlet or JSP page. The additional value supplied by `<s:param>` tag in `include` tag are not accessible through the `<s:property>` tag. It has two attributes—`id` and `name`. Its `name` attribute specifies the name of Servlet or JSP page which will be included.

Let's consider an example for describing this tag. Let us have two JSP pages `header.jsp` and `footer.jsp` that have been provided in the CD. To include these pages in our new JSP page, we can use the `<s:include/>` tag, as shown in the following code snippet:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@page import="java.util.Date"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<body>
<s:include value="header.jsp"/>
<s:date name="new java.util.Date()" format="dd/MMM/yyyy" />
<s:include value="footer.jsp"/>
</body>
</html>
```

The **param** Tag

This tag is used to pass the parameters to other tags. It is used to add additional information in other tags. The parameters can be added with or without name as key. The `param` tag does nothing by itself, but at the same time, it's incredibly useful for many of the tags you have seen here as it is used with other tags. It is useful while working with the UI tags also, as you'll see in the next chapter. The `param` tag has three attributes—`id`, `name`, and `value`. All attributes are of `String` type. Note that neither the

name nor value attribute is required for the param tag. Remember that for all tags, except the text tag, parameters are given in the form of a name/value pair. But, because parameters are given in an indexed form for the text tag, the name attribute is not required.

Having an optional value may also seem a bit odd. That is only because we haven't looked at the alternative ways to supply a value with the param tag, via the tag body content. When you declare a param tag, the value can be defined in either a value attribute or as text between the start tag and the end tag. You can declare <s:param> tag in the following two ways:

- <s:param name="username">Kogent</param>
- <s:param name="username" value="Kogent" />

In the first case, the value would be evaluated to the stack as a java.lang.String object, whereas in the second case it would be treated as a java.lang.Object object.

The push Tag

This tag is used to push the value on value stack. It has two attributes—id and value. The value attribute specifies the value to be pushed on the stack. The push tag also allows you to push references onto the value stack. This tag makes it easy to work with a single object. It is useful when you wish to do a lot of work revolving around a single object. Rather than having to prefix every expression with the object name, you can push the object down on the value stack and then operate on it directly.

Suppose you have two pages to be rendered. Though the contents of the value stack are very different when both pages are invoked, you still want to display the contents of a user profile. Suppose that in page 1, the user model can be accessed via the expression cart.user, and in page 2, the user model can be accessed via the expression order.user. Using the push tag and a Struts 2 include, you can easily render the same user details.

The following code snippet provides a simple JSP for page 1:

```
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<title></title>
</head>
<body>
<s:push value="cart.user">
<s:include value="user-details.jsp" >
</s:push>
</body>
</html>
```

The following code snippet provides a simple JSP for page 2:

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

<body>
<s:push value="order.user">
<s:include value="user-details.jsp" >
```

```
</s:push>
</body>
</html>
```

Here's the common JSP page (`user-details.jsp`) included by both pages:

```
<%@ taglib uri="/struts-tags" prefix="s" %>
<ul>
<li>Username: <s:property value="username"/></li>
<li>Age: <s:property value="age"/></li>
<li>Address: <s:property value="address"/></li>
</ul>
```

This example shows how common JSP pages can be shared through the power of the value stack. The value stack allows you to write views that make assumptions about the top of the stack without having to worry what might be lower in the stack. In the JSP for page 1, the stack may have very different values than it has in the JSP for page 2, but it does not matter because by the time the final fragment is included, the top of the stack is the same in both pages. Notice that you can also access the user model by using the `property` tag with `push` tag. Suppose, in this case, the user model can be accessed via the expression `userprop`. Then the following code snippet shows how to access the user's details:

```
<s:push value="userprop" >
    <s:property value="username" />
    <s:property value="age" />
    <s:property value="address" />
</s:push>
```

The `push` tag has only two attributes—`id` and `value`. Table 6.10 describes these attributes.

Table 6.10: The push tag attributes

Attribute name	Description
<code>id</code>	It is used for referencing element
<code>value</code>	This tag specifies the value, which is push on the stack

The `set` Tag

The `set` tag is useful for evaluating an expression in the value stack and assigning it to a name in the specified scope. This is especially useful for placing temporary variables in your JSP, thereby making your code easier to write and read.

To understand how you typically use a `set` tag, let's look at a simple example. The following example shows the `property` tag accessing several fields of a `User` object that is stored in the session:

```
<s:property value="#session['user'].username" />
<s:property value="#session['user'].age" />
<s:property value="#session['user'].address" />
```

Repeating `#session['user']` every time is not only tiresome, but also error prone. A better way to do this is to set up a temporary variable that point to the User object. Here's the `set` tag in action, making the overall code easier to read:

```
<s:set name="user" value="#session['user']" />
<s:property value="#user.username" />
<s:property value="#user.age" />
<s:property value="#user.address" />
```

The `set` tag makes your pages simpler because it lets you refactor your expressions to smaller, more manageable pieces. However, the `set` tag can be used for more than just refactoring. It can also set values in different scopes. The `set` tag supports five scopes—action, application, session, request, and page. If no scope is specified, it will default to action scope. You have already seen the action scope in the previous example. It places objects in the action context, which can then be retrieved using the `#foo` notation you are already familiar with.

The other four scopes map directly to the four scopes that Servlet applications provide. Typically, you won't need to set values into these scopes unless you are trying to integrate with a tag or Servlet that doesn't know how to communicate with Struts 2. We'll not cover these scopes because they are most often used for legacy code. However, you should know their names. Extending the previous example, here's how the `push` tag can be used with `set` tag to simplify the view even further:

```
<s:set name="user" value="#session['user']" />
<s:push value="#user" >
<s:property value="username" />
<s:property value="age" />
<s:property value="address" />
</s:push>
```

The `set` tag has four attributes—`id`, `name`, `scope`, and `value`. Table 6.11 shows all attributes of String type along with their descriptions.

Table 6.11: The `set` tag attributes

Attribute Name	Description
<code>id</code>	It is used for referencing element.
<code>name</code>	It is used to set name of variable and it is required.
<code>scope</code>	It assigns a scope to variable. It can be application, session, request, page, or action.
<code>value</code>	It is used to set the value that is assigned to the variable.

The `text` Tag

The `text` tag is used for accessing a text message from the resource bundle. If the named message is not found, then the body of the tag will be used as the default message.

Let's suppose that the name of the resource bundle used in your application is `ApplicationResources` and it has a key `app.message` with the value `Welcome`. The following JSP file shown here is using

<s:text> tag, which will display a welcome string after getting a value for key app.message from the resource bundle:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<body>
<s:i18n name="ApplicationResources">
    <s:text name="app.message"/>
</s:i18n>
</br>
</body>
</html>
```

The name attribute specifies the key of the message resource property in the resource bundle. If the <s:text/> tag is used without enclosing it in <s:i18n>....<s:i18n>, then the default configured resource bundle of the application is searched for the message for corresponding key.

The url Tag

When you are building Web applications, it is extremely common to create URLs that link your various pages together. Struts 2 provides a URL tag to help you do this. It renders relative or absolute URLs, handles parameters, and encodes the URL so that it can be used with browsers that don't have cookies enabled. If you are planning to build a robust site that works with many different browsers and can be deployed in any application context, we highly recommend that you render all your URLs using the URL tag. You can use the param tag for additional request parameters.

The following code snippet generates an URL with the hyperlink ‘Using URL Tag’, clicking which takes you to appendAction.action:

```
<s:url id="url" action="appendAction"></s:url>
<s:a href="#">Using URL Tag</s:a>
```

Like other Struts 2 tags, the URL tag can be parameterized. The parameters are used to construct the URL and make up the query string.

The following example generates the URL string search.action?query=XYZ, where XYZ is the value the expression the name evaluates to:

```
<s:url value="search.action">
<s:param name="query" value="name"/>
</s:url>
```

The URL tag has many attributes and all attributes are optional. The most commonly used attributes are id, value, and action. The descriptions of all attributes are listed in Table 6.12.

Table 6.12: The url tag attributes

Attribute name	Description
action	It is used to set the action generated for the URL

Table 6.12: The url tag attributes

Attribute name	Description
anchor	It is used to set the name of anchor file for this URL
encode	It can be set to true or false. If set to true the parameters are encoded.
id	It is used for referencing element
includeContext	It can be set to true or false to decide whether the actual context should be included in URL or not.
includeParams	It is set with the params to be included. The three possible values for this attribute can be 'none', 'get' or 'all'.
method	It is set with the name of the method of action class to be invoked.
namespace	It is used to set the namespace for the action to be used.
portletMode	It is used to set the resulting portlet mode
scheme	This attribute is used to set the scheme attribute
value	It is set with the target value to use.
windowState	It defines the state of resulting portlet window.

The **property** Tag

The `property` tag is probably the most commonly used tag in Struts 2. If the `value` attribute isn't specified, it returns the object from the top of the value stack. Note that because the `value` data type isn't a string, it is automatically parsed. You have already seen examples of the `property` tag. The `default` attribute is useful if you want a value to be displayed when the value can't be evaluated to a non-null value. The `escape` attribute is useful if you want the output to be HTML escaped. By default, the values printed by the `property` tag are HTML escaped, i.e. the HTML tags are not taken care while rendering the output. It means that the default value of `escape` is true. This means that if a string contains the value `this & that`, it will be printed out exactly like that, which is technically invalid HTML. If you wish the output to be `this & that`, then you must set `escape` to false.

The following code snippet shows how the `property` tag accesses the property of an action or model:

```
Your name is <s:property value="userName" default="Could not Found"/><br><br>
Your password is <s:property value="password" /><br><br>
```

The `property` tag has four attributes, all of which are optional. Table 6.13 details these attributes.

Table 6.13: The property tag attributes

Attribute name	Description
default	It specifies the default value, which will be used when the value attribute is null.
escape	It specifies whether to escape HTML
id	It is used for referencing element
value	It defines the value to be displayed.

By now, you must be familiar with all the Control tags and Data tags provided by Struts 2 Framework. Let's move to the "Immediate Solutions" section to understand the implementation of these important Generic tags in the JSP pages.

Immediate Solutions

To implement a given functionality, the use of tags is always preferred over inserting Java code in the page. Hence, the framework has provided these helpful tags to replace the traditional way of writing Java code in the page to complete some common and frequent tasks, like controlling the flow of execution, accessing data, etc.

Here, we'll develop a Struts 2 application, named `generictags`, which follows the standard directory structure similar to other Struts 2 applications developed in the previous chapters. Hence, instead of discussing directory structure, we'll emphasize on using Struts 2 Generic tags.

Using Control Tags with Data Tags

Before designing your first JSP page for this application and using Struts 2 tags, we need a brief discussion about enabling your JSP page of using these tags. All JSP pages using Struts 2 tag library must have a `taglib` directive with `uri` and `prefix` attribute set as shown here;

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

All Struts 2 tags can be accessed using the prefix defined, i.e. the prefix 's'. The first page that we are going to design is the `index.jsp`. The application, `generictags`, follows the standard directory structure similar to other Struts 2 based application. So, we are not discussing the role of files, like `web.xml`, `struts.xml`, `struts.properties` and `ApplicationResources.properties`. Assume that these files are created and placed at their usual location in the application.

Here's the code, given in Listing 6.1, for designing the `index.jsp` page (you can find `index.jsp` file in `Code\Chapter 6\generictags` folder in CD):

Listing 6.1: `index.jsp` code

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <div align="center">
        <h2><s:text name="app.welcome"/></h2>
        <br><br>
    </div>
    <s:date name="new java.util.Date()" format="%{getText('app.date.format')}"/>
    <br><br>
    <s:a href="yourcity.jsp">Using if, elseif and else tag</s:a>
    <br><br>
```

```
<s:url id="url" action="iterateAction"/>
<s:a href="#">Iteration</s:a>
<br><br>
<s:a href="person.jsp">User Info</s:a>
</body>
</html>
```

The first statement to be highlighted here in this JSP page is the use of taglib directive. The standard uri is set to /struts-tags and prefix="s" is used. The very first page created here uses a number of tags, which are as follows:

- <s:text/>
- <s:date/>
- <s:a/>, and
- <s:url/>

This page basically provides hyperlinks to further actions or JSP pages showing implementation of some other tags. The output of this page is shown in Figure 6.1, which is the first screen that appears when the application starts. The <s:a/> and <s:url/> tags is used to create hyperlinks shown in this page.

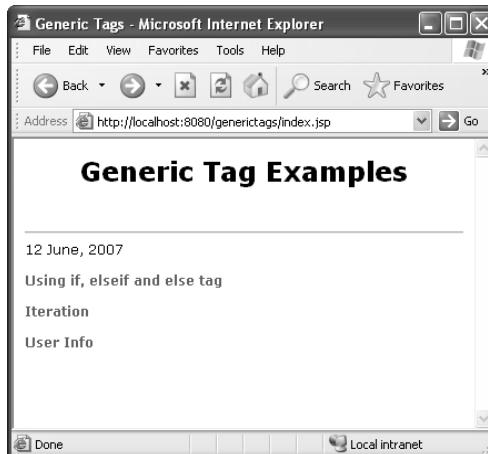


Figure 6.1: The index.jsp showing hyperlinks.

The tag <s:text/> is used to display internationalized messages from the resource bundle (ApplicationResources.properties) file for the corresponding key provided within the tag itself.

Here's the first <s:text /> tag used:

```
<s:text name="app.title"/>
```

This will display the message contained in the ApplicationResources.properties file for key app.title. The other key/value pairs, which are used in this application, is shown here. Update your copy of ApplicationResources.properties file with these key/value pairs:

```
# Resources for parameter 'ApplicationResources'
# Project generictags
app.title=Generic Tags
```

```
app.welcome=Generic Tag Examples
app.city>Select Your City.
app.date.format=dd MMMM, yyyy
```

Place the ApplicationResources.properties file according to the value set for struts.custom.i18n.resources key in struts.properties file placed in WEB-INF/classes folder. The value set for this application is as follows:

```
struts.custom.i18n.resources=ApplicationResources
```

Using **if**, **elseif** and **else** Tags

Earlier, JSP scriptlet was used to implement the conditional flow of execution in a JSP page. This approach can be cumbersome when it is implemented by the page designers who do not know much about Java coding. But now, with the help of **if**, **elseif** and **else** tags, we are capable of applying conditional execution of some code blocks in a JSP page. The concept of using these three tags is similar to the basic **if-elseif-else** constructs used in Java programming language.

The **else** and **elseif** tags cannot be used without using **if** tag. For a single **if** tag there may be one or more **elseif** tags and a single **else** tag. The **if** and **elseif** tags have an attribute **test**, which takes a Boolean expression and returns a value of true or false. The **if** and **elseif** blocks are executed when the test expression provided returns true. Let's make a JSP page implementing these three tags.

Here's the code, given in Listing 6.2, for yourcity.jsp (you can find yourcity.jsp file in Code\Chapter 6\generictags folder in CD):

Listing 6.2: yourcity.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Select Your City</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<s:form action="cityAction" method="post">
    <s:select key="app.city" name="city"
        list="#{'Delhi':'Delhi', 'Paris':'Paris', 'Pune':'Pune',
        'Colombo':'Colombo'}" />
    <s:submit/>
</s:form>

    <s:if test="city!=null">
        <s:if test="city=='Delhi'">
            <hr width="400" align="left">
            <b>Using if tag:</b> Delhi is Capital of India.
        </s:if>

        <hr width="400" align="left">
        <b>Using if and else tag:</b>
        <s:if test="city=='Paris'">
```

```
        Paris is Capital of France.  
</s:if>  
<s:else>  
        This is not the capital of France.  
</s:else>  
  
<hr width="400" align="left">  
<b>Using if elseif and else tag:</b>  
<s:if test="city=='Delhi'">  
        Capital of India.  
</s:if>  
<s:elseif test="city=='Pune'">  
        An Indian city.  
</s:elseif>  
<s:else>  
        A Non-Indian city.  
</s:else>  
</s:if>  
  
<br><br><s:a href="index.jsp">Back</s:a>  
</body>  
</html>
```

The page creates a form prompting you to select a city of your choice. The form on submission invokes an action `cityAction`. Rest of the logic is implemented using `if`, `elseif`, and `else` tag. Now we need an action class to process the submission of this form.

Here's the code, given in Listing 6.3, for the `CityAction` action class (you can find `CityAction.java` file in `Code\Chapter 6\generictags\WEB-INF\src\com\kogent\action` folder in CD):

Listing 6.3: CityAction.java

```
package com.kogent.action;  
  
import com.opensymphony.xwork2.ActionSupport;  
public class CityAction extends ActionSupport{  
    String city;  
    public String getCity() {  
        return city;  
    }  
    public void setCity(String city) {  
        this.city = city;  
    }  
    public String execute(){  
        return SUCCESS;  
    }  
}
```

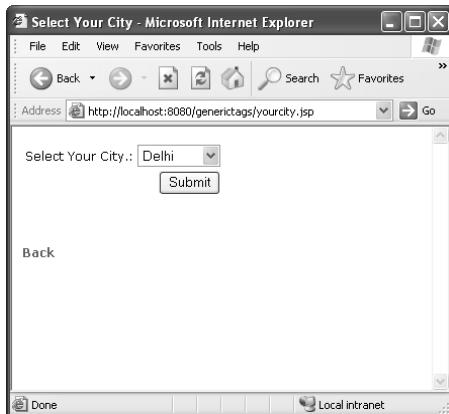
Compile the action class and place it in the `WEB-INF/classes/com/kogent/action` folder and provide action mapping for this action class.

Here's the action mapping, given in Listing 6.4, for this action class in `struts.xml` file (you can find `struts.xml` file in `Code\Chapter 6\generictags\WEB-INF\classes` folder in CD):

Listing 6.4: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="mydefault" extends="struts-default">
        <action name="cityAction" class="com.kogent.action.CityAction">
            <result>/yourcity.jsp</result>
        </action>
    </package>
</struts>
```

The action class returns ‘SUCCESS’ as result code. One result configured for this action mapping is `yourcity.jsp` itself. Now we are ready to access the page, `yourcity.jsp`, by clicking the hyperlink ‘Using if, elseif and else tag’ created in `index.jsp` and shown in Figure 6.1. When this page is executed for the first time, the code block enclosed using `<s:if test="city!=null"></s:if>` is not executed as there is yet no property with the name ‘city’ in the current value stack and the test expression returns false. The output of `yourcity.jsp` page is shown in Figure 6.2.

**Figure 6.2: Page yourcity.jsp with a form.**

Select the city of your choice and click on the ‘Submit’ button. The different `if`, `elseif`, and `if` tag blocks are executed according to the selected city. The three different constructs used here are as follows:

```
//using if tag
<s:if test="city=='Delhi'">
    <hr width="400" align="left">
    <b>Using if tag:</b> Delhi is Capital of India.
</s:if>
//Using if and else tag
<s:if test="city=='Paris'">
    Paris is Capital of France.
```

```
</s:if>
<s:else>
    This is not the capital of France.
</s:else>

//Using if elseif and else tags.
<s:if test="city=='Delhi'>
    Capital of India.
</s:if>
<s:elseif test="city=='Pune'">
    An Indian city.
</s:elseif>
<s:else>
    A Non-Indian city.
</s:else>
```

The output of `yourcity.jsp` page can change according to the city selected. For the city ‘Paris’ and ‘Delhi’, we have two different outputs, as shown in Figures 6.3 and 6.4.



Figure 6.3: Page `yourcity.jsp` when Paris is selected.

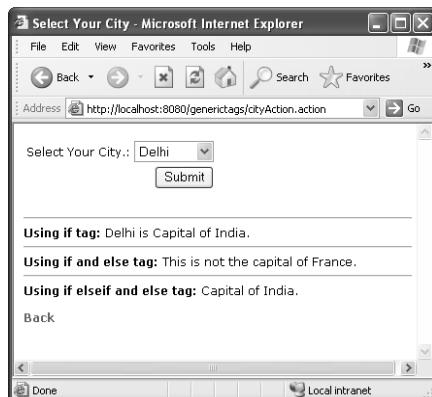


Figure 6.4: Page `yourcity.jsp` when Delhi is selected.

This is how we can use `if`, `elseif`, and `else` tags to customize the flow of execution.

Using Iteration Tags

It is very frequent that we need to iterate over some collection object, like `ArrayList`, etc. to display their content or to process some other logic over all the elements of the collection. The language constructs, like `while` and `for` loops, are, of course, helpful, but the iterator tag has changed the way iteration is implemented in JSP pages. Let's first create another action class, which adds some `ArrayList` object in the value stack which our JSP page can use to iterate over. The action class created here is `IterateAction`.

Here's the code, given in Listing 6.5, for this action class (you can find `IterateAction` file in `Code\Chapter 6\generictags\WEB-INF\src\com\kogent\action` folder in CD):

Listing 6.5: `IterateAction.java`

```

package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class IterateAction extends ActionSupport{
    ArrayList<String> fruits;
    ArrayList<String> cities;
    ArrayList<String> colors;

    public ArrayList<String> getCities() {
        return cities;
    }
    public ArrayList<String> getColors() {
        return colors;
    }
    public ArrayList<String> getFruits() {
        return fruits;
    }
    public String execute(){
        fruits=new ArrayList<String>();
        cities=new ArrayList<String>();
        colors=new ArrayList<String>();
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");

        cities.add("Delhi");
        cities.add("Mumbai");
        cities.add("Pune");
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");
        return SUCCESS;
    }
}

```

The `IterateAction` class populates three array lists (`fruits`, `cities`, and `colors`) with some string values. These array lists are the properties of this action class and have getter methods for them. The

getter method is required when the array lists are referred from JSP page using the iterator tag. Add the new action mapping with name="iterateAction" in struts.xml:

```
<struts>
    <include file="struts-default.xml"/>
    <package name="mydefault" extends="struts-default">
        <action>
        </action>

        <action name="iterateAction" class="com.kogent.action.IterateAction">
            <result>/iterator.jsp</result>
        </action>
    </package>
</struts>
```

The iterator.jsp page implements the iterator tag with some other tags useful for iteration operation.

Here's the code, given in Listing 6.6, for designing the iterator.jsp page before discussing the different types of implementation using append, merge, generator, and subset tags (you can find iterator.jsp file in Code\Chapter 6\generictags folder in CD):

Listing 6.6: iterator.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Iteration using Tags</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>

    <b>Using iterator tag:</b>
    <s:iterator status="stat" value="{23, 12, 34, 12, 15, 89}" >
        [ <s:property value="#stat.index" /> | <s:property value="top" /> ]
        <s:if test="#stat.last==false">, </s:if>
    </s:iterator>
    <br><br>

    <b>Using generator and iterator tag:</b>
    <s:generator val="%{'Mon,Tue,Wed,Thu,Fri,Sat,Sun'}"
        count="6"
        separator="," id="someid">
        <s:iterator>
            <s:property />
        </s:iterator>
    </s:generator>

    <br><br><b>Iterating over ArrayList:</b><br>
    ArrayList fruits:
    <s:iterator value="fruits">
        <s:property/>
```

```

</s:iterator>
<br>ArrayList cities:
<s:iterator value="cities">
    <s:property/>
</s:iterator>
<br>ArrayList colors:
<s:iterator value="colors">
    <s:property/>
</s:iterator>

<br><br><b>Appending Iterators:</b><br>
<s:append id="appendedItr">
    <s:param value="%{fruits}"/>
    <s:param value="%{cities}"/>
    <s:param value="%{colors}"/>
</s:append>
<s:iterator value="%{#appendedItr}">
    <s:property />,
</s:iterator>
<br><br><b>Merging Iterators:</b><br>
<s:merge id="mergedItr">
    <s:param value="%{fruits}"/>
    <s:param value="%{cities}"/>
    <s:param value="%{colors}"/>
</s:merge>
<s:iterator value="%{#mergedItr}">
    <s:property />,
</s:iterator>

<br><br><b>Using subset and iterator tag:</b><br>
With count=3:
<s:subset source="cities" count="3">
<s:iterator>
    <s:property />
</s:iterator>
</s:subset>
<br>With count=2:
<s:subset source="cities" count="2">
<s:iterator>
    <s:property />
</s:iterator>
</s:subset>
<br>With count=2 and start=1:
<s:subset source="cities" start="1" count="2">
<s:iterator>
    <s:property />
</s:iterator>
</s:subset>
<br><br><s:a href="index.jsp">Back</s:a>
</body>
</html>

```

The first iterator tag example used in this JSP page uses value attribute to define an array or integers which are iterated over. The other important attribute is status, which helps in getting index and checking for the last element in the collection.

Here's an example of iterator tag used in Listing 6.6 along with its output:

```
<s:iterator status="stat" value="{23, 12, 34, 12, 15, 89}">
    [ <s:property value="#stat.index" /> | <s:property value="top" /> ]
    <s:if test="#stat.last==false">, </s:if>
</s:iterator>

//Output
[ 0 | 23 ] , [ 1 | 12 ] , [ 2 | 34 ] , [ 3 | 12 ] , [ 4 | 15 ] , [ 5 | 89 ]
```

The next example is of the generator tag. It generates an iterator, which can further be iterated over using iterator tag. The attributes of a generator tag—val, count, separator—further makes it easy to use and define a collection of your own type.

Here's the code from Listing 6.6 and its output:

```
<s:generator val="%{'Mon,Tue,Wed,Thu,Fri,Sat,Sun'}"
    count="6"
    separator="," id="someid">
    <s:iterator>
        <s:property />
    </s:iterator>
</s:generator>

//Output:
Mon Tue Wed Thu Fri Sat
```

The next example is of iterating over some collection object in the value stack. Setting of attribute value="fruits" in <s:iterator> tag invokes getFruits(). Consequently, the iterator tag can access fruits ArrayList from the current value stack which has been populated by the IterateAction action class. The iteration displays all the elements of ArrayList fruits.

Here's the syntax and output:

```
<s:iterator value="fruits">
    <s:property/>
</s:iterator>

//Output
Apple Mango Orange
```

Appending and Merging Iterators

The iterators can be appended and merged to create a single iterator object which further can be iterated using the iterator tag. The append tag is used to append iterators. The iterators to be appended are supplied using the <s:param/> tags. The iterator tag can use the id provided in append tag to access the appended iterator.

Here's the syntax and output of using the append tag (from Listing 6.6):

```
<s:append id="appendedItr">
    <s:param value="%{fruits}" />
    <s:param value="%{cities}" />
    <s:param value="%{colors}" />
</s:append>

<s:iterator value="#{appendedItr}">
    <s:property />,
</s:iterator>
```

```
//Output
Apple, Mango, Orange, Delhi, Mumbai, Pune, Red, Green, Blue,
```

The output shows that all elements of the first appended iterator are displayed first before getting to the first element of second appended iterator. This is the only difference in append and merge tag. The merge tag merges the iterators to make a single iterable object.

Here's the syntax and output of merge tag with iterator tag:

```
<s:merge id="mergedItr">
    <s:param value="%{fruits}" />
    <s:param value="%{cities}" />
    <s:param value="%{colors}" />
</s:merge>
<s:iterator value="#{mergedItr}">
    <s:property />,
</s:iterator>
```

```
//Output
Apple, Delhi, Red, Mango, Mumbai, Green, Orange, Pune, Blue,
```

The last example that we'll use here is of the subset tag, which can iterate over the subset of a collection. We can fix the number of elements to be iterated using the count attribute and select the index to start with using the start attribute. Here's the syntax and output:

```
<s:subset source="cities" start="1" count="2">
    <s:iterator>
        <s:property />
    </s:iterator>
</s:subset>
```

```
//Output:
Mumbai Pune
```

These examples will help in understanding how the iterator tag and other tags, like generator, append, merge, and subset, have made implementation of iteration in JSP pages easy..

Click on the ‘Iteration’ hyperlink to see the output of the iterator.jsp page, as shown in Figure 6.5.

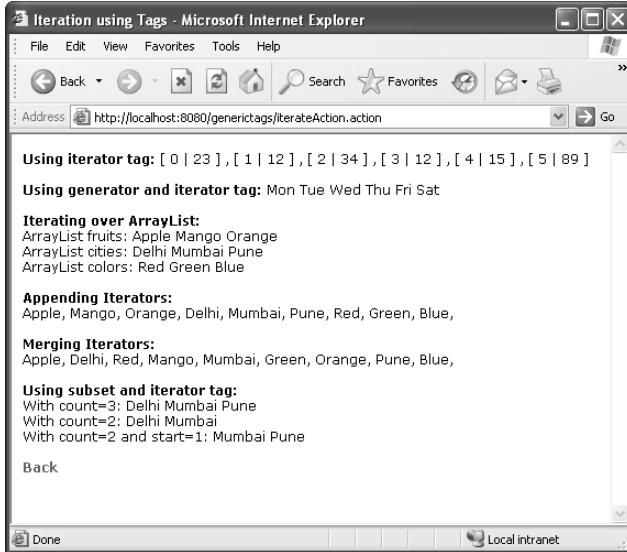


Figure 6.5: The iterator.jsp showing output of various iteration examples.

Using Other Data Tags

After implementing the Control tags with some Data tags in the previous sections, let's implement some of the most frequently-used Data tags which have not yet been implemented in this chapter.

Here's the code, given in Listing 6.7, for creating person.jsp (you can find person.jsp file in Code\Chapter 6\generictags folder in CD):

Listing 6.7: person.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <s:include value="header.jsp"/>
    <br><br><b>Running Action and its result</b><br><br>

    <s:action name="personAction" executeResult="true"/>

    <br><br><b>Using bean tag</b><br><br>
    <s:bean name="com.kogent.action.PersonAction">
        <s:param name="name">Kogent</s:param>
        <s:param name="address">Ansari Road, Dariyaganj</s:param>
        <s:param name="city">Delhi</s:param>
        User's Name: <s:property value="name"/><br><br>
        Address: <s:property value="address"/><br><br>
```

```

        City: <s:property value="city"/>
</s:bean>
<s:include value="footer.jsp"/>
</body>
</html>

```

The three important tags used in this JSP page are as follows:

- <s:include/>,
- <s:action/>, and
- <s:bean/>

The <s:include/> tag includes the output of other Servlet or JSP page. We have used <s:include/> tags at two places in person.jsp page (Listing 6.7)—one to included content of header.jsp and another to included content of footer.jsp page.

Here's the code, given in Listing 6.8, for creating header.jsp and footer.jsp (you can find header.jsp and footer.jsp files in Code\Chapter 6\generictags folder in CD):

Listing 6.8: header.jsp and footer.jsp

```

//header.jsp
<table align="center" bgcolor="#d5f0f9" width="100%">
<tr><td height="28" align="center"><b>User Info</b></td></tr>
</table>
<hr>
//footer.jsp
<hr>
<table align="center" bgcolor="#d5f0f9" width="100%">
<tr><td height="28" align="center">All rights are reserved with ABC
    Pvt. Ltd.</td></tr>
</table>

```

The next tag used is <s:action/>. This tag helps in invoking some action class and gives the option of rendering the result of action into the current JSP page. The <s:action/> tag uses the name attribute to set the action to be used. The attribute executeResult is set to true to assure the execution of results defined for this action in struts.xml file. The action mapping provided in struts.xml file for the corresponding name attribute set in <s:action/> tag is as follows:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="mydefault" extends="struts-default">
        .
        .
        .
        <action name="personAction" class="com.kogent.action.PersonAction">
            <result>/user.jsp</result>
        </action>
    </package>
</struts>

```

This needs the creation of com.kogent.action.PersonAction class and user.jsp page. The PersonAction class is an action class extending ActionSupport class with three fields, i.e. ‘name’, ‘address’, and ‘city’. Its execute() method sets the values for these fields. The class also has getter/setter methods for all these three fields.

Here’s the code, given in Listing 6.9, for creating PersonAction class (you can find PersonAction.java file in Code\Chapter 6\generictags\WEB-INF\src\com\kogent\action folder in CD):

Listing 6.9: PersonAction.java

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class PersonAction extends ActionSupport{
    String name;
    String address;
    String city;
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name; }
    public String execute(){
        setName("John");
        setAddress("C3, Lawrence Street");
        setCity("London");
        return SUCCESS;
    }
}
```

The execute() method of this action class returns ‘SUCCESS’, which in turn executes the user.jsp page. Here’s the code, given in Listing 6.10, for user.jsp page that displays the different property fields using the <s:property> tag (you can find user.jsp file in Code\Chapter 6\generictags folder in CD):

Listing 6.10: user.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
User's Name: <s:property value="name"/><br><br>
Address: <s:property value="address"/><br></br>
City: <s:property value="city"/>
```

The action class configured here is executed from `person.jsp` using the `<s:action />` tag. Hence the output of `user.jsp` page which is configured as the result of the action mapping provided for `PersonAction` action will be rendered and the content generated will be inserted in `person.jsp` page. See the output in Figure 6.6.

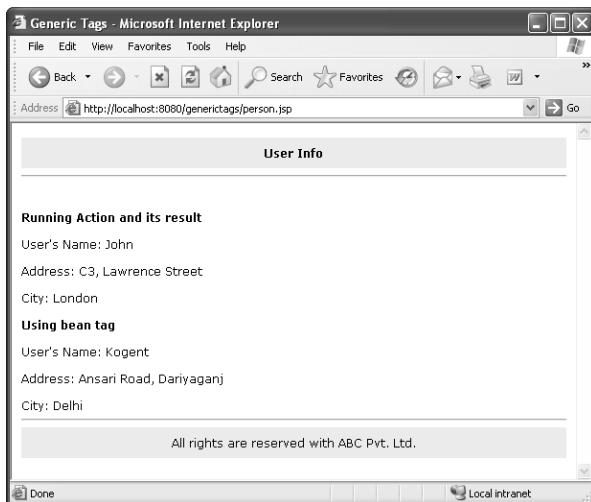


Figure 6.6: person.jsp showing use of `<s:action/>` and `<s:bean/>` tags.

Another useful tag is the `<s:bean/>`. This helps in instantiating a JavaBean like class and accessing its setter and getter methods to modify and access its fields. The `<s:bean name="com.kogent.action.PersonAction"/>` tag instantiates an object of `PersonAction` class and makes it available here. To set the field values and to obtain field values, we can use `<s:param/>` and `<s:property/>` tags as follows:

```
<s:param name="name">Kogent</s:param> // calls setName( "Kogent" ) method on the PersonAction object
```

```
<s:property value="name" /> //calls String getName( ) method on PersonAction object
```

You can see the output of the `<s:action/>` tag and `<s:bean>` tag used in `person.jsp` in Figure 6.6.

This chapter is focused on the detailed discussion of various Generic tags, subdivided into Control tags and Data tags, along with their attributes and syntax. This theme was further taken forward to the “Immediate Solutions” section making the implementation of these tags clear to the readers. The next chapter will discuss another set of Struts 2 tags, i.e. UI tags, which are used for designing components for user interfaces.



7

Designing User Interface in Struts 2

If you need an immediate solution to:

Using UI Tags

See page:

265

In Depth

Struts 2 comes with another set of tags, which can be used to design a JSP page with quality components for the required user interface, for example a form with some input fields and submit button. Tags are basically used to enhance the functionality and also encapsulate the technology behind a specific application. Using tags is always easy in comparison to writing lines of code. After discussing Generic tags in the previous chapter, we'll now start exploring another category of Struts tags, i.e. UI Tags. Generic tags are used to control the flow of a page. The flow control is handled by a set of tags available in the generic tag library, whereas the UI tags provides the way to create user interface (UI) components on the screen for the user. The UI tags are basically used to design the web page with high quality UI components. The UI components may include a form itself with other individual components contained in it, like text box, combo box, submit and reset buttons, radio buttons, checkboxes etc. There is a tag associated with each UI components which is used to create and place these components on the page. These UI components can be customized by setting different values for different attributes of the corresponding tags.

Implementing UI Tags

The UI tags provide the logic behind the flow of an execution. These tags involve the use of data in an application and focus on how the data is entered by the user and accepted by the application. These data can be retrieved from the actions, stack values, or from the data tags available in the Struts Framework. The UI tags represent the tags and generate the reusable HTML format. The UI tags work on the basis of templates. Templates again group together to form the themes, which are used to perform the actual operation. The UI tags are classified into the following two categories:

- Form tags
- Non-form tags

The Form UI tags are those tags that are used in designing the basic input form and are different from the other components in a form, like text box, combo box, text area, submit button, etc. The rest of the UI tags used in designing user interfaces, other than the form and its elements, are known as Non-form UI tags, like the tags used to display error messages.

The Form UI tags provide the output on the basis of the forms available. This includes displaying data to the user in the form of forms, text fields, text area and other input fields. All the UI tags that include the HTML tags are driven by templates. A template, in its simplest form, is bits of code that are used in HTML pages. The templates need to be loaded before they are used in the tags. Templates need a directory in which they are loaded and that particular directory is called the template directory. Templates are again combined to form the themes. The Form UI tags involve the following three themes:

- simple – The simplest theme for tags.
- xhtml – The xhtml; theme extends the simple theme
- ajax – The ajax in its simple form extends the xhtml theme. So we can say that the ajax theme can extend the simple theme.

The predefined themes in the UI tags can be used as it is defined. It cannot be modified or changed to any other form. For every tag, there is a Java class, which is executed to perform the specific function for which the tag is developed. Hence, there are different classes for different UI tags. All the form UI tags available in Struts 2.0 extend the base class `org.apache.struts2.views.jsp.ui.AbstractUITag` class.

Table 7.1 shows the protected fields defined in the `AbstractUIBean` class. All fields are of type `String`.

Table 7.1: Fields of AbstractUITag class				
accesskey	name	onkeypress	onselect	title
cssClass	onblur	onkeyup	required	tooltip
cssStyle	onchange	onmousedown	requiredposition	tooltipConfig
disabled	onclick	onmousemove	tabindex	value
key	ondblclick	onmouseout	template	
label	onfocus	onmouseover	templateDir	
labelPosition	onkeydown	onmouseup	theme	

These fields represent the various attributes, which is provided with the tags extending `AbstractUITag` as base class. The `AbstractUITag` also has setter methods to set all these fields associated with the tag. The values set for different attributes set, are used as arguments for these methods. Table 7.2 shows the methods used to set the fields associated with `AbstractUITag` class. The names of these methods are self-explanatory for the function they provide. For example, `setCssClass(String accesskey)` method sets the value for protected `String cssClass` field.

Table 7.2: Methods of AbstractUITag class			
<code>setAccesskey (String accesskey)</code>	<code>setOnblur (String onblur)</code>	<code>setOnmousedown (String onmousedown)</code>	<code>setTabindex (String tabindex)</code>
<code>setCssClass (String cssclass)</code>	<code>setOnchange (String onchange)</code>	<code>setOnmousemove (String onmousemove)</code>	<code>setTemplate (String template)</code>
<code>setCssStyle (String cssstyle)</code>	<code>setOnClick (String onclick)</code>	<code>setOnmouseout (String onmouseout)</code>	<code>setTemplateDir (String templatedir)</code>
<code>setDisabled (String disabled)</code>	<code>setOndblclick (String ondblclick)</code>	<code>setOnmouseover (String onmouseover)</code>	<code>setTheme (String theme)</code>

Table 7.2: Methods of AbstractUITag class

setKey(String key)	setOnfocus (String onfocus)	setOnmouseup (String onmouseup)	setTitle (String title)
setLabel (String label)	setOnkeydown (String onkeydown)	setOnselect (String onselect)	setTooltip(String tooltip)
setLabelposition(String labelposition)	setOnkeypress (String onkeypress)	setRequired (String required)	setTooltipConfig(String tooltipconfig)
setName (String name)	setOnkeyup (String onkeyup)	setRequiredposition(String requiredposition)	setValue (String value)

We have tags available to create UI components. Using these tags is preferred when designing a web page with UI components, like text boxes, combo-boxes etc. All UI components are represented by a specific Java class, which extends `org.apache.struts2.components.UIBean` which is a standard super class of all UI components. `UIBean` is an abstract class, which defines common Struts 2 and HTML component properties. We have different classes to represent different UI components like `TextField`, `Checkbox`, `TextArea`, `DatePicker`, and more. As the different UI component classes extend `UIBean` class, they have access to all the properties of `UIBean` class through the different methods inherited to set these properties. But not all the properties (attributes with tags) are rendered for all UI components. For example, the `tabindex` attribute is supported by the `Form` tag, but it is not rendered by any theme. The attributes of the base class, i.e. `UIBean` class, are divided into the following categories:

- ❑ General attribute
- ❑ JavaScript-related attribute
- ❑ Template-related attribute
- ❑ Tooltip-Related Attributes

All these common attributes have been discussed later in this chapter after going through all UI Tags for their functionality and syntax. Let us discuss different types of UI tags with all the functionality they are used for. All tags have been explained here with their syntax and different parameters to customize their behavior.

Form UI Tags

The Form UI tags allow the display of data in a simple and reusable format. These tags also include the HTML representation, which provides the feature of reusability. These tags are used to create a basic HTML form and other form components. Table 7.3 shows the tags enclosed within the Form UI tags.

Table 7.3: Form UI tags

form	checkboxlist
file	token

Table 7.3: Form UI tags	
password	label
textarea	hidden
checkbox	doubleselect
select	combobox
radio	submit
head	datetimepicker
optiontransferselect	optgroup
reset	textfield
updownselect	

These tags are described in the coming heads with all their syntaxes and the attributes, which control the behaviors of the UI components rendered using these tags.

The **form** Tag

The **form** tag is the most common tag and is used frequently when designing a form. The **form** tag acts as the container for other form UI components. It starts with a `<s:form>` and ends with a `</s:form>`. It allows the form to be submitted to the next page or to perform specific action against to the **form** tag. The **form** tag has attributes that control the submission of a form to some action, which process the data sent using input fields designed in a form. Table 7.4 shows these attributes.

Table 7.4: Parameters for form tag		
Parameter name	Data type	Description
action	String	It gives the name of the action to which the page is to be submitted; it does not take any .action extension with it
method	String	It defines the HTML methods attribute. The methods assigned for this parameter is GET and POST
name	String	It provides the name for the form element
namespace	String	It defines the namespace of the action. No default value is assigned to this parameter
target	String	It also takes string type of data. It defines the target of the HTML page
validate	Boolean	It defines the client/server side validation
windowState	String	It defines the state of the window in which the page, after the form submission, is displayed

The other attributes, which are common to all UI tags extending `AbstractUITag` class, are discussed later in the chapter.

The following snippet shows the use of a `form` tag, which creates a form:

```
<s:form action="actionName" method="post">
    . . .
</s:form>
```

The `file` Tag

This tag renders an HTML file input element. It prompts to choose any file location. Table 7.5 shows the parameters included in this tag.

Table 7.5: Parameters for file tag		
Parameter name	Data type	Description
accept	String	It is the HTML accept attribute to indicate accepted file mime types
id	String	It is used for referencing element; used as HTML id attribute
key	String	It sets the key (name, value, label) for the file component
name	String	It is used to set the name of element
size	Integer	It sets the HTML size attribute
value	String	It is used to preset the value of input element

The snippet for using the `file` tag is as follows:

```
<s:form>
    .
    .
    <s:file name="anyName" label="Upload File" accept="text/html" />
    .
    .
</s:form>
```

The `textfield` Tag

The `textfield` tag renders an HTML input field of type text. The `textfield` tag is meant to enter only limited characters. Table 7.6 shows the attributes of `textfield` tag.

Table 7.6: Parameters for textfield tag		
Parameter name	Data type	Description
id	String	It is used for referencing element and used as HTML id attribute

Table 7.6: Parameters for textfield tag		
Parameter name	Data type	Description
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element
maxlength	Integer	It is used to set HTML maxlength attribute
name	String	It sets the name of the element
readonly	Boolean	It can be set as true or false; decides whether the input field is readonly
size	Integer	It sets the HTML size attribute
value	String	It presets the value of input element

The following code snippet shows the use of <s:textfield> tag:

```
<s:form>
  ...
<s:textfield name="username" label="Username"/>

<s:form />
```

The **textarea** Tag

The **textarea** tag creates an input field to enter text, which supports multiple lines. The **textfield** tag is meant to enter only limited characters, whereas the **textarea** tag can insert larger texts with line breaks. Table 7.7 shows the parameters of this tag.

Table 7.7: Parameters for textarea tag		
Parameter name	Data type	Description
cols	Integer	It sets the HTML cols attribute
id	String	It is used for referencing element and used as HTML id attribute
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element
name	String	It sets name of the element
readonly	Boolean	It can set as true or false; decides whether the

Table 7.7: Parameters for textarea tag		
Parameter name	Data type	Description
		input field is readonly
rows	Integer	It sets the HTML rows attribute
value	String	It presets the value of input element
wrap	String	It sets the HTML wrap attribute

The following code snippet shows the syntax of <s:textarea> tag in your JSP page:

```
<s:form>
    .
    .
<s:textarea name="address" label="Address" cols="15" rows="2"/>
</s:form>
```

The **password** Tag

The password tag is similar to `textfield` tag as the class `org.apache.struts2.views.jsp.ui.PasswordTag` extends `org.apache.struts2.views.jsp.ui.TextFieldTag`. The value entered for the password field is not visible to the user. Table 7.8 shows the parameters of the `password` tag.

Table 7.8: Parameters for password tag		
Parameter name	Data type	Description
id	String	It is used as the HTML id attribute
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element
maxlength	Integer	It sets the HTML maxlength attribute
name	String	It sets the name of the element
readonly	Boolean	It can set as true or false; decides whether the input field is readonly
showPassword	Boolean	It specifies whether to show input
size	Integer	It sets the HTML size attribute
value	String	It presets the value of input element

The following code snippet shows the use of <s:password> tag in a JSP page:

```
<s: form>
  .
  .
<s:password name="password" label="Password"/>
  .
  .
</s: form>
```

The checkbox Tag

This tag renders a checkbox option on the page. The values of a checkbox tag need not always be a String. The types can be converted into Boolean. The checkbox tag often takes the values in Boolean format. Table 7.9 shows the parameters of the checkbox tag.

Table 7.9: Parameters for checkbox tag

Parameter name	Data type	Description
fieldValue	String	It is used to set the actual HTML value attribute of the checkbox
id	String	It is used for referencing elementl and used as HTML id attribute.
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element.
name	String	It sets name of the element.
value	String	It presets the value of input element

The following code snippet shows the use of <s:checkbox> tag:

```
<s: form>
  .
  .
<s:checkbox name="cricket" label="Cricket" fieldValue="true" />
<s:checkbox name="hockey" label="Hockey" fieldValue="true" />
<s:checkbox name="tennis" label="Tennis" fieldValue="true" />
</s: form>
```

The **checkboxlist** Tag

The checkboxlist tag is same as the radio tag, except that instead of allowing a user to select a single option, it enables the user to select none, one, many or all options. This is same as is provided by the select tag with multiple parameters set to true. But, instead of displaying a list of options, the checkboxlist renders a row of checkboxes to select and deselect. Table 7.10 shows the parameters of the checkboxlist tag.

Table 7.10: Parameters for checkboxlist tag

Parameter name	Data type	Description
<code>id</code>	<code>String</code>	It is used for referencing element and used as HTML id attribute.
<code>key</code>	<code>String</code>	It sets the key (name, value, label) for this particular component
<code>label</code>	<code>String</code>	It sets the string to be displayed as label with the element.
<code>list</code>	<code>String</code>	It is used to set an Iterable source to populate from. If the list is a Map (key, value), the Map key will become the option 'value' parameter and the Map value will become the option body
<code>listKey</code>	<code>String</code>	It is used to set the property of list objects to get field value from
<code>listValue</code>	<code>String</code>	It is used to set the property of list objects to get field content from
<code>name</code>	<code>String</code>	It sets the name of the element.

The following code snippet shows the use of `<s:checkboxlist>` tag:

```

<s:form>
  ...
<s:checkboxlist
    name="fruits"
    label="Select Your Fruits"
    list="{'Apple', 'Mango', 'Orange'}/>
</s:form>

```

The **select** Tag

This tag is used to create a list of options to select from. It renders an HTML input tag of type select. When its `multiple` parameter is set `true`, we can choose multiple options. Table 7.11 shows the parameters of the `select` tag.

Table 7.11: Parameters for select tag

Parameter name	Data type	Description
<code>headerKey</code>	<code>String</code>	It sets the key for the first item in list.
<code>headerValue</code>	<code>String</code>	It gives the value expression for the first item in the list
<code>id</code>	<code>String</code>	It is used as HTML id attribute.

Table 7.11: Parameters for select tag		
Parameter name	Data type	Description
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element.
list	String	It is used to set an Iterable source to populate from. If the list is a Map (key, value), the Map key will become the option 'value' parameter and the Map value will become the option body
listKey	String	It sets property of list objects to get field value from
listValue	String	It sets the property of list objects to get field content from
multiple	Boolean	This can be set as true or false. If set as true, it enables a multiple selections. The tag will pre-select multiple values if the values are passed as an Array (of appropriate types) via the value attribute.
name	String	It sets the name for element
size	Integer	It sets the size of the element box (number of elements to show)

The following code snippet shows the use of <s:select> tag:

```

<s:form>
    .
    .
    <s:select name="book"
        label="Select Book"
        list="{'Core Java', 'Struts', 'JavaScript'}"
        emptyOption="true"/>
</s: form>

```

The **radio** Tag

This tag is used for enabling users to select an option of their choice from the list of options provided in the form of radio options. Unlike **select** tag, we can select a single option out of the options provided. It renders a radio button input field. Table 7.12 shows the parameters defined for a **radio** tag.

Table 7.12: Parameters for radio tag

Parameter name	Data type	Description
id	String	It is used as HTML id attribute
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element
list	String	It is used to set an iterable source to populate from. If the list is a Map (key, value), the Map key will become the option 'value' parameter and the Map value will become the option body
listKey	String	It sets the property of list objects to get field value from
listValue	String	It sets the property of list objects to get field content from
name	String	It sets the name for element

The following code snippet shows the use of <s:radio> tag:

```

<s: form>
    .
    .
    <s:radio name="m_status"
        label="Marital status"
        list="{'Single', 'Married', 'Divorcee'}" />
</s: form>
</p>

```

The **combobox** Tag

The combobox tag extends the textfield tag. A combo box is used to choose any value from the pre-existing list of options. The user can also edit or change the values according to their need. Table 7.13 shows the attributes of the combobox tag.

Table 7.13: Parameters for combobox tag

Parameter name	Data type	Description
key	String	It sets the key (name, value, label) for this particular component
emptyOption	Boolean	It can be set to true or false to decide whether an empty option is to be inserted or not; default is false
headerKey	String	It sets the header key for the header option

Table 7.13: Parameters for combobox tag

Parameter name	Data type	Description
headerValue	String	It sets the header value for the header option
id	String	It is used for referencing element; used as HTML id attribute.
label	String	It sets the string to be displayed as label with the element.
list	String	It is used to set an iterable source to populate from. If this is missing, the select widget is simply not displayed
listKey	String	It sets the key used to retrieve the option key
listValue	String	It sets the value used to retrieve the option value
maxlength	Integer	It sets the HTML maxlength attribute
name	String	It sets the name for the element
readonly	Boolean	It can be set to true or false to decide whether the input is read-only or not.
size	Integer	It sets the HTML size attribute

The following code snippet shows the use of <s:combobox> tag:

```
<s:form>
    ...
<s:combobox
    name="language"
    label="Select Language"
    list="{'Java', 'C', 'C++'}"
    headerKey="-1"
    headerValue="---Select Your Language---"
    emptyOption="true"/>
</s: form>
```

The **token** Tag

The token tags are used to avoid the double form submission. The token name acts as the key value for the form. It places a hidden token with form so that the form is not re-submitted. The token tag is required if we are using TokenInterceptor or TokenSessionInterceptor. The parameters of the token tag are described in Table 7.14.

Table 7.14: Parameters for token tag

Parameter name	Data type	Description
id	String	It is used for referencing element and used as HTML id attribute.
key	String	It sets the key (name, value, label) for this particular component .
name	String	It sets the name for the element .
value	String	It presets the value of input element.

The following code snippet shows the use of `<s:token>` tag, we simply add a `<s:token/>` tag between a `<s:form></s:form>` tags:

```
<s:form>
    <s:token />
    ....
</s: form>
```

This tag places a hidden element, which contains a unique token.

The **label** Tag

The `label` tag is a unique tag and is used to label a field. It does not depend upon the value of the label. Table 7.15 shows the parameters defined for the `label` tag:

Table 7.15: Parameters for label tag

Parameter name	Data type	Description
for	String	It sets HTML for attribute
id	String	It is used for referencing element and used as HTML id attribute
key	String	It sets the key (name, value, label) for this particular component
label	String	It sets the string to be displayed as label with the element.
name	String	It sets the name for the element.

The following code snippet shows the use of `<s:label>` tag:

```
<s: form>
    ....
    <s:label label="Enter Your Name" required="true"/>
</s: form>
```

The **hidden** Tag

The hidden tag is used to create an input field, which is hidden from the user. This tag renders an HTML input element of type hidden. Table 7.16 shows the parameters for this tag.

Table 7.16: Parameters for hidden Tag

Parameter name	Data type	Description
id	String	It is used for referencing element and used as HTML id attribute.
key	String	It sets the key (name, value, label) for this particular component
name	String	It sets the name for the element
value	String	It presets the value of input element.

Sometimes, when we wish to pass some parameters from one HTML form to the action class, which is required for processing, and don't want the user to see it, we can use the hidden tag. The following code snippet shows the use of <s:hidden> tag:

```
<s:form>
  .
  .
  <s:hidden name="employee_id" value="some value" />
</s: form>
```

The **doubleselect** Tag

The doubleselect tag is an extension of the select tag. It is used both for selecting the list of items and to put them in a group. The user makes his choice from the group. The doubleselect encapsulates two tags together. The first tag is used for grouping and the second is used for the list. Table 7.17 shows the parameters for doubleselect tag.

Table 7.17: Parameters for doubleselect tag

Parameter name	Data type	Description
disabled	String	It sets the disabled attribute on the HTML element
doubleDisabled	String	It defines the use of the disabled attribute in the second list.
doubleEmptyOption	Boolean	It can be set to true or false. If set to true, the second list includes an empty option.
doubleHeaderKey	String	It defines the key value for the second list
doubleHeaderValue	String	It defines the value for the element in the second list

Table 7.17: Parameters for doubleselect tag

Parameter name	Data type	Description
doubleId	String	It is used to reference an element in the second list
doubleList	string	It defines the list of items for a user from which the user makes his option
doubleListkey	String	It sets the key for the second list
doubleListvalue	String	It gives the values of the second list
doubleMultiple	String	It decides whether multiple attributes can be taken in the second list or not
doubleName	String	It defines the name for the component
doubleSize	String	It defines the size of the second list
doubleValue	string	It defines the value for the complete doubleselect list
id	String	It is used to reference an element
key	String	This parameter sets the keys, like the name, value, or label for a component
label	String	It is used to label a specific element
list	String	It defines the list of items for a user from which the user makes his option
listkey	String	It is the property of the list element from where the field value of an element can be accessed
listvalue	String	It is the property of list element to get the content of the field value
name	String	It provides the name for an element
size	Integer	It defines the size of the element
value	String	It sets the value for an element

The following code snippet shows the use of <s:doubleselect> tag:

```
<s:form>
  ...
<s:doubleselect
  label="Category and items"
  list="{'Fruit', 'Animal'}"
  name="category"
  doubleName="item"
```

```
doubleList="top=='Fruit'?{'Apple','orange','Mango'}:{'Monkey','Tiger','Lio  
n'}"/>  
</s: form>
```

The **submit** Tag

The submit tag is used in order to submit a page for further execution. After the submission of the form, the control of the current page is sent to some action class where it is processed. The submit tag has three different types of uses:

- ❑ Input: It renders an HTML <input type="submit".....> tag.
- ❑ Image: It renders an HTML <input type="image".....> tag.
- ❑ Button: It renders an HTML <button type="submit".....> tag.

Table 7.18 shows the parameters of the submit tag.

Table 7.18: Parameters for submit tag

Parameter name	Data type	Description
action	String	It sets the action attribute
align	String	It sets the HTML align attribute
errorText	String	It is used to set the text to be displayed if an error occurs, while fetching the content
executeScripts	Boolean	It can be set to true or false to decide if the JavaScript code in the fetched content will be executed
formFilter	String	It is used to set the function name used to filter the fields of the form
formId	String	It is used to set the form id whose fields will be serialized and passed as parameters
handler	String	It is used to set the JavaScript function name that will make the request
href	String	It sets the URL to call to obtain the content Note: If used with Ajax context, the value must be set as an url tag value
id	String	It is used for referencing element and used as HTML id attribute
indicator	String	It sets the indicator
key	String	It sets the key (name, value, label) for this particular component

Table 7.18: Parameters for submit tag

Parameter name	Data type	Description
label	String	It sets the string to be displayed as label with the element
listenTopics	String	It sets the topic that will trigger the remote call
loadingText	String	It sets the text to be shown, while the content is being fetched
method	String	It sets the method attribute
name	String	It sets the name for the element
notifyTopics	String	It sets the topics that will be published when the remote call completes.
showErrorTransportText	Boolean	It sets whether errors will be shown
src	String	It is used to supply an image source for image type submit button.
type	String	It sets the type of submit to use. Valid values are input, button and image
value	String	It presets the value of input element.

The following code snippet shows the use of <s:submit> tag:

```
<S: form>
  .
  .
<s:submit type="image" src="images/sign_in.jpg"/>
  Button Submit
<s:submit type="button" value="Login"/>
</s: form>
```

The **head** Tag

This tag renders the head part of an HTML file. It is used when your page contains some themes. If your page has AJAX components integrated, then using this tag can set the theme to ajax. This can be done by using <s:head theme="ajax" />. Table 7.19 shows the parameters for head tag.

Table 7.19: Parameters for head tag

Parameter name	Data type	Description
theme	String	It sets the theme to use for rendering the element

Table 7.19: Parameters for head tag		
Parameter name	Data type	Description
id	String	It is used to reference an element
key	String	It sets the keys, like the name, value or label for a component
name	String	It provides the name for an element
value	String	It sets the value for an element

The following code snippet shows the use of <s:head> tag:

```
<html>
<head>
<s:head theme="ajax" />
</head>
<s: form>
<s:datepicker name="datetime" label="Birthday Date(yyyy-MM-dd)"/> . .
</s: form>
</p>
```

The **datepicker** Tag

This tag is used for picking date and time from a dropdown container. The Date attributes are passed in the 'RFC 3339' format. The format supported by this component are as follows:

- #dd – It displays date in two-digit format.
- #d – It displays date in one-digit format. If not possible, then use two-digit format.
- #MM – It displays month in two-digit format.
- #M – It displays month in one-digit format. If not possible, then use two-digit format.
- #yyyy – It displays year in four-digit format.
- #yy – It displays year in two-digit format.
- #y – It displays year in one-digit format.

Table 7.20 shows the parameters for head tag.

Table 7.20: Parameters for datetpicker tag		
Parameter name	Data type	Description
disabled	String	It sets the disabled attribute on the HTML element
displayFormat	String	It sets a pattern for visual display of formatted date
displayWeeks	Integer	It sets the total number of weeks to be displayed; default value is 6

Table 7.20: Parameters for datetimepicker tag

Parameter name	Data type	Description
id	String	It is used to reference an element.
key	String	This parameter sets the keys like the name, value or label for a component
name	String	It provides the name for an element
value	String	It sets the value for an element
theme	String	It specifies the theme to use for rendering the element

The following code snippet shows the use of `<s:datetimepicker>` tag:

```

<head>
<s:head theme="ajax" />
</head>
<s:form>
<s:datetimepicker name="dob" label="Date of Birth"/>
<s:datetimepicker name="dot" type="time" label="Time of Birth"/>
</s:form>

```

The **optiontransferselect** Tag

This tag creates an option transfer select component, which renders two list boxes to select from. Further we can enable the options to transfer items from one list box to another and vice versa. We can also move the items in the list boxes to adjust their order in the list. The attributes for this tag help in customizing the behavior of the set of list boxes. Table 7.21 shows the parameters for this tag.

Table 7.21: Parameters for optiontransferselect tag

Parameter name	Data type	Description
addAllToLeftLabel	String	It sets Add To Left button label
addAllToRightLabel	String	It sets Add All To Right button label
addToLeftLabel	String	It sets Add To Left button label
addToRightLabel	String	It sets Add To Right button label
allowAddAllToLeft	String	It can be set to true to enable Add All To Left button
allowAddAllToRight	String	It can be set to true to enable Add All To Right button

Table 7.21: Parameters for optiontransferselect tag

Parameter name	Data type	Description
allowAddToLeft	String	It can be set to true to enable Add To Left button
allowAddToRight	String	It can be set to true to enable Add To Right button
allowSelectAll	String	It can be set to true to enable Select All button
allowUpDownOnLeft	String	It can be set to true to enable up / down on the left side
allowUpDownOnRight	String	It can be set to true to enable up / down on the right side
buttonCssClass	String	It sets buttons css class.
buttonCssStyle	String	It sets button css style.
cssClass	String	It sets the css class to use for element
cssStyle	String	It sets the css style definitions for element to use
disabled	String	It sets the html disabled attribute on rendered html element
doubleAccesskey	String	It sets the html accesskey attribute
doubleCssClass	String	It sets the css class for the second list
doubleCssStyle	String	It sets the css style for the second list
doubleDisabled	String	It decides if a disable attribute should be added to the second list
doubleEmptyOption	String	It decides if the second list will add an empty option
doubleHeaderKey	String	It sets the header key for the second list
doubleHeaderValue	String	It sets the header value for the second list
doubleId	String	It sets id of the second list
doubleList	String	It sets the second iterable source to populate from
doubleListKey	String	It sets the list key of the second attribute

Table 7.21: Parameters for optiontransferselect tag

Parameter name	Data type	Description
doubleListValue	String	It sets the value expression to use for second list.
doubleMultiple	String	It is used to decide whether multiple attribute should be set on the second list
doubleName	String	It is used to set the name for complete component
doubleSize	String	It sets the size attribute of the second list
doubleValue	String	It sets value expression for complete component
emptyOption	Boolean	It is used to decide if an empty option is to be inserted in the second list
formName	String	It sets the form name this component resides in and populates to
headerKey	String	It sets the header key of the second list.
headerValue	String	It sets the header value of the second list
id	String	It sets id for referencing element. For UI and Form tags it will be used as HTML id attribute
key	String	It sets the key (name, value, label) for this particular component
leftDownLabel	String	It can be set to enable down label for the left side
leftTitle	String	It sets left title
leftUpLabel	String	It sets up label for the left side
list	String	It is used to set an Iterable source to populate from. If the list is a Map (key, value), the Map key will become the option 'value' parameter and the Map value will become the option body
listKey	String	It sets the property of list objects to get field value from
listValue	String	It sets the property of list objects to get field content from

Table 7.21: Parameters for optiontransferselect tag		
Parameter name	Data type	Description
multiple.	String	It can be set to enable multiple selection. The tag will pre-select multiple values if the values are passed as an Array (of appropriate types) via the value attribute
name	String	It sets the name for element
rightDownLabel	String	It sets down label for the left side
rightTitle	String	It sets right title
rightUpLabel	String	It sets up label for the right side
selectAllLabel	String	It sets Select All button label
size	Integer	It sets the size of the element box (number of elements to show)

The following code snippet shows the use of <s:optiontransferselect> tag:

```
<s:form>
<s:optiontransferselect
    name="booklist"
    label="Select Books"
    list="{'Book 1','Book 2','Book 3', 'Book 4','Book 5'}"
    size="5"
    doubleName="books"
    doubleList="{}"
    doubleSize="5"
    leftTitle="Books"
    rightTitle="Your Books"
    allowUpDownOnLeft="false"
    allowUpDownOnRight="false"
    allowSelectAll="false"

    allowAddAllToLeft="false"
    allowAddAllToRight="false"
    addToRightLabel="">><"/>
    addToLeftLabel="<>"/>
</s:form>
```

The **optgroup** Tag

This tag provides an option to select a component. It is always used within the `select` tag. The options are classified under some headings and you can select multiple options too. This tag groups the set of options into a single category. Table 7.22 shows the parameters for `optgroup` tag.

Table 7.22: Parameters for optgroup tag

Parameter name	Data type	Description
id	String	It is used to reference an element.
disabled	String	It sets the disabled attribute on the HTML element
list	String	It provides a list of elements for first list
listkey	String	It is the property of the list element from where the field value of an element can be accessed
listvalue	String	It is the property of list element to get the content of the field value

The following code snippet shows the use of <s:optgroup> tag:

```

<s:form>
<s:select
    label="Select Language"
    name="language"
    list="#{'C':'C', 'C++':'C++'}">
    <s:optgroup
        label="Java"
        list="#{'Core Java':'Core Java', 'Groovy':'Groovy'}" />
    <s:optgroup
        label=".Net"
        list="#{'VB':'VB', 'ASP':'ASP', 'C#':'C#'}" />
</s:select>

```

The **updownselect** Tag

This tag creates a Select component with buttons to move the elements in the select components up and down. After the form submission, its elements are submitted in the order they are arranged. Table 7.23 shows the parameters for this tag.

Table 7.23: Parameters for updownselect tag

Parameter name	Data type	Description
id	String	It is used to reference an element
key	String	This parameter sets the keys, like the name, value or label for a component
name	String	It provides the name for an element
list	String	It provides list of elements for first list
listkey	String	It is the property of the list element from where the field value of an element can be accessed

Table 7.23: Parameters for updownselect tag		
Parameter name	Data type	Description
listvalue	String	It is the property of list element to get the content of the field value
headerKey	String	It sets the key for the first item in the list
headerValue	String	It sets the value for the first item in the list

The following code snippet shows the syntax of <s:updownselect /> tag:

```
<s:form>
    .
    .
    <s:updownselect label="Order in your order"
        list="#{'book1':'Book 1','book2':'Book 2','book3':'Book 3'}"
        name="books"
        headerKey="-1"
        headerValue="--- Please order items---"/>
</s:form>
```

Non-Form UI Tags

We have seen a number of Form UI tags used to create various components, which are used in a simple form created in a web page. There is another category of UI Tags which are referred to as the Non-form UI tags. The UI tags used for other than designing form components are known as Non-form UI tags. Though the UI tags are used along with the HTML tags, they have some different functionalities. The Non-form UI tags are used to display the output texts, like action-level messages or errors, and field-level errors, etc. These tags also create some other UI components, like tabbed panel, table, etc. It means that a Non-form UI tag does not take the attributes of the Form tag to generate the output.

The Non-form UI tags are also grouped under certain templates. The templates are again combined to form a theme. The themes available for the Non-form UI tags are same as for the Form UI tags. All the Non-form tags must extend one of these themes to draw the output of the page. Table 7.24 lists the tags available in the Non-form UI tags category.

Table 7.24: Non-form UI tags		
actionerror	div	tabbedpanel
actionmessage	fielderror	tree
component	table	treenode

These Non-form tags are used for creating UI components, like tabs, div, tables, and trees. In addition, these tags include tags to display action errors, action messages, and field errors. The Non-form UI tags are used to display the output to the user without using the forms. It does not take the input from a Form tag to generate the output. Let's understand the Non-form tags, along with their parameters.

The **actionerror** Tag

The `actionerror` tag is used to draw the errors in a component. It specifies the errors, which are related to action and displays the errors that are set in the action class using `addActionError()` method. The class for this tag `org.apache.struts2.views.jsp.ui.ActionErrorTag` extends the `org.apache.struts2.views.jsp.ui.AbstractUITag` tag and all the basic attributes that are available with this tag, but most of them are not rendered.

The following code snippet shows the use of the `actionerror` tag in a UI component:

```
<s:actionerror/>
<s:form>
. . .
</s:form>
```

The **actionmessage** Tag

This tag displays the message associated with an action. It displays the message that are set in the action class using the `addActionMessage()` method. All attributes extended from `AbstractUITag` can be used with this tag also, but most of them do not render.

The following code snippet shows the use of the `actionmessage` tag in a UI component:

```
<s:actionmessage/>
<s:form . . .>
. . .
</s:form>
```

The **fieldError** Tag

The use of the `actionerror` tag displays all the errors in a form, whereas the use of a `fielderror` tag displays the error related to a particular field. This tag displays all errors added using the `addFieldError()` method in the action. All basic attributes extended from `AbstractUITag` class are available with this tag.

The following snippet shows the use of the `fielderror` tag in a UI component:

```
<s:fielderror/>
<s:form>
. . .
</s:form>
```

The **component** Tag

The `component` tag is used to build custom UI tags. It is a three state configuration. This means that the `component` tag includes the HTML tags, Java scripts, and the custom tags. It renders a custom UI widget using specified templates. We can pass additional objects using the `param` tag. It inherits all the attributes from the `AbstractUITag` class. Remember to put the JSP template in the webapp itself not in the class path of the application, like Velocity and Freemaker template. The following code snippet shows the use of the `component` tag:

```
<s:component template="path to some jsp page">
<s:param name="key1" value="value1"/>
<s:param name="key2" value="value2"/>
</s:component>
```

The **div** Tag

This tag is used in the AJAX theme and makes remote calls to the page, which is included in the current page. Refreshing the entire page is not required, as it provides remote calls from the current page to update a section of content. It creates a HTML <DIV /> element. This obtains its content through a remote XMLHttpRequest call using Dojo Framework. Though the basic parent class here is AbstractUITag, the class for this tag org.apache.struts2.views.jsp.ui.DivTag extends the org.apache.struts2.views.jsp.ui.AbstractRemoteCallUITag class.

The **table** Tag

The table tag displays the contents of a table used in a form. The class org.apache.struts2.views.jsp.ui.table.WebTableTag extends the org.apache.struts2.views.jsp.ui.ComponentTag class. In addition to the attributes inherited from AbstractUITag and ComponentTag, this tag has some new attributes, like sortColumn (integer index of column to sort data), sortOrder (set sort order like NONE, ASC and DESC) and sortable (whether the table is sortable).

The **tabbedPanel** Tag

The tabbedpanel tag is an AJAX specification tag used in UI components. The tags enclosed with tabbedpanel tag may take local content or remote content. The additional attributes for this tag are

- ❑ closeButton – It sets the value to decide where the close button will be placed, possible values are tab and pane.
- ❑ doLayout – It can be set to true or false. If doLayout is false, the tab container's height equals the height of the currently selected tab.
- ❑ openTemplate – It sets template to use for opening the rendered html.
- ❑ selectedTab – It is set with the id of the tab that will be selected by default.

Here's the sample use of tabbedPanel tag with div tag:

```
<s:tabbedPanel
    id="one">

    <s:div id="two"
        label=""
        theme=ajax
        labelposition="top" >
        This is the first pane<br/>
        <s:form>
        <s:textfield name=" " label=" "/><br/>
        <s:textfield name="tt2" label="Test Text2"/>
        </s:form>
    </s:div>
```

```
<s:div id="three" label="remote" theme="ajax" href="/AjaxTest.action" >
    This is the remote tab
</s:div>
</s:tabbedPanel>
```

The **tree** Tag

The **tree** tag is used with the AJAX support and renders a tree-like structure where nodes can be clicked to expand and collapse. Table 7.25 shows the parameters of the **tree** tag.

Parameter name	Data type	Description
blankIconSrc	String	It defines the source of a blank icon
childCollectionProperty	String	It describes the properties of a child
iconHeight	String	It specifies the height of the icon. The default value for this parameter is 18px
iconWidth	string	It specifies the height of the icon. The default value for iconWidth is 19px
id	String	It is used to reference an element
key	String	This parameter sets the keys, like the name, value, or label for a component
name	String	It provides the name for an element
nodeIdProperty	String	It defines the property of the node id for the tree
nodeTitleProperty	String	It defines the title for the tree
rootNode	String	It defines the properties of the root node
showGrid	Boolean	It is set to enable displaying of the grid; default value is set to true
showRootGrid	String	It defines the property of the root grid; default value is set to true
toggle	String	It defines the toggle property
toggleDuration	string	It sets the duration in milliseconds; default value is 150
treeCollapsedTopic	String	It defines the treeCollapsedTopic's property
treeExpandedTopic	String	It defines the treeExpandedTopic's property

Table 7.25: Parameters for the tree tag

Parameter name	Data type	Description
treeSelectedTopic	string	It defines the treeSelectedTopic's property
value	String	It sets the value for an element

The following code snippet shows the use of the tree tag with treenode tag:

```
<s:tree id="...">  
    <label="...">  
        <s:treenode id="...">  
            <label="..." />  
        <s:treenode id="...">  
            <label="..." />  
            <s:treenode  
                id="..."  
                <label="..." />  
            <s:treenode id="...">  
                <label="..." />  
  
        </s:treenode>  
        <s:treenode  
            id="..."  
            <label="..." />  
    </s:tree>
```

The **treenode** Tag

The treenode tag is also used in AJAX components. The properties of the treenode tag depend upon the creation of the tree. The tree can be created both statically and dynamically.

When the tree is created statically, it must satisfy the following two properties:

- id**—It returns the id associated with the particular tree node
- title**—It returns the label to be displayed for the tree

The properties of a dynamically created tree are as follows:

- rootNode**—The node from where the tree is sub divided. It is also called as the parent node.
- nodeIdProperty**—The properties related to the current node id
- nodeTitleProperty**—The properties of the current nodes title
- childCollectionProperty**—The properties of the root node's subsequent children

This concludes the discussion over different UI Tags available in Struts 2. All UI tags have been discussed with their functionality and different parameters which can be set with different values to customize behavior of a given tag.

Discussing Common Attributes in UI Tags

All UI components extend `org.apache.struts2.components.UIBean` class and hence all UI components can be set for some common attributes. Similarly all UI tag classes extend `org.apache.struts2.views.jsp.ui.AbstractUITag` class so all UI tags have some common set of attributes to be used. We can set different attributes of a given Tag class by using parameters with the Tag with same name. We have already described different parameters which can be used with different UI tags, but in addition we are now discussing some common attributes of a Tag class which can be set by giving a same name parameter with tag. For example to set label attribute, we use label parameter with the tag. All common attributes have been discussed here. These tags has been categorized as General, JavaScript-Related, Template-Related and Tooltip Related attributes. They all have been explained here.

General Attributes

The general attributes of the Form tags include some tag attributes, which are general in use and can be seen using with different tags frequently. Table 7.26 shows the attributes coming under this category.

Table 7.26: General attributes of UIBean class.

Attributes	Description
<code>cssClass</code>	This attribute extends the simple theme as defined by the Form UI tag. The data type specified for this tag is of string type. It defines the HTML class attribute.
<code>cssStyle</code>	It extends the simple theme. It encapsulates the string type of data. It defines the HTML style attribute.
<code>title</code>	It takes string type of data. It defines the HTML title attribute.
<code>disabled</code>	It defines the HTML disabled attribute.
<code>label</code>	It defines the HTML label attribute.
<code>labelPosition</code>	It defines position of the label for the UI components, which can be top or left. The default position is left.
<code>requiredPosition</code>	It defines the required label position of the component. The required label position of a form can be left or right. The default required position is right. Use with required attribute only.
<code>name</code>	It defines the HTML name attribute for the UI component.
<code>required</code>	It also adds a '*' to the label of the element. Its value may be true or false.
<code>tabIndex</code>	It defines the HTML <code>tabIndex</code> attribute.
<code>value</code>	It defines the value of the component.

In case of tags, the name and value attributes for a specific tag are related to each other. The name attribute of a tag represents the name given to the tag for identification when the tag is submitted to perform its action. The value attribute of a tag represents the value that is bound to the name when the form is submitted. All the form UI component classes extend the `UIBean` class. So the bean class must provide identification for the tag, when the tag is submitted. This identification is done by the name associated with the tags reference. The name of the Form tag gives a map to the simple JavaBean class associated with the form's component. The value for the name of the tag is set by invoking the properties of the JavaBeans. After the submission of the form, the form's id element is used to access the properties of the JavaBean. The value for the name attribute will be set from the properties of the JavaBean. The following snippet shows the use of a name attribute along with the value attribute:

```
<s:form action="someAction">  
    <s:textfield label="User Name" name="username" value="kogent"/>  
    . . .  
    . . .  
</s:form>
```

In the preceding code snippet, the `username` is a JavaBean, which is used to encapsulate the properties of the UI component rendered through this tag. The bean is invoked to obtain the values of the required name attributes. The `value` attribute for a specified tag is optional. When the tag is submitted, the beans for the tag is accessed implicitly and the name attributes get the values automatically.

JavaScript-Related Attributes

JavaScripts are used to distinguish the feature of specific actions used to build a web page. The attributes specified for the JavaScript-related attributes have a specified theme named as the simple theme. Table 7.27 shows all the attributes specified for this category.

Table 7.27: JavaScript-related attributes of `UIBean` class

Attributes	Description
<code>onclick</code>	It represents the HTML JavaScript <code>onclick</code> attribute
<code>ondbclick</code>	It represents the HTM JavaScript <code>ondbclick</code> attribute
<code>onmousedown</code>	It represents the HTML JavaScript <code>onmousedown</code> attribute
<code>onmouseup</code>	It represents the HTML JavaScript <code>onmouseup</code> attribute
<code>onmouseover</code>	It represents the HTML JavaScript <code>onmouseover</code> attribute
<code>onmouseout</code>	It represents the HTML JavaScript <code>onmouseout</code> attribute
<code>onfocus</code>	It represents the HTML JavaScript <code>onfocus</code> attribute
<code>onblur</code>	It represents the HTML JavaScript <code>onblur</code> attribute
<code>onkeypress</code>	It represents the HTML JavaScript <code>onkeypress</code> attribute

Table 7.27: JavaScript-related attributes of UIBean class

Attributes	Description
onkeyup	It represents the HTML JavaScript onkeyup attribute
onkeydown	It represents the HTML JavaScript onkeydown attribute
onselect	It represents the HTML JavaScript onselect attribute
onchange	It represents the HTML JavaScript onchange attribute

Template-Related Attributes

Table 7.28 shows the attributes coming under this category.

Table 7.28: Template-related attributes of UIBean class

Attributes	Description
templateDir	The attributes defines template directory for the template being used.
theme	It defines the theme name of the tag
template	It defines the name of the template

Tooltip-Related Attributes

In addition to Generic, JavaScript and Template-related attributes, another type of attribute set defined in the Form UI tag is the Tooltip-related attributes.

Table 7.29 shows the attributes coming under this category.

Table 7.29: Tooltip-related attributes of UIBean class

Attributes	Description
tooltip	The tooltip attribute takes the string type of data. The default value for this attribute is set to none. This attribute sets the tooltip of a particular form component.
jsTooltipEnabled	This attribute also takes the string type of data and the default value of this attribute is set to false. This attribute is used to enable the JavaScript tooltip of a form's component.
tooltipIcon	It also takes the string type of data and the default value of this attribute is set to /struts/static/tooltip/tooltip.gif. It represents the url for the tooltip icon specified for the form's component.

Table 7.29: Tooltip-related attributes of UIBean class

Attributes	Description
tooltipDelay	It takes the string values and the default value is set to 500. The default representation is based in milliseconds. After the timeout of the specified time the tooltip performs its action. The behavior of a tooltip is same as in the Operating system's tooltip.
key	The key attribute extends the simple theme. The default value is set to strings. It defines the name of the property of the input fields associated with the form's action. It automatically handles the name, value, and the label of the form's component.

All common attributes have been discussed here and hence we have not provided all these attributes when discussing various UI tags. The only attributes which have been mentioned there are those which are required and are specific for that particular tag. Hence the attribute list given with each tag discussed earlier in the chapter does not contain all the attributes supported by that tag.

Let's move to the "Immediate Solutions" section to build an application, which uses all Form UI tags.

Immediate Solutions

Using UI Tags

The application, which we'll build here, will use all the Form UI tags as well as some Non-form UI tags. It will prompt the user to fill some information through some user interface designed using UI tags in the form of two JSP pages, containing forms with input fields. The basic directory structure of this application will be similar to the other simple Struts 2 Web applications created earlier. The JSP files using UI tags and the action class processing the data will also be discussed here.

NOTE

You can directly copy `struts.properties`, `struts.xml`, `ApplicationResources.properties`, and one style sheet `mystyle.css` from `Code\Chapter 7\uitags\WEB-INF` folder in the CD provided with this book and place them into your Web application accordingly. See the code listings provided here for the user of different UI tags and find how the user of different attributes changes the behavior of the rendered UI component to make a web page more sophisticated, both in looks and in functionality.

Developing index.jsp

The first page of our application is `index.jsp`. It provides two links which can be used by the user to navigate to two different pages to input personal and general information. These pages are `personal_info.jsp` and `general_info.jsp`.

Here's the code, given in Listing 7.1, for `index.jsp` (you can find `index.jsp` file in `Code\Chapter 7\uitags` folder in CD):

Listing 7.1: `index.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>

<html>
<head>
<title>Struts 2 UI Tags</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2>Using UI Tags</h2>
<s:a href="personal_info.jsp">Personal Information</s:a><br><br>
<s:a href="general_info.jsp">General Information</s:a>
</body>
</html>
```

Listing 7.1 provides the links for other JSP pages. The user view of this listing is the simple HTML code generated by the user. The output of the `index.jsp` page is shown in Figure 7.1.



Figure 7.1: The index.jsp page showing links to navigate.

Developing personal_info.jsp

The `personal_info.jsp` page gives a form with input fields, like ‘User Name’, ‘Password’, ‘Name’, ‘Date of Birth’, ‘Address’, etc. When the user hits the link labeled ‘Personal Information’, the `personal_info.jsp` page appears.

Here’s the code, given in Listing 7.2, for the `personal_info.jsp` page (you can find `personal_info.jsp` file in `Code\Chapter 7\uitags` folder in CD):

Listing 7.2: `personal_info.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Struts 2 UI Tags</title >
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="mystyle.css" type="text/css" />
<s:head theme="simple"/>
</head>

<body>
<h2 align="center">Enter Personal Information</h2>
<table align="center" width="600" >
<tr>
<td bgcolor="#fbefed" align="center" >
<s:actionerror/>

<s:form action="personalAction" method="post" name="personalform"
cssStyle="color:#7a3d3d">

<s:textfield name="username" label="User Name" size="15" cssClass="fieldtext"/>
```

```

<s:password name="password" label="Password" size="15" cssClass="fieldtext"/>

<s:textfield name="fullname" label="Name" size="25" cssClass="fieldtext"/>

<s:datetimepicker name="dob" label="Date of Birth" adjustweeks="true"
displayFormat="dd MMMM, yyyy" toggleType="wipr"/>

<s:textarea name="address" label="Address" rows="4" cols="20"
cssClass="fieldtext"/>

<s:doubleselect name="country"
label="Select Country and City"
list="{'Australia', 'India'}"
doubleName="city"
doubleList="top == 'Australia'? {'Sydney', 'Melbourne', 'Brisbane', 'Perth'}
: {'Delhi', 'Mumbai', 'Chennai', 'Kolkata'}"
cssClass="fieldtext" doubleCssClass="fieldtext"/>

<s:select name="language" label="Preferred Language(s)" list="{'English',
'Hindi', 'French', 'German'}" cssClass="fieldtext"/>

<s:radio name="mstatus" label="Marital Status" list="{'Single', 'Married',
'Divorced'}" value="Single" />

<s:checkboxlist name="skill" label="Your Interest" list="{'Programming',
'Testing', 'Research', 'Web Designing'}"/>

<s:submit value="Submit" align="center"/>

</s:form>
<s:a href="index.jsp">Back</s:a>
</td>
</tr>
</table>
</body>
</html>

```

Listing 7.2 contains many UI tags, like `textfield`, `password`, `textarea`, `select`, `datetimepicker` and others. The first UI tag is `<s:head>`. This tag specifies the theme that is applied on this page. Its value can be a simple, `xhtml` or `ajax`. Figure 7.2 shows the output of the various Form UI tags.

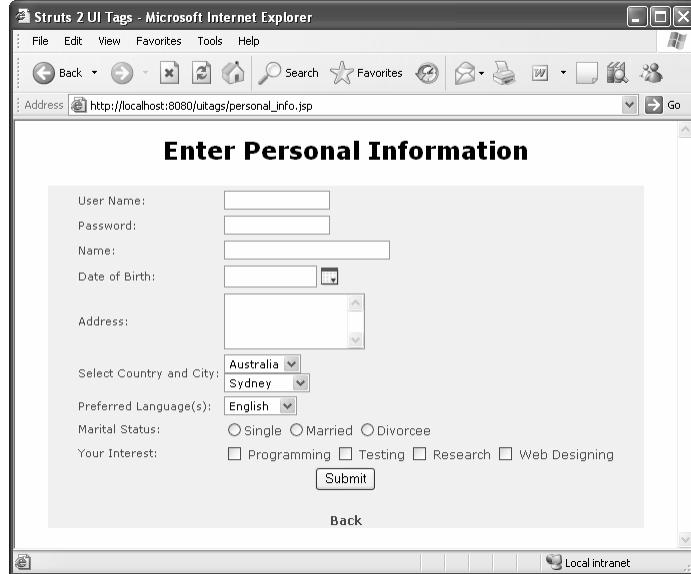


Figure 7.2: The personal_info.jsp page showing the form designed using Form UI tags.

The most important Form UI tag is the `<s:form>`, which contains other Form UI tags and defines the action which is responsible for processing the data submitted through this form. The action attribute of this form is set to `personalAction`. Here's the following action mapping provided to the request path in `struts.xml` file:

```

<action name="personalAction" class="com.kogent.action.PersonalAction">
    <result name="input">personal_info.jsp</result>
    <result name="success">info1.jsp</result>
</action>

```

The JSP page transfers the control to its corresponding action class through the POST method. This action is discussed further in the next section.

Developing **PersonalAction.java**

The action class to be created here is `PersonalAction`, which has properties matching all input fields created in `personal_info.jsp`. The action class also has setter and getter methods for these properties. We use `String` type for each field property.

Here's the code, given in Listing 7.3, for `PersonalAction.java` (you can find `PersonalAction.java` file in `Code\Chapter 7\uitags\WEB-INF\src\com\kogent\action` folder in CD):

Listing 7.3: PersonalAction.java

```

package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class PersonalAction extends ActionSupport {
    private String username;

```

```
private String password;
private String fullname;
private String dob;
private String address;
private String country;
private String city;
private String language;
private String mstatus;
private String skill;

    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getDob() {
        return dob;
    }
    public void setDob(String dob) {
        this.dob = dob;
    }
    public String getFullscreen() {
        return fullname;
    }
    public void setFullscreen(String fullname) {
        this.fullname = fullname;
    }
    public String getLanguage() {
        return language;
    }
    public void setLanguage(String language) {
        this.language = language;
    }
    public String getMstatus() {
        return mstatus;
    }
    public void setMstatus(String mstatus) {
        this.mstatus = mstatus;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
```

```
        this.password = password;
    }
    public String getSkill() {
        return skill;
    }
    public void setSkill(String skill) {
        this.skill = skill;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

If the execution of the page results in success, then the `info1.jsp` is called for displaying the output to the user. Here's the code, given in Listing 7.4, for `info1.jsp` page (you can find `info1.jsp` file in `Code\Chapter 7\uitags` folder in CD):

Listing 7.4: `info1.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"><html>
<head>
<title>Struts2 UI Tags</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2>Welcome! <s:property value="username" /></h2>
<table cellspacing="5" bgcolor="#fdf5ea">
<tr>
<td colspan="2" bgcolor="#fbefd9" height="25" align="center">Login Details</td>
</tr>
<tr>
<td>User Name</td>
<td><s:property value="username"/></td>
</tr>
<tr>
<td>Password</td>
<td><s:property value="password"/></td>
</tr>
<tr>
<td colspan="2" height="25" bgcolor="#fbefd9" align="center">Personal Details</td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">Name</td>
<td><s:property value="fullname"/></td>
</tr>
```

```

<tr>
<td bgcolor="#fbef9" height="25">Date of Birth</td>
<td><s:property value="dob"/></td>
</tr>
<tr>
<td bgcolor="#fbef9" >Address</td>
<td>
<:property value="address" /><br>
<:property value="city" />, <:property value="country" />
</td>
</tr>
<tr>
<td bgcolor="#fbef9" height="25">Language(s)</td>
<td><s:property value="language" /></td>
</tr>
<tr>
<td bgcolor="#fbef9" height="25">Marital Status</td>
<td><s:property value="mstatus" /></td>
</tr>
<tr>
<td bgcolor="#fbef9" height="25">Skills</td>
<td><s:property value="skill" /></td>
</tr>
</table>
<s:a href="index.jsp">Back</s:a>
</body>
</html>

```

The output of Listing 7.4 is shown in Figure 7.3.

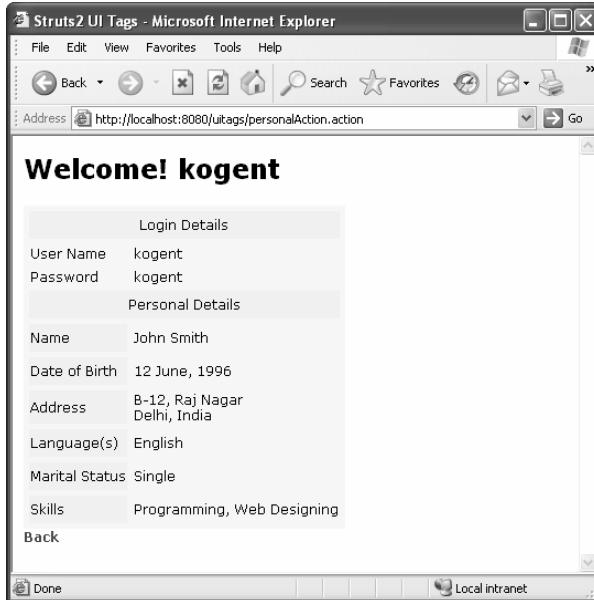


Figure 7.3: Output of info1.jsp.

Developing *general_info.jsp*

When the user hits the link labeled ‘General Information’ created in `index.jsp`, as shown in Figure 7.1, the `general_info.jsp` page appears. This page, similar to `personal_info.jsp`, gives a form using different Form UI tags.

Here’s the code, given in Listing 7.5, for `general_info.jsp` (you can find `general_info.jsp` file in `Code\Chapter 7\uitags` folder in CD):

Listing 7.5: `general_info.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Struts2 UI Tags</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="mystyle.css" type="text/css" />
<s:head theme="simple" />
</head>
<body>
<h2 align="center">General Information</h2>
<table align="center" width="600">
<tr>
<td bgcolor="#fbefd9" align="center">
<s:actionerror/>
<s:form action="generalAction" method="post" cssStyle="color:#7a3d3d">
<s:combobox name="platform"
    label="Select Platform"
    list="{'Windows', 'Linux', 'Unix'}"
    headerKey="-1"
    headerValue="---Select Platform---"/>
<s:optiontransferselect
    name="serverslist"
    label="Select Servers"
    list="{'Tomcat 5.5', 'WebLogic 8.1', 'JBoss 5.0', 'JRunner', 'Sun Server'}"
    size="5"
    doubleName="servers"
    doubleList="{}"
    doubleSize="5"
    leftTitle="List of Servers"
    rightTitle="Your Servers"
    allowUpDownOnLeft="false"
    allowUpDownOnRight="false"
    allowSelectAll="false"
    allowAddAllToLeft="false"
    allowAddAllToRight="false"
    addToRightLabel="">>''
    addToLeftLabel="'<<'"/>
<s:select
    label="Select Language"
    name="language"
    list="%{#{'C':'C', 'C++':'C+'}}">
    <s:optgroup
```

```

        label="Java"
        list="#{'Core Java':'Core Java','Groovy':'Groovy'}" />
    <s:optgroup
        label=".Net"
        list="#{'VB':'VB','ASP':'ASP','C#':'C#'}" />
</s:select>
<s:updownselect
    label="Select Technologies and put them in order(Most comfortable with on
    the top)"
    labelposition="top"
    list="#{'Servlet':'Servlet', 'JSP':'JSP', 'Struts':'Struts', 'EJB':'EJB'}"
    name="technology"
    headerKey="-1"
    headerValue="----Select----"
    moveDownLabel="Down"
    moveUpLabel="Up"
    selectAllLabel="All"/>

<s:file name="file" label="Upload Resume" size="30"/>
<s:submit value="Submit"/>
</s:form>
<s:a href="index.jsp">Back</s:a>
</td>
</tr>
</table>
</body>
</html>

```

This page also contains many UI tags. The output of this page is shown in Figure 7.4. Observe how these UI components created here are rendered.

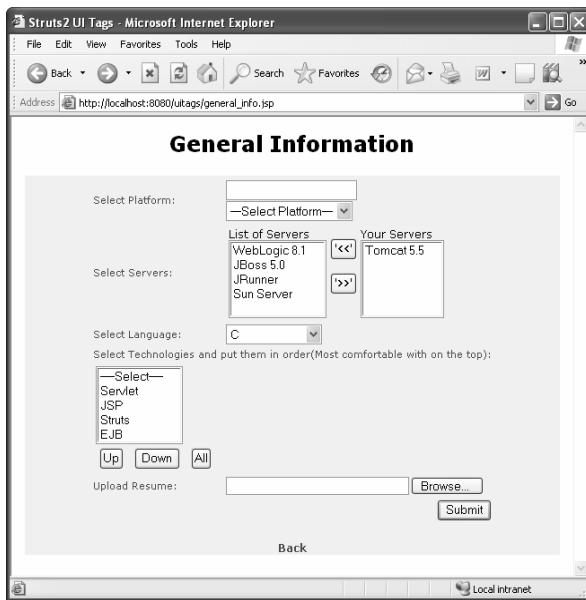


Figure 7.4: The general_info.jsp page showing another form designed using Form UI tags.

The JSP page transfers the control to its corresponding action class through the POST method. The action attribute of the Form tag is set here to be generalAction. Here's the mapping provided in struts.xml file for this request path:

```
<action name="generalAction" class="com.kogent.action.GeneralAction">
    <result name="input">general_info.jsp</result>
    <result name="success">info2.jsp</result>
</action>
```

Developing GeneralInfoAction.java

The GeneralInfoAction.java contains getter and setter methods for each of the input fields created in general_info.jsp page. We use String type for each field property.

Here's the code, given in Listing 7.6, for the action (you can find GeneralAction.java file in Code\Chapter 7\uitags\WEB-INF\src\com\kogent\action folder in CD):

Listing 7.6: GeneralAction.java

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class GeneralAction extends ActionSupport {

    private String platform;
    private String servers;
    private String language;
    private String technology;
    private String file;

    public String getFile() {
        return file;
    }
    public void setFile(String file) {
        this.file = file;
    }
    public String getLanguage() {
        return language;
    }
    public void setLanguage(String language) {
        this.language = language;
    }
    public String getPlatform() {
        return platform;
    }
    public void setPlatform(String platform) {
        this.platform = platform;
    }
    public String getServers() {
        return servers;
    }
    public void setServers(String servers) {
        this.servers = servers;
    }
}
```

```

public String getTechnology() {
    return technology;
}
public void setTechnology(String technology) {
    this.technology = technology;
}
public String execute() throws Exception {
    return SUCCESS;
}
}

```

If the execution of the page results success, then the `info2.jsp` is called for displaying the output to the user.

Here's the code, given in Listing 7.7, for `info2.jsp` (you can find `info2.jsp` file in `Code\Chapter 7\uitags` folder in CD):

Listing 7.7: `info2.jsp`

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"><html>
<head>
<title>Struts2 UI Tags</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<table cellspacing="5" bgcolor="#fdf5ea">
<tr>
<td colspan="2" bgcolor="#fbefd9" height="25" align="center">General Details</td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">Selected Platform</td>
<td><s:property value="platform"/></td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">Selected Servers</td>
<td><s:property value="servers"/></td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">Programming Language</td>
<td><s:property value="language"/></td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">Technologies</td>
<td><s:property value="technology"/></td>
</tr>
<tr>
<td bgcolor="#fbefd9" height="25">File Uploaded</td>
<td>
<s:property value="file" />
</td>
</tr>
</table>

```

```
<s:a href="index.jsp">Back</s:a>
</body>
</html>
```

The output of this page is shown in Figure 7.5 with the data selected and entered in the input fields created in general_info.jsp page.

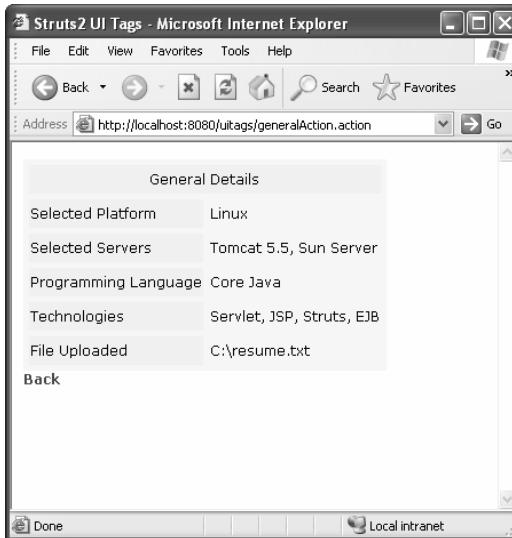


Figure 7.5: The Output of info2.jsp.

In this way, we can see how the various UI tags can be implemented in creating a form and other form components. With the user of these UI tags we get some more features which has definitely improved the way some UI components are rendered on the page. Some additional components created using doubleselect, updownselect has introduced some other types of UI components, which couldn't be implemented on a web page with such ease.

This chapter introduced the notion of building reusable HTML components that tie in with Struts 2 along with the concept of themes and templates for building up standard forms, with a detail of all the UI tags included in Struts 2. Now you can begin to build complex forms using a library of rich tags.

We recommend that you spend some time putting together some sample forms and even explore writing your own templates or overriding the existing ones. Learning how to extend the UI Framework is the best way to get the most out of the framework and make your applications easy to develop and modify.

The next chapter will focus on one more important characteristic of Struts 2 Framework, that is, Result.



8

Controlling Results in Struts 2

If you need an immediate solution to:

See page:

Developing projects_results Application

301

Building Your Own Result Type

311

In Depth

Action classes are used for the processing of user action, requested with the data sent as input parameters. Interceptors provided for all required preprocessing are executed before the execution of action class. The processing of request must consequently provide some result, which is to be sent as the response back to the user. After the execution of action class, we always get a string as result code. This result code is mapped to a specific source to be rendered as view to the user, which may be a JSP page, or an HTML page but it is not limited to these two things. There are different classes implementing `com.opensymphony.xwork2.Result` interface, which are responsible for rendering output to the user. Struts 2 provides a set of Result classes that needs to process different types of results, like rendering JSP or HTML page, generating output for the user using Freemaker or Velocity templates or sometimes invoking other action to get results to the user.

This chapter will describe all the types of Result classes provided with the Struts 2 Framework and their need with all their implementation and configuration details.

Understanding Results

Struts 2, similar to Struts 1, is based on Model-View-Controller architecture and Result in Struts 2 are purely associated with View part of the MVC implementation. The term result is used to refer the output to be displayed to the user as a consequence of the action performed for its request. All user requests are processed by some action class and each request user needs some response in return. Result in Struts 2 can be defined as something which can be obtained with the help of the result code returned by the action and the result class, which renders the associated source (JSP or HTML) as the next view to the user.

Action classes can return one or many types of result codes ('success', 'error') and the results generated and returned to the user for these different result codes need not be of the same type. The different classes provided to generate view make sure that the response is in the proper format so that it is easily understandable by the user.

The supported Result classes are configured in the Struts configuration file so that they can be used by different actions. The association between action and the possible results is also mapped through the result configuration provided with each action mapping in `struts.xml` file. The Struts 2 API included different interface and classes associated with the working of Results. We need to discuss them to understand how an Struts 2 Result is used.

Result Interface

Different types of required result formats need different result classes to render or process the source. Struts 2 provides a general interface `com.opensymphony.xwork2.Result`, which is implemented by all result classes. The Struts 2 Framework comes up with some result classes that are already implemented. You can use these classes, for providing different views to the user. In addition to these classes, you can also build your own result class by implementing this interface.

The structure of the Result interface is shown here with the signature of its single method void execute(ActionInvocation):

```
package com.opensymphony.xwork2;

public interface Result {
    public void execute(ActionInvocation invocation) throws Exception;
}
```

The Result interface is simple, just like the Action interface, but there is a difference between the execute() method present in both the interfaces. The execute() method of the Action interface returns the string value, while the execute() method of the Result interface returns void. The ActionInvocation object which is passed as an argument to the execute() method of Result interface is used to obtain the required details for further processing. The methods invoked over the object of ActionInvocation includes getResultCode() and getAction().

ResultConfig Class

The com.opensymphony.xwork2.config.entities.ResultConfig class represents a result configured with an action mapping provided in struts.xml file. This class defines a set of fields, constructors, and methods to work with. Tables 8.1 and 8.2 displays the fields and methods, respectively, of the ResultConfig class.

Table 8.1: Fields of ResultConfig class

Field Name	Description
private String className	It is the name of result class
private String name	It is the name of result
private Map params	It is the map of all result parameters

Table 8.2: Methods of ResultConfig class

Methods	Description
void addParam(String name, Object value)	It adds parameter
String getClassName()	It returns result class name
String getName()	It returns result name
Map getParams()	It returns a Map object containing parameters
void setClassName(String className)	It sets result class name
void setName(String name)	It sets result name
void setParams(Map params)	It sets parameter map

ResultTypeConfig Class

The com.opensymphony.xwork2.config.entities.ResultTypeConfig class represents a result type configured in the configuration file. The configuration of result types is provided by using the <result-type> tag in the configuration file. Tables 8.3 and 8.4 displays the fields and methods, respectively, of the ResultTypeConfig class.

Table 8.3: Fields of ResultTypeConfig class

Fields	Description
private String clazz	It is name of the result class
private String defaultResultParam	It is name of the default result parameter
private String name	It is the reference name of the Result
private Map params	It is a parameter map

Table 8.4: Methods of ResultTypeConfig class

Methods	Description
void addParam(String key, String value)	It adds a new parameter
String getClazz()	It returns result class
String getDefaultResultParam()	It returns default parameter name
String getName()	It returns reference name of the result
Map getParams()	It returns parameter map
void setClazz (String clazz)	It sets result class
void setDefaultResultParam(String defaultResultParam)	It sets default result parameter
void setName(String name)	It sets reference name for the result
void setParams(Map params)	It sets parameter map

After discussing these interface and classes, we can now move to our discussion on how Struts 2 Results are configured and implemented in the application.

Configuring Results

The method of the action class processing the user request returns a string as result code. The String value could be ‘success’, ‘error’, ‘input’, etc. The string result code is matched with the name of the result element that is to be selected in response to the outcome of the given action. The action mapping defines the set of possible results, which describes the different possible outcomes. The Action interface has a defined standard set of result tokens. This result tokens describe the names of the predefined result. These result names are mentioned here as follows:

- String SUCCESS = “success”;
- String NONE = “none”;
- String ERROR = “error”;
- String INPUT = “input”;
- String LOGIN = “login”;

The preceding mentioned names of results are predefined, but you can also add your own result name to match the specific cases in your Web application. For example, when the `execute()` method of your Action will return ‘success’ string value then this string value is matched with the result element having `name=“success”` in `struts.xml`.

So we need to configure all supported result types to be used in the application. In addition, the set of possible results to be rendered are also defined in `struts.xml` file for different action mappings. So, the basic elements used, while configuring results, are `<result-types>`, `<result-type>`, and `<result>`.

Configuring Result Types

To configure allowed result types to be used in the application we have two approaches. The first one is to define all the result types to be used in the application providing a set of `<result-type>` elements for each. This is always required if you want to configure your customized result types.

Here’s the sample, given in Listing 8.1, for `struts.xml` file:

Listing 8.1: A sample result configuration

```
<struts>
<package name="mypackage">

    <result-types>
        <result-type name="dispatcher"
            class="org.apache.struts2.dispatcher.ServletDispatcherResult"
            default="true"/>
        <result-type name="redirect"
            class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
    </result-types>

    <interceptors>
        //Interceptors declaration.
    </interceptors>

    <action name="login" class="com.kogent.LoginAction">
        <result name="input">login.jsp</result>
        <result name="success" type="redirect">/secure/admin.jsp</result>
    </action>
</package>
</struts>
```

```
</action>
</package>
</struts>
```

This type of configuration needs you to define all result types to be used and, of course, all types of interceptors too, which is required to preprocess the request before the action class is executed. The second approach is to include `struts-default.xml` file in your copy of `struts.xml` file. The `struts-default.xml` file defines a package named `struts-default` with all the available result types configured there with other configurations. The package defined in our `struts.xml` can extend the `struts-default` package to make all result types available for all action mappings provided here in our package.

Here's the sample `struts.xml` file, given in Listing 8.2, again with the second approach:

Listing 8.2: A sample results configuration

```
<struts>
    <include file="struts-default.xml"/>
    <package name="mypackage" extends="struts-default">
        <action name="login" class="com.kogent.LoginAction">
            <result name="input">login.jsp</result>
            <result name="success" type=
                "redirect">/secure/admin.jsp</result>
        </action>
    </package>
</struts>
```

The second approach is better, as it provides a standard way of implementation of all framework results. We can use all types of result types directly without providing any `<result-type>` element for them.

Result Elements

An action may return a result code, which is matched with the names of the results configured for this particular action. The result configured for an action includes the details, like name of the results, which is to be matched with the result code string returned by the action and the result type. All results for an action are declared by using the `<result>` tag. Here's a sample result configuration given for an action:

```
<action name="login" class="com.kogent.LoginAction">
    <result name="input">login.jsp</result>
    <result name="success" type="redirect">/secure/admin.jsp</result>
</action>
```

The two results are configured here for action `login`. The two results with `name="success"` and `name="input"` are here to work for two different result codes 'success' and 'input' returned by method of `LoginAction` action class. The `type` attribute in `<result>` element defines the result type to be used here to render the output to the user.

The result element provides these logical names to the result, without caring about the implementation details of these results. Most of the results simply forward to Java Server Page (JSP) or template.

There are some intelligent defaults, which can be set in the configuration files, i.e. `struts.xml` and `struts-default.xml`. A default result type can be set as a part of the configuration file for each of the

packages. If a package extends some other package, then this child package can have its default result type or it may inherit its default result type from the parent package. The Dispatcher Result is set as the default result type in `struts-default.xml`. All packages extending `struts-default` package will have the dispatcher as default result, unless configured separately in the child package. Here's how the Dispatcher Result is made default:

```
<result-types>
<result-type name="dispatcher"
class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true" />
</result-types>
```

If a result type is not specified, the Struts 2 Framework uses the `dispatcher` result as the default result type. The Dispatcher Result type forwards to another web resource. If the web resource is a JSP page, then the container will render it using the JSP engine. Similarly, if the result name is not specified at the requisite place, the Struts 2 Framework will give it the default name as 'SUCCESS'. The use of these default result types make the result configuration simple.

You can see some intelligent defaults used here, like setting values intelligently for the attributes of `<result>`, like name, type, and the value for the default parameter of the result class:

- Result element without using any default:

```
<result name="success" type="dispatcher">
    <param name="location"/>/ThankYou.jsp</param>
</result>
```

- Result element which is using some of the defaults, like name and type:

```
<result>
    <param name="location"/>/ThankYou.jsp</param>
</result>
```

The name is taken as `success` and the type is taken as `dispatcher` here. The `param` tag has an attribute `name`, which sets the location of the web resource.

- The text set between the `<result></result>` tags is taken as the value for the default parameter defined for the result type being used. Usually the location is defined as the default parameter for the respective result class:

```
<result>/ThankYou.jsp</result>
```

Global Results

Global Results are used to reduce the amount of configuration in the `struts.xml` file. This feature is achieved by using global mapping, for the Global Results. Global Results are useful because there are some results, which are associated with a number of actions. So, instead of configuring the same result with different actions, we'll use global mapping. Such a situation often arises in the real world Web applications. In secure Web applications, which come across in the real world, a user may try to access a page without having the rights for accessing that page. For example, in a secure application a client

might try to access a page without being authorized and the other example is many actions may need access to a ‘login’ result. A user may try to access the account summary of some other person, without proper login. In such a case, the application will prompt the user to first login. Such a request for login may also arise in some other situation also. So instead of configuring the login result with each and every action, we’ll define it in the global- results.

If actions need to share the results, then a set of Global Results can be defined for each package. The Struts 2 Framework will first look for a local result that is nested in the specific action. In case no local result is found, it will check the Global Results.

The following code shows the Global Result mappings for login and unauthorized result codes:

```
<global-results>
    <result name="login" type="redirect-action"/>/login.jsp</result>
    <result name="unauthorized"/>/unauthorized.jsp</result>
</global-results>
```

You can define any number of Global Result mappings for a single package. Since Global Results are searched after local results, we can override any Global Result by creating a local result mapping for a specific action. It is best to use the absolute path for Global Results because you don’t know the context in which they are going to be invoked.

After having this general discussion over Results in Struts 2, we are further explaining different types of Struts 2 Results available which can be used in different scenarios. All Result types have been discussed here with their implementation and functionality.

Implementing Different Result Types in Struts 2

The different results generated and delivered to the user for different result codes are not of the same type. For example, the result ‘success’ may render a JSP or HTML page and another result ‘error’ may need to send a HTTP header to the user browser. Struts 2 comes bundled with several inbuilt result types. The Struts 2 Framework provides several implementation of the com.opensymphony.xwork2.Result interface. These implementations can be directly used in your application. Although you can make your own Result class and use it, generally these inbuilt implementations are sufficient for developing the View portion of the Web application. We can define a group of supported `<result-type>` under `<result-types>` elements in `struts.xml`. We can alternately include the `struts-default.xml` file and extend the struts-default package to make all the `<result-type>` definitions available in our `struts.xml` file.

Here’s the result types, given in Listing 8.3, that are configured in `struts-default.xml` file:

Listing 8.3: Framework Results configured in `struts-default.xml` file

```
<package name="struts-default">
    <result-types>
        <result-type name="chain"
            class="com.opensymphony.xwork2.ActionChainResult"/>
        <result-type name="dispatcher"
            class="org.apache.struts2.dispatcher.ServletDispatcherResult"
            default="true"/>
        <result-type name="freemarker"
            class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
        <result-type name="httpheader"
```

```

        class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
<result-type name="redirect"
class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
<result-type name="redirect-action"
class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
<result-type name="stream"
class="org.apache.struts2.dispatcher.StreamResult"/>
<result-type name="velocity"
class="org.apache.struts2.dispatcher.VelocityResult"/>
<result-type name="xslt" class=
            "org.apache.struts2.views.xslt.XSLTResult"/>
<result-type name="plaintext"
class="org.apache.struts2.dispatcher.PlainTextResult" />
</result-types>

</package>
```

Struts 2 provides the following Result types, which are configured in the `struts-default.xml` file for the name of the result with their associated Result classes. Table 8.5 shows the list of results types.

Table 8.5: Different Result types

Result name	Description
chain	Chain Result is used for Action Chaining
dispatcher	Dispatcher Result is used for integration of web resource such as JSP integration
redirect	Redirect Result is used for redirecting the browser to a new URL
velocity	Velocity Result is used for integration of Velocity
freemarker	FreeMarker Result is used for integrating the FreeMarker
httpheader	HttpHeader Result is used for controlling the special Http behaviors
redirect-action	Redirect Action Result is used for redirecting to another action mapping
stream	Stream Result is used to stream an InputStream back to the browser usually for tasks such as file downloading
xslt	XSL Result is used for integrating the XML/XSLT in the output
plaintext	PlainText Result is used for displaying the raw content of a particular page such as HTML, JSP

You can also create additional Result types and plug them in your Web application, by implementing the `com.opensymphony.xwork2.Result` interface. You can make Result types for operations, such as generating the e-mails, generating images, etc. Out of these Result types, certain Result types (chain,

dispatcher, redirect, etc.) are used frequently, as compared to other Result types. We'll discuss the different Result types available with Struts 2 Framework in detail here, for their use cases and their configurations.

Chain Result

The Chain Result is used for the purpose of action chaining. Action chaining is the ability to chain multiple actions into a defined sequence or pattern. The Chain Result type invokes an action with its own interceptor stack and Result. The interceptor of this action allows the action to forward the request to the target Action. The state of the source Action is propagated along with this Action. Action chaining is a powerful technique, for building complex and dynamic web-based applications. The advantage of action chaining is that actions can share data easily. This happens, because both actions are executing for the same request, i.e. they share the same `ActionContext` and `OgnlValueStack`. By using this technique, you can chain two or more actions in the single Result. Table 8.6 lists four parameters for Chain Result.

Table 8.6: Parameters for Chain Result

Parameters	Description
actionName	This is the default parameter, which provides the name of the action that will be chained to the first action
namespace	This parameter is used to determine the namespace in which the Action, to which we are chaining, is present. If the namespace is null, then it means that the chained Action is present in the current namespace
method	This parameter is used to specify another method of the target Action to be invoked; If the parameter is null, then it defaults to <code>execute</code> method
skipActions	This parameter is optional, and used to give comma-separated list of action names for the actions that could be chained to the source action

Here's Listing 8.4 showing how you can implement action chaining:

Listing 8.4: Chain result in struts.xml configuration

```

<package name="public" extends="webwork-default">
    <!-- Chain AccountDetail to login, using the default parameter -->
    <action name="accountDetail" class="...">
        <result type="chain">login</result>
    </action>

    <action name="login" class="...">
        <!-- Chain to another namespace -->
        <result type="chain">
            <param name="actionName">chainingAction</param>
            <param name="namespace">/secure</param>
        </result>
    </action>

```

```

</package>

<package name="secure" extends="webwork-default" namespace="/secure">
    <action name="chainingAction" class="...">
        <result>admin.jsp</result>
    </action>
</package>

```

From Listing 8.4, the action name accountDetail is chained with the action name login by using the default parameter and the action name login is chained with the action name chainingAction by using actionPerformed and nameSpace of Chain Result parameter. After transferring these chaining Actions, the admin.jsp is finally used to display as your result view.

Figure 8.1 shows how Chain Result invokes another action, which in turn invokes the Dispatcher Result. The Dispatcher Result ends the action chain. This figure shows the chain of only two actions, but you can add more than two actions, depending upon the need of the Web application.

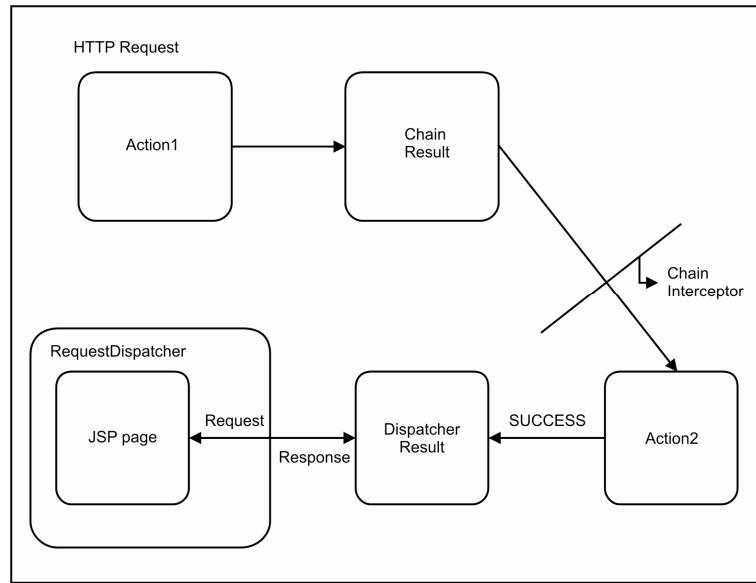


Figure 8.1: Action chaining between two actions using Chain Result.

Figure 8.1 shows action chaining using Chain Result. As soon as, the Action1 gets completed and returns SUCCESS, the Chain Result gets invoked, which in turn invokes another action Action2. Action2 does its processing in the same HTTP request, and gives control to Dispatcher Result, which finally provides the required view and ends the action chaining. The Chain Result works in conjunction with Chain Interceptor.

Dispatcher Result

The Dispatcher Result is used to integrate a view in your Web application. It will include the content of a view or forward to a view, which is generally a JSP page. The Dispatcher Result is the default result type in Struts 2. Struts 2 uses a RequestDispatcher, where the target Servlet or JSP receives the same request or response objects, as the original request or response. This will allow us to share the data between them by using `request.setAttribute()`. Display of different page, under the same request

is possible by using Servlets and RequestDispatcher class. Table 8.7 lists two parameters in Dispatcher Result.

Table 8.7: Parameters for Dispatcher Result	
Parameters	Description
location	This is the default parameter, which provides the location, where the control will go to after processing; for example, a JSP page like success.jsp
parse	This parameter is set to true by default; If parse is set to false, then the location parameter will not be parsed for OGNL expressions

The following code snippet shows how a Dispatcher Result is used in your Struts configuration to dispatch a JSP page. In this snippet code, the location parameter will forward your result as error.jsp view:

```
<result name="error" type="dispatcher">
    <param name="location">error.jsp</param>
</result>
```

Figure 8.2 shows how the Dispatcher Result works. Action invokes the Dispatcher Result, which in turn invokes the RequestDispatcher class to send the response back to the result.

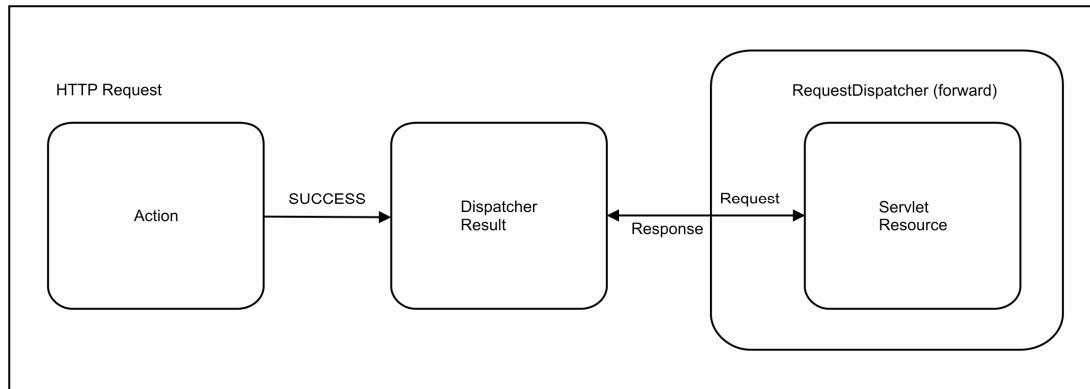


Figure 8.2: Handling request by RequestDispatcher using Dispatcher Result.

There are three different contexts in which Dispatcher Result can be called. Depending on the context in which Dispatcher Result is called, the way of dispatching changes. In all there are three ways of dispatching. This difference in ways is due to the Servlet specification, which has few restrictions about the two ways of dispatching, i.e. include and forward. The three contexts are mentioned here:

- ❑ **JSP context**—If we are in the scope of a JSP, i.e. a page context is available, then PageContext's {@link PageContext#included(String)} method is called.
- ❑ **Forward context**—If there is no PageContext and also there is no include, then in such a case, a call to {@link RequestDispatcher#forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse)} method is called.

- ❑ **Included context**—This is the case, only when the preceding two scenarios fail to happen then the


```
{@link RequestDispatcher#include(javax.servlet.ServletRequest, javax.servlet.ServletResponse) include}
```

 method is called. This context is used rarely.

The JSP context is discovered, when a `PageContext` object is present in the `ActionContext`. If `PageContext` exists, then its `include()` method is called, and the location is specified in the result. This will allow a JSP to include an action and its corresponding result. The Dispatcher Result is responsible, for sending arbitrary request to any Servlet resource. Usually, a JSP is dispatched, as shown in Figure 8.3; however, we can dispatch any resource, such as Servlet, HTML files, etc.

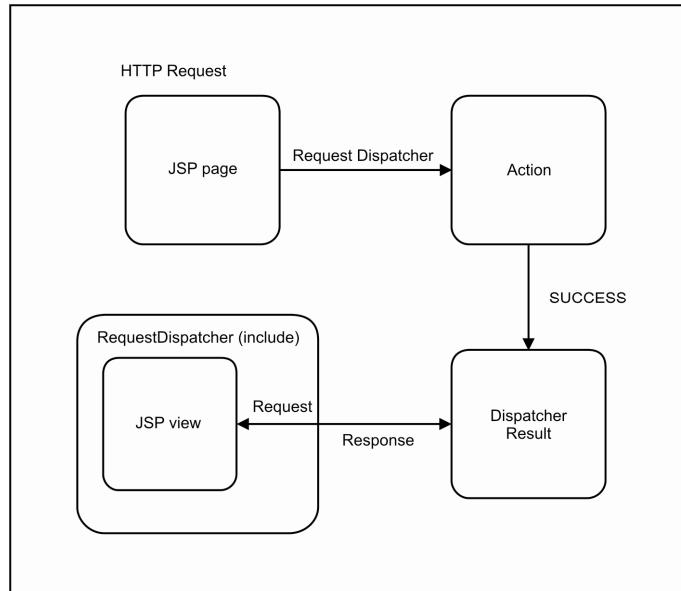


Figure 8.3: Dispatcher Result in JSP context.

Here's the code, given in Listing 8.5, where a JSP is invoking action `HelloWorld` under JSP context (the scenario shown in Figure 8.3):

Listing 8.5: A JSP invoking an action under JSP context

```

<%@ taglib prefix = "s" uri = "/struts-tags"%>
<html>
    <head><title> JSP context</title></head>
    <body>
        <s:action = "/HelloWorld"/>
    </body>
</html>

```

The outcome this JSP page (Listing 8.5) will include the output of `HelloWorld` action directly in the form of `RequestDispatcher include`. See the use of `<s:action/>` tag in the Listing 8.5.

As mentioned earlier, if there is no page context and no include, then a call to forward be made. In this case, an action gets executed because of a direct request from the Web browser, and the result is Dispatcher Result. There is no `PageContext` in the `ActionContext`. So, the `RequestDispatcher` is

used to forward another resource. This resource may be a JSP, which, in turn, invokes another action. This results in a JSP context, which is described earlier. This whole scenario is shown in Figure 8.4.

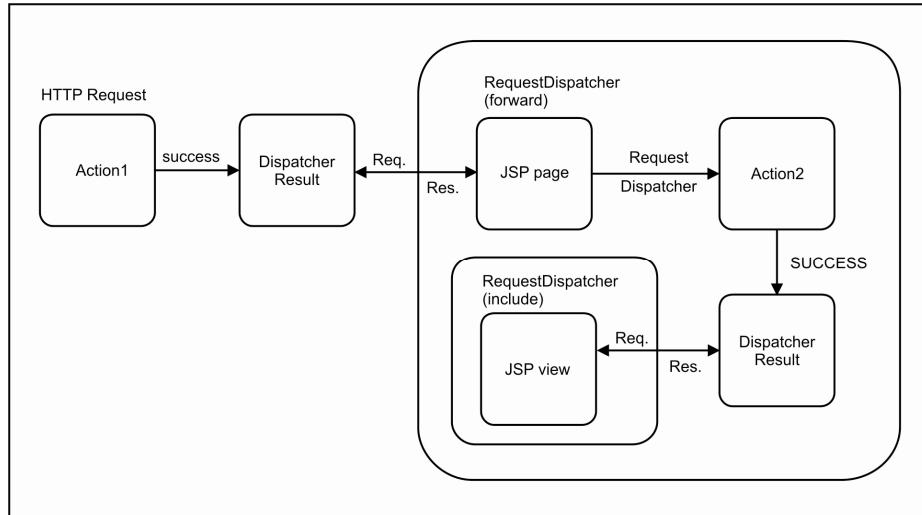


Figure 8.4: Dispatcher Result in forward context.

The last context, i.e. included context, is rarely used. This context is similar to the JSP context. The difference is only that, in JSP context, the action is executed in the JSP, while in included context the action is executed because of RequestedDispatcher include. This happens because, instead of using `s:action`, we are using `jsp:include` for displaying a resource for an action. This difference is not so big, but it can create different contexts, in which the Dispatcher Result is called.

Here's the code, given in Listing 8.6, that shows a JSP with included context:

Listing 8.6: A JSP in the included context

```

<%@ taglib prefix = "s" uri = "/struts-tags" %>
<html>
    <head><title>included context</title></head>
    <body>
        <jsp:include page= "HelloWorld.action"/>
    </body>
</html>

```

It is very difficult, to find a difference between forward and included context, but one of the important difference between these two contexts, is that forwards can occur a number of times, until an include takes place. After the occurrence of include, only include may take place. The Dispatcher Result takes care of this fact, and never calls a forward, after include has taken place. Figure 8.5 shows the included context, which is demonstrated in Listing 8.6.

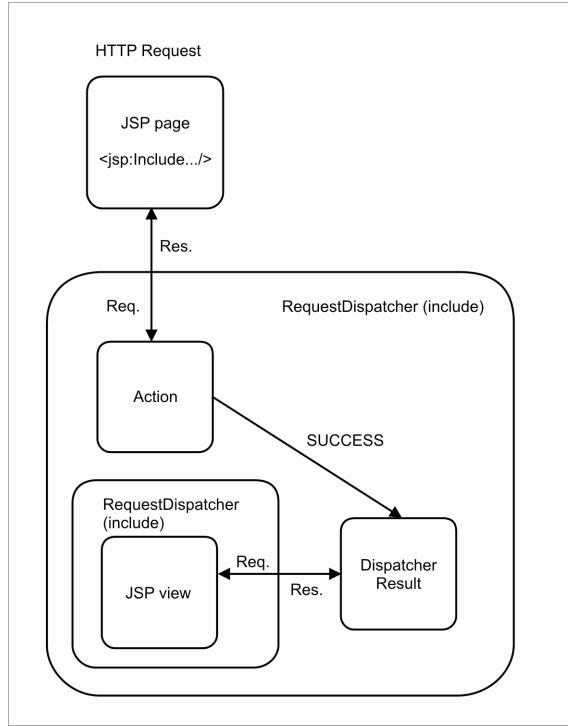


Figure 8.5: Dispatcher Result in included context.

NOTE

Sometimes people think that dispatching to another action is same as action chaining. They share some similar qualities—namely, that they execute in the same thread/request—but they don’t do exactly the same thing. An action chain executes both actions in the same action invocation whereas dispatching to a second action causes two invocations to take place. Also, dispatching doesn’t copy the parameters of the first action and set them on the second action. As a general rule, if you find yourself dispatching to another action, you probably should use action chaining.

FreeMarker Result

FreeMarker Result is used for the integration of FreeMarker templates in the view. Its function is almost same as the Velocity Result. FreeMarker renders a view using the FreeMarker template engine. While velocity is strictly a template engine, FreeMarker is more than just a template engine. FreeMarker provides features, such as the ability to use any JSP tag library. The FreeMarkerManager class configures the loaders of the template, so that the template location can either be relative to the web root folder, or this location can be a classpath resource. The FreeMarkerManager class configures the template loaders so that the template location can be either Relative to the web root folder (/WEB-INF/views/home.ftl) or a classpath resource (com/company/web/views/home.ftl). The parameters for FreeMarker Result are listed in Table 8.8.

Table 8.8: Parameters for FreeMarker Result

Parameters	Description
location	This is the default parameter, and gives the location of the template to be processed
parse	This parameter is “true” by default. If it is set to false, then the location parameter will not be parsed for the OGNL expressions
contentType	This parameter defines the content type of the template; Its value is set to “text/html” by default

The following code snippet shows how the FreeMarker template engine is used in your Struts configuration to invoke a FreeMarker template view.ftl:

```
<result name="success" type="freemarker">view.ftl</result>
```

- ❑ Here, when the result type is set to freemarker, the Web application will display a view based on FreeMarker template.

HttpHeader Result

Sometimes we need to send a HTTP header back to the browser, a different result type, i.e. HttpHeader, is used. HttpHeader result type is a custom result type that is used for setting the HTTP headers and status by optionally evaluating against the ValueStack. There are three parameters available in the HttpHeader result type mentioned here in Table 8.9.

Table 8.9: Parameters for HttpHeader Result

Parameters	Description
status	This parameter provides the HTTP servlet response status code; this status code is set on the response, provided to the user
parse	This parameter is set to true by default. If it is set to false, then the headers parameter will not be parsed for the OGNL expressions
headers	This parameters provides the value for the headers

The following code snippet shows how to set the HTTP header and status by using HttpHeader result in your Struts configuration file:

```
<result name="success" type="httpheader">
<param name="status">204</param>
<param name="headers.a">a custom header value of HttpHeader result</param>
<param name="headers.b">another custom header value of HttpHeader result</param>
</result>
```

In the preceding snippet code, status 204 is Http Servlet response status code and headers.a and headers.b are the custom header value of your `HttpHeader` result.

Redirect Result

The Redirect Result is used to redirect the browser to a new location. This results in the form of a new request. The redirect is done by calling the `HttpServletResponse`'s `sendRedirect` method. The consequence of redirect is that the action (including action instance, action errors, field errors, etc.), which is just executed is lost and is no longer available in the current request. This is because the actions are built, using the single thread model. The only way of passing the data is with the help of session or with web parameters through `url?name=value`, which can be an OGNL expression. The Redirect Result uses two parameters, which are given here in Table 8.10.

Table 8.10: Parameters for Redirect Result

Parameters	Description
location	This is the default parameter, which provides the location where the control will go to after processing; For example, a JSP page like <code>success.jsp</code>
parse	This parameter is set to true by default; if parse is set to false, then the location parameter will not be parsed for OGNL expressions

The following code snippet shows how a Redirect Result is used in your Struts configuration to redirect a JSP page. In this snippet code, the location parameter will redirect your result as `welcome.jsp` view:

```
<result name="success" type="redirect">
<param name="location">welcome.jsp</param>
<param name="parse">false</param>
</result>
```

Figure 8.6 shows how the Redirect Result takes place.

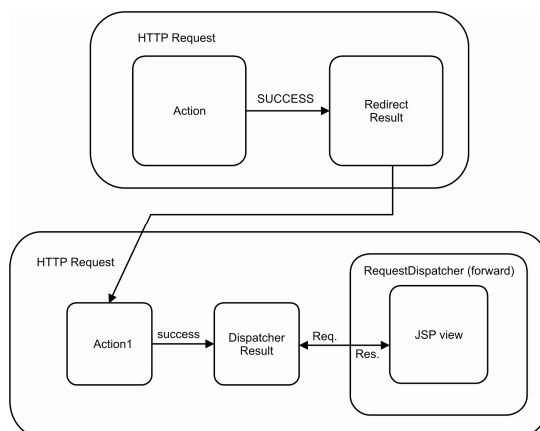


Figure 8.6: A Redirect Result.

Figure8.6 shows that a Redirect Result redirects the browser to another action, but this doesn't happen always. You are free to redirect the user to any URL, even to a URL that is not a part of your Web application.

You may think that the Dispatcher Result and Redirect Result are almost the same; however, they work differently. The Redirect result sends a HTTP return code back to the browser along with a new location in HTTP headers. The browser captures the requisite information from the header, and issues the HTTP request for the new location automatically. The browser does not get to know about the actions that are taken internally. It appears to the browser, as if all the processing is taking place in the single request. Redirect, on the other hand, affects the performance to some extent because the browser has to take two rounds of trips, instead of just one. Due to this, the Redirect becomes slower, as compared to the Dispatcher Result.

Redirect Action Result

Redirect Action Result is used to redirect the browser to another URL that invokes the specified action. The Redirect Action Result uses the ActionMapper provided by the ActionMapperFactory to redirect the browser to other URL, which will invoke the specific action and an optional namespace. This way of redirecting is better as compared to the Servlets SendRedirectResult, because it does not require the encoding of the URL patterns processed by the ActionMapper in the struts.xml configuration files. Due to this feature, you can change the URL pattern at any point of time without affecting your Web application, i.e. your Web application will still work properly. Instead of using the previously discussed Redirect Result, it is strongly recommended to use the Redirect Action Result whenever redirecting the browser to any action. There are two parameters in the Redirect Action Result as mentioned here in Table 8.11.

Table 8.11: Parameters for Redirect Action Result

Parameters	Description
actionName	This is the default parameter. This parameter provides the name of the action, to which the URL is redirected
namespace	This parameter provides the namespace in which the action to which we are redirecting the URL is present. If namespace is set to null, then this defaults to the current namespace

The following code snippet shows how the browser is redirected to another URL by invoking the Redirect Action Result:

```
<package name="..." extends="struts-default" namespace=". . . ">

    <action name="someAction" class="com.kogent.action.SomeAction">

        <result name="success" type="redirect-action">

            <param name="actionName">actionName</param>
            <param name="namespace">/someNamespace</param>
            <param name="userid">kogent</param>
            <param name="password">kogent</param>
        </result>
```

```
</action>
</package>
```

In the preceding snippet code, the `passingRequestParameters` will redirect the browser to another URL by using redirect action URL like this:

```
/someNamespace/actionName.action?username=kogent&password=kogent
```

Stream Result

Stream Result is used for streaming the `InputStream` back to the browser, usually for tasks, such as file downloading. This is custom result type, which is used for sending the raw data directly to the `HttpServletResponse` via an `InputStream`. This result type is very useful for allowing the users to download the content. There are five parameters available in the Stream result type as listed in Table 8.12.

Table 8.12: Parameters for Stream Result

Parameters	Description
<code>contentType</code>	This parameter provides the stream MIME-type, as sent to the Web browser; The default value for <code>contentType</code> is ‘text/plain’
<code>contentLength</code>	This parameter provides the stream length in bytes; The browser shows the <code>contentLength</code> by displaying the progress bar
<code>contentDisposition</code>	This parameter provides the content disposition header value for specifying the file name; The default value for this parameter is ‘inline’; The value for this parameter is mentioned as “filename= “document1.pdf”
<code>inputName</code>	This parameter provides the name of the <code>InputStream</code> property from the chained action; The default value for this parameter is <code>inputStream</code>
<code>bufferSize</code>	The parameter provides the size of the buffer from which we’ll copy from input to the output the default value for the <code>bufferSize</code> parameter is “1024”

The following code snippet shows how raw data is send directly to the `HttpServletResponse` via an `InputStream` in your Struts configuration file:

```
<result name="success" type="stream">

<param name="contentType">image/jpeg</param>
<param name="inputName">myStream</param>
<param name="contentDisposition">filename="somefile.pdf"</param>
```

```
<param name="buffersize">1024</param>
</result>
```

Here, the `somefile.pdf` file is sent directly to the `HttpServletResponse` via a `myStream` and the `contentType` is `image/jpeg`.

Velocity Result

Velocity Result is used for the integration of Velocity in the Web application. Velocity Result uses the Servlet container's `JspFactory` and mocks a JSP environment. Velocity Result then displays a `velocity` template that is streamed directly into the Servlet output. The integration of velocity is needed, because whenever Java code is included in the JSP page by using scriptlet, it becomes very complex and usually results in a poorly designed code. The parameters used by the Velocity Result are listed in Table 8.13.

Table 8.13: Parameters for Velocity Result

Parameters	Description
<code>location</code>	This is the default parameter, which provides the location, where the control will go to after processing; For example, a JSP page, like <code>success.jsp</code>
<code>parse</code>	This parameter is set to true by default. If <code>parse</code> is set to false, then the <code>location</code> parameter will not be parsed for OGNL expressions

The following code snippet shows the usage of Velocity Result in your Struts configuration to invoke a Velocity Result `user.vm`:

```
<result name="success" type="velocity">
<param name="location">user.vm</param>
</result>
```

There are several reasons why velocity is preferred over JSP. These reasons are mentioned here:

- ❑ Velocity is much faster than JSP for most of the containers.
- ❑ Scriptlet usually gives a code smell, i.e. if the developers are given the option of using Java code in their JSP page, they will be using Java code whenever required, which is treated as bad programming.
- ❑ Since velocity has a simpler format (no angled brackets), it becomes easier to integrate with the existing editors for XML, HTML, etc.
- ❑ Velocity is a part of Apache Jakarta project and available for free download, from the <http://velocity.apache.org>. Till now, we have seen that `RequestDispatcher` and `redirect` are the only ways for providing the output to the user. Velocity gives you a new way of providing the output to the user. Velocity inserts the content directly into the output without using the expensive `RequestDispatcher`. There is one more advantage of velocity over JSP, which is the lesser compilation time. JSP should be converted into the Java file, and then into the corresponding `.class` file, while velocity is parsed and ready to be executed more quickly. Figure 8.7 shows how

the velocity template is streamed directly into the HTTP response without the overhead of how RequestDispatcher result or a Redirect Result incurs.

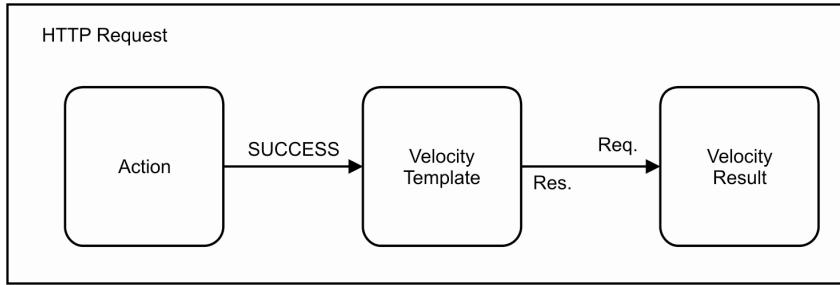


Figure 8.7: a Velocity Result streamed directly into the output in the browser.

XSLT Result

The XSLT result is used for integrating XML/XSLT. XSLT result uses the XSLT to transform the Action object into XML format. The latest version is modified to specifically deal with the Xalan flaws. When you use Xalan, you may find that, even though your style sheet is minimal like the one given here, Xalan would still iterate through every property present in your action and all its descendants:

```
<xsl: template match= "/result">
<result/>
</xsl: template>
```

If you had double-linked objects, then Xalan would work forever analyzing infinite object tree, even if your stylesheet was not constructed for the processing of all the objects. This happens because the current Xalan eagerly and extensively converts everything into its internal DTM model, before continuing with further processing. To solve this problem, a loop eliminator is added, which works by indexing every object-property combination during the processing. Suppose, during the processing it finds that a particular property of the object is already processed, and then it doesn't process down to the lower levels.

For example, we have two objects A and B with the following set of properties:

```
A.B = B;
```

and

```
B.A = A;
action.A = A;
```

Because of the introduction of loop eliminator, the resulting XML document based on A would be:

```
<result>
<A>
    <B/>
</A>
</result>
```

Without it, there would be an endless A/B/A/B/A/B..... elements.

The XSLT Result code tries to deal with the fact that DTM model is built in such a manner that childs are processed before siblings. The result is that if there is object A that is set in action's A property and very deeply under the action's C property, then, in that case, it would appear under C, and not under A. That is not what, we are expecting and that is why XSLT result allows objects to repeat in various places to some extent.

NOTE

In your .xslt file, the root match must be named as result.

There are five parameters available in the XSLT result, as mentioned in Table 8.14.

Table 8.14: Parameters for XSLT Result

Parameters	Description
location	This parameter is the default parameter, and provides the location to go to after execution
parse	This parameter is set to true by default; if it is set to false, then the location parameter is not going to be parsed for the OGNL expressions
matchingPattern	This parameter provides the pattern that matches only the desired elements; by default it matches everything
excludingPattern	This parameter provides the pattern, which will eliminate the unwanted elements; by default it matches none
struts.xslt.nocache	This parameter is specific to struts.properties file. By default, this property is set to false, which disables stylesheet caching.

In the following snippet code, the XSLT result would only walk through the action's properties without their childs:

```
<result name="success" type="xslt">
<param name="location">foo.xslt</param>
<param name="matchingPattern">^/result/[/*]${}</param>
<param name="excludingPattern">.*(hugeCollection).*</param>
</result>
```

It would also skip every property that has `hugeCollection` in their name. Element's path is first compared to the `excludingPattern`; if it matches, it's no longer processed. Then it is compared to the `matchingPattern` and processed only if there's a match.

PlainText Result

PlainText Result is used to display the raw contents of a particular page for example JSP or HTML page. This result sends the contents in the form of plain text. Table 8.15 consists two parameters for this result type.

Table 8.15: Parameters for PlainText Result

Parameters	Description
location	This is the default parameter and provides the location of the JSP/HTML page to be displayed as plain text
charSet	This parameter is optional, and provides the char set to be used in the JSP or HTML page. This character set is used for setting the response type (<code>content-type=text/plain; charset=UTF-8</code>) when reading the content by using a reader

The following code snippet shows how raw contents of a particular page is displayed in JSP format:

```
<action name="displayJspFormat" >
    <result type="plaintext">
        <param name="location"/>/somefile.jsp</param>
        <param name="charset">UTF-8</param>
    </result>
</action>
```

Here, the result type is set to `plaintext` and the `somefile.jsp` is displayed as your result in plain text format.

After discussing the different result types available with the Struts 2 Framework, we'll describe how different Struts 2 results are configured and implemented with the actions to be executed in an application. This has been discussed with a Struts 2 application created in the "Immediate Section" here.

Immediate Solutions

After this description of the different types of Result classes representing different result types, we are now ready to implement various result types in our application. The different results types are required to implement different strategy of view generation, which also includes the proper view format according to different view technologies used, like JSP, HTML, Freemaker and Velocity.

Developing project_results Application

The sample Web application developed here will focus on the result configuration provided in struts.xml file and their basic implementation details, like different attributes, and their use. The sample application created here follows a standard directly structure, shown in Figure 8.8.

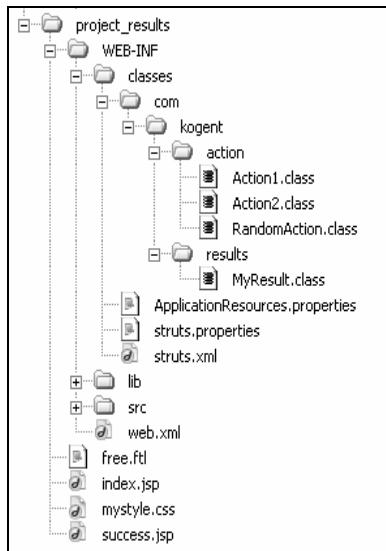


Figure 8.8: Directory structure of the project_results application.

The action classes used in this application are quite basic in the structure as our main aim is to show how to use different result types and how these result types behave to generate the appropriate format view for the user.

The two action classes used in this application are Action1 and Action2. The structure and details of these action classes are kept simple so that we can concentrate on the logic of different results implementation.

The first action class Action1 is `ServletRequestAware`, which has access to `HttpServletRequest` object to set the attribute in the request scope. The `String` property of this action class named `message` has its getter/setter methods also.

Here's the code, given in Listing 8.7, for Action1 action class (you can find Action1.java file in Code\Chapter 8\project_results\WEB-INF\src\com\kogent\action folder in CD):

Listing 8.7: Action1.java

```
package com.kogent.action;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts2.interceptor.ServletRequestAware;

import com.opensymphony.xwork2.Action;

public class Action1 implements Action, ServletRequestAware{

    HttpServletRequest request;

    String message;

    public void setServletRequest(HttpServletRequest request) {
        this.request=request;
    }

    public String getMessage() {
        return message;
    }

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

    public String execute() throws Exception {
        System.out.println("Action1.execute() running...");
        setMessage("Welcome from Action");
        request.setAttribute("name", "Kogent");
        return SUCCESS;
    }

}
```

The Action1.execute() methods invoke the setMessage() method and sets a name attribute in request scope.

Similarly, we can create another action class Action2. Action2 action class is similar to Action1 action class, except it does not have any input parameter. The Action2.execute() method sets an attribute named company in the request scope.

Here's the code, given in Listing 8.8, for this simple action class (you can find Action2.java file in Code\Chapter 8\project_results\WEB-INF\src\com\kogent\action folder in CD):

Listing 8.8: Action2.java

```
package com.kogent.action;
```

```

import javax.servlet.http.HttpServletRequest;
import org.apache.struts2.interceptor.ServletRequestAware;
import com.opensymphony.xwork2.Action;
public class Action2 implements Action, ServletRequestAware{
    HttpServletRequest request;
    public void setServletRequest(HttpServletRequest request) {
        this.request=request;
    }
    public String execute() throws Exception {
        System.out.println("Action2.execute() running...");
        request.setAttribute("company", "Kogent Solutions Inc.");
        return SUCCESS;
    }
}

```

Compile these source files and get the class files in WEB-INF/classes/com/kogent/action folder. One of the view (JSP file) created in this application is success.jsp. This JSP page simply prints two request scoped attributes namely name and company. The page also displays a property message by invoking getMessage() with the current value stack. The three <s:property> tags displays a default value “Not Set” if the specified attribute or property is not found.

Here’s the code, given in Listing 8.9, for success.jsp (you can find success.jsp file in Code\Chapter 8\project_results folder in CD):

Listing 8.9: success.jsp

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2><s:property value="message" default="Not Set"/></h2>
<br> <br>Name: <s:property value="#request.name" default="Not Set"/>
<br> <br>Company: <s:property value="#request.company" default="Not Set"/>
</body>
</html>

```

Another JSP page created here is our index.jsp page, which provides links to invoke various actions created with different sets of result types configured for them. Each hyperlink created here will search for a different action mapping, which shows the implementation of different result types.

Here's the code, given in Listing 8.10, for index.jsp page (you can find index.jsp file in Code\Chapter 8\project_results folder in CD):

Listing 8.10: index.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>

<h2><s:text name="app.welcome"/></h2>

<s:url id="url1" action="action1"/>
<s:a href="${url1}">dispatcher</s:a>

<br><br>
<s:url id="url2" action="action2"/>
<s:a href="${url2}">chain</s:a>

<br><br>
<s:url id="url3" action="action3"/>
<s:a href="${url3}">redirect</s:a>

<br><br>
<s:url id="url4" action="action4"/>
<s:a href="${url4}">redirect-action</s:a>

<br><br>
<s:url id="url5" action="action5"/>
<s:a href="${url5}">freemarker</s:a>

<br><br>
<s:url id="url6" action="action6"/>
<s:a href="${url6}">myresult</s:a>

</body>
</html>
```

Configuring different Results in Application

After creating action classes and JSP pages, we now need to provide different action mappings in our copy of struts.xml file with all configuration details for the results. For this, we are using framework results that are already defined in struts-default.xml file, shown in Listing 8.3. The package defined in our struts.xml file extends struts-default package from struts-default.xml file, hence, there is no need to provide any <result-types> and <result-type> elements to define result types as shown here:

```

<result-types>
    <result-type name="chain"
        class="com.opensymphony.xwork2.ActionChainResult"/>
    <result-type name="dispatcher"
        class="org.apache.struts2.dispatcher.ServletDispatcherResult"
        default="true"/>
    .
    .
    .
    .
</result-types>

```

This is not needed if we are using the standard framework result types and extending our package from struts-default package. If some custom result type is to be used, then it is required to be configured using `<result-type>` element.

Let's create a `struts.xml` file for this application, as shown in Listing 8.11 (you can find `struts.xml` file in `Code\Chapter 8\project_results\WEB-INF\classes` folder in CD):

Listing 8.11: `struts.xml` file

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">

    <result-types>
        <result-type name="myresult" class="com.kogent.results.MyResult"/>
    </result-types>

    <action name="action1" class="com.kogent.action.Action1">
        <result name="success" type="dispatcher">/success.jsp</result>
    </action>

    <action name="action2" class="com.kogent.action.Action2">
        <result name="success" type="chain">action1</result>
    </action>

    <action name="action3" class="com.kogent.action.Action1">
        <result name="success" type="redirect">/success.jsp</result>
    </action>

    <action name="action4" class="com.kogent.action.Action2">
        <result name="success" type="redirect-action">action1</result>
    </action>

    <action name="action5" class="com.kogent.action.Action1">
        <result name="success" type="freemarker">free.ftl</result>
    </action>

    <action name="action6" class="com.kogent.action.RandomAction">
        <result name="success" type="myresult">
            <param name="property">pr1</param>

```

```
</result>
<result name="error" type="myresult">
    <param name="property">pr2</param>
</result>
<result name="input" type="myresult">pr3</result>
</action>
</package>
</struts>
```

Let's understand the basic flow and implementation of different result types used in different application mappings provided in struts.xml file here (shown in Listing 8.11). Run your application to see the output of index.jsp page in Figure 8.9.

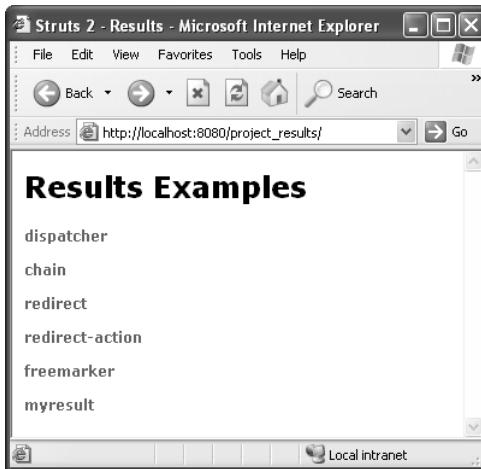


Figure 8.9: The index.jsp page showing different hyperlinks.

dispatcher Result

The first hyperlink created here is 'dispatcher', which invokes an action names action1. This executes Action1 as its action class. The only result configured for this action is for result code 'success' and the type of the result defined here is dispatcher.

Here's the first action mapping from struts.xml shown in Listing 8.11:

```
<action name="action1" class="com.kogent.action.Action1">
    <result name="success" type="dispatcher"/>/success.jsp</result>
</action>
```

This simply dispatches to success.jsp page after the execution of Action1.execute() method and the output of success.jsp page, after this execution, is shown in Figure 8.10.



Figure 8.10: The `success.jsp` with `Action1` as action and dispatcher as result type.

The `success.jsp` is showing only the message property of the current action class and the request attribute name set by `Action1`. The other request attribute `company` is not set here in this example and, thus, shows `Not Set`.

chain Result

The ‘chain’ hyperlink, created in Listing 8.10 and shown in Figure 8.9, can be clicked to invoke another action mapped with the name `action2`. The result configured for this action is set as chain for its result type. The action mappings, included in this example from Listing 8.11, are shown here:

```
<action name="action1" class="com.kogent.action.Action1">
    <result name="success" type="dispatcher"/>/success.jsp</result>
</action>

<action name="action2" class="com.kogent.action.Action2">
    <result name="success" type="chain">action1</result>
</action>
```

See how the flow the invocation of `action2` action invokes the action `Action2` and its `execute()` method is executed first. But the success returned as result code from the `Action2.execute()` method will result in action chaining and will invoke another action configured with name `action1`, as the result configure for `action2` shows. The action chaining with the help of Chain Result and Chain Interceptor makes sure that the request is shared among all the action in the sequence and the action context is also maintained throughout the sequence of action execution. See the output of `success.jsp` page in Figure 8.11, which is again being rendered for the user after the execution of action `Action1`.



Figure 8.11: The success.jsp with Action2, Action1 as action and chain as result type.

You can see the request parameter `company`, which is set by the `Action2.execute()` method and available here to be displayed, even after the execution of `Action1.execute()` method. It shows that the request is the same throughout the process of action chaining.

redirect Result

You can click over the redirect hyperlink created in Listing 8.10 and shown Figure 8.9 and invoke another action mapped with the name `action3`. This action mapping uses a single result configure with `redirect` as result type. See the action mappings used in this example from Listing 8.11:

```
<action name="action3" class="com.kogent.action.Action1">
    <result name="success" type="redirect"/>/success.jsp</result>
</action>
```

This action mapping makes the invocation of `Action1.execute()` method which consequently sets a request attribute "name" (see Listing 8.7). But the output, shown in Figure 8.12, does not show the request attribute set. Even the page is not able to display the message property from the current value stack. All this happens because of the result type `redirect`, which creates a new request and then renders `success.jsp`. Also the valued stack is not available now.



Figure 8.12: The success.jsp with Action1 as action and redirect as result type.

redirect-action Result

This is similar to the redirect example shown earlier but the only difference is the use of redirect-action as result type. The redirect-action is used to redirect another action, while the redirect can be used to redirect to any other URL. See the action mappings used in this example from Listing 8.11:

```

<action name="action1" class="com.kogent.action.Action1">
    <result name="success" type="dispatcher"/>/success.jsp</result>
</action>

<action name="action4" class="com.kogent.action.Action2">
    <result name="success" type="redirect-action">action1</result>
</action>

```

The Action2.execute() method is executed and the request attribute company is set. But this request attribute does not exist as soon as the user is redirected to another action (Action1). The Action1.execute() method is executed and the success.jsp page, as shown in Figure 8.13, is displayed to the user.



Figure 8.13: The success.jsp with Action2, Action1 as action and redirect-action as result type.

freemarker Result

This result type is used when some output using Freemarker template is to be shown. See the action mapping used in this example taken again from Listing 8.11:

```
<action name="action5" class="com.kogent.action.Action1">
    <result name="success" type="freemarker">free.ftl</result>
</action>
```

The result with the name `success` and type `freemarker` is set here, which also configures the `freemarker` page, `free.ftl`, that is rendered for the user as response.

Here's the code, given in Listing 8.12, for `free.ftl` (you can find `free.ftl` file in `Code\Chapter 8\project_results` folder in CD):

Listing 8.12: free.ftl

```
<html>
<head>
<title>Freemarker Page</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2>Freemarker Example</h2>
<%if Request.name?exists>
    <p>Request attribute "name" set : <b>${Request.name}</b>
<%else>
    <p>There is no request attribute "name".
<%if>
</body>
</html>
```

The output of this page, shown in Figure 8.14, shows the request attribute name set by the Action1.execute() method.

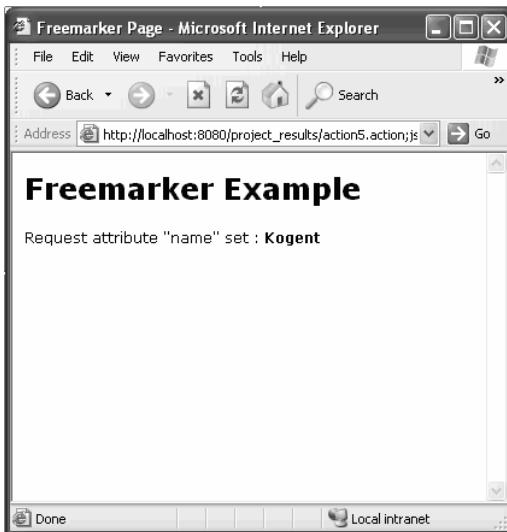


Figure 8.14: The free.ftl with Action1 as action and freemarker as result type.

Building Your Own Result Type

So far you are studying about the result types which are inbuilt and come along with the Struts 2 Framework APIs distribution. Now, we'll build our own result type. This whole activity is given to give you an idea about developing your own result types. Although the result types that come in Struts 2 package are sufficient enough to fulfill the requirements of your Web application, we sometimes need a different result type customized to our needs. We can create our own result type by creating a result class that implements com.opensymphony.xwork2.Result interface. Now, let's take a look at an example of creating a simple result type for debugging.

Suppose, you have an action that behaves erratically. Now you need to determine the values of the properties presented in the action. This particular action returns 'SUCCESS', 'INPUT' or 'ERROR' with equal probability. This action also sets up the properties pr1, pr2, and pr3 to have a value containing random numbers.

Here's the code, given in Listing 8.13, for RandomAction action (you can find RandomAction.java file in Code\Chapter 8\project_results\WEB-INF\src\com\kogent\action folder in CD):

Listing 8.13: RandomAction with the random behavior

```

package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
import java.util.Random;

public class RandomAction extends ActionSupport {

    private String pr1;
    private String pr2;
    private String pr3;
}

```

```
public String execute() throws Exception{

    int random = new Random().nextInt(100);
    pr1 = "pr1-" + random;
    pr2 = "pr2-" + random;
    pr3 = "pr3-" + random;
    if (random <= 33) {
        return SUCCESS;
    } else if (random <= 66) {
        return ERROR;
    } else {
        return INPUT;
    }
}

public void setPr1(String pr1)
{
    this.pr1= pr1;
}
public String getPr1()
{
    return this.pr1;
}
public void setPr2(String pr2)
{
    this.pr2= pr2;
}
public String getPr2()
{
    return this.pr2;
}
public void setPr3(String pr3)
{
    this.pr3= pr3;
}
public String getPr3()
{
    return this.pr3;
}

}
```

To debug an action, we have to determine the values for the various properties of this action after the execution of that action. So, to debug this particular action, we'll develop our own result type, `MyResult`.

Here's the source code, given in Listing 8.14, for `MyResult` (you can find `MyResult.java` file in `Code\Chapter 8\project_results\WEB-INF\src\com\kogent\results` folder in CD):

Listing 8.14: `MyResult.java`

```
package com.kogent.results;

import com.opensymphony.xwork2.Result;
```

```

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.Action;
import java.lang.reflect.Method;
public class MyResult implements Result{
    public static final String DEFAULT_PARAM = "property";
    String property;
    public void execute(ActionInvocation invocation) throws Exception {

        String resultCode = invocation.getResultCode();
        System.out.println("Result code: " + resultCode);

        Action action = (Action)invocation.getAction();

        String methodName = "get" + property.substring(0, 1).
            toUpperCase() + property.substring(1);
        System.out.println(methodName);

        Method method = action.getClass().getMethod(methodName,
            new Class[0]);

        Object o = method.invoke(action, new Object[0]);

        System.out.println(property + ": " + o);
    }

    public void setProperty(String property)
    {
        this.property = property;
    }
}

```

The MyResult result class will convert the property in the form of `xxx` to a method name of `getXXX`. After that, it will call the corresponding getter method to retrieve the value of the property.

After making the result definition, we'll configure this result in `struts.xml` file, so that we can use it for debugging our action. You can see the `<result-type>` element added to configure our customized result in Listing 8.11. A new action mapping is provided with the name="action6" using RandomAction as action class and MyResult as result type for all the three results configured for this action.

Here's the code snippet taken from Listing 8.11 to this configuration:

```

<result-types>
    <result-type name="myresult" class="com.kogent.results.MyResult"/>
</result-types>
<action name="action6" class="com.kogent.action.RandomAction">
    <result name="success" type="myresult">
        <param name="property">pr1</param>
    </result>
    <result name="error" type="myresult">
        <param name="property">pr2</param>
    </result>
    <result name="input" type="myresult">pr3</result>
</action>

```

If you go through the code given in Listing 8.11 carefully, you'll notice that the result we are using is defined in the <result-types> section of the struts.xml. All the other results that we may use in our Web application are defined in the struts-default.xml file, which is the reason for including that file in our struts.xml file. We have also given the class for the particular result type and made the result type as the default result type, i.e. if we do not specify the result type then it will be used by default.

The next important thing in the Listing 8.14 is the result for the action. In this portion, the result code is mapped to the actual result. We have defined three mappings for this case, namely 'success', 'error', and 'input'. In each of the mappings, a value for the property property is specified (pr1, pr2, pr3). This means that when the result will be instantiated, it will call the setProperty which will specify the required value.

The mapping for the success is the most formal way of defining the result. In mapping for success, both the parameters (type and property) are specified. The mapping for the other results, i.e. error and input, are less verbose. To execute this example using our own result class (MyResult class), you can click on myresult hyperlink created in index.jsp (See Figure 8.9) which invokes action6 action (See Listing 8.11 for this action mapping). The invocation of action6 action does not bring a new page for you as it prints output texts in the log file instead. You can see the log file for the output of various execution of this example, as shown here:

```
Result code: input
getPr3
pr3: pr3-98
Result code: success
getPr1
pr1: pr1-1
Result code: error
getPr2
pr2: pr2-66
Result code: success
getPr1
pr1: pr1-13
```

This example was given to provide you an idea about developing your own result type. Usually, you do not need to develop your own result type because there are so many result types that are already available in Struts 2 and you can use them directly in your Struts 2 based Web application.

This chapter focused on the various result types available in the Struts 2 Framework, along with the configuration of the result and Global Results. The "Immediate Solutions" section, next, concentrated on about the practical aspect of setting the predefined Result types for different actions configured in struts.xml file. We also studied, in detail, the configuration of different result types available in the Struts 2 and proceeded towards developing our own result types.

The next chapter deals with all the validation support provided by the Struts 2 Framework for validating data through various simple and flexible ways.



9

Performing Validations in Struts 2

If you need an immediate solution to:

See page:

Creating Basic Validation Example	346
Using Field Validators	351
Using Non-Field Validators	359
Performing Validation Using Annotation	364

In Depth

In this chapter, we'll discuss about Validation Framework of Strut 2 applications. Struts 2 Validation Framework is based on XWork Validation Framework. The XWork Validation Framework is a powerful addition to the Struts 2 Framework. This framework allows you to manage the validations in a separate configuration file, where they can be reviewed and modified without changing Java or JavaServer Page code, because localization and validations are tied to the business tier, not to the presentation tier. The XWork Validation Framework also provides several basic validators that will meet most of the daily requirements. It also allows you to add Custom validators for special requirements. You can also use the original Struts validate() method in tandem with the Struts validators, if required. Struts validator also provides the localization feature same as the main Struts Framework. It allows you to share the standard message resource file with the main framework, providing an error free solution for the translators, you are working with. Struts validators provide validating input for meeting the requirements of complex applications.

Understanding XWork Validation support in Struts 2

When a client submits data through a form, then it is necessary to check whether the data submitted by the client is valid or in proper format as required. Validating form data is essential to prevent incorrect data from getting into your applications. Validations can also be performed on the back-end, i.e. validation on the basis of the database used in your web application. However, if you perform validation on back-end services then the client cannot show it. Suppose, you perform validation on back-end systems and the welcome page of your application is displayed before the client. This welcome page contains a simple login form, which requires an email address. The proper format of the required email address is defined on the back-end system, but the client is unaware of this format. Now if the client enters the email address in the wrong format, then an error message is displayed. Therefore, it is a good idea to show users where they have made a mistake and what they need to do to correct it. Validating form data and providing meaningful error messages is one of the keys to providing feedback to the user.

Struts 2 is based on a Validation Framework, which is provided by XWork. Validations can be performed in the code of your action class, but this approach can mess your code and become complicated when you want different validations on different fields in different scenarios. For example, you may need to check for empty fields, or the format of the entered data, or both of these, or none of these. Implementing validation checks declaratively is always preferred over hard coding of logic in the classes. These concerns led to the development of the XWork Validation Framework, which is part of XWork. The Validation Framework uses external metadata in the form of XML files to describe what validation should be performed on your action.

Understanding Validators in Struts 2

A class which implements `com.opensymphony.xwork2.validator.Validator` interface can be defined as validator class or simply a validator. Validators are called by the framework to validate an object by accessing its properties. We have a set of validators, which are already defined by the XWork Validation Framework and associated classes are bundled in the Struts 2 API. Hence, these validators are

also known as bundled validators. These bundled validators are defined in an XML file called `validators.xml`. The `validators.xml` file must be available in your classpath (`/WEB-INF/classes`). However, in case there is no Custom validator used in the application, then there is no need to put this file in the classpath. All the bundled validators are automatically registered with `ValidatorFactory`, as the `com/opensymphony/xwork2/validator/validators/default.xml` file containing the configuration for all bundled validators is loaded. You can see all Struts 2 bundled validators configured shown in Listing 9.1.

But if, Custom validator is defined and you need to create a `validators.xml` and put it in your classpath, always remember to configure all the bundled validators in `validators.xml` file in the similar way as defined in Listing 9.1. Once a `validators.xml` is detected in the classpath, the `com/opensymphony/xwork2/validator/validators/default.xml` file will not be automatically loaded; it is only loaded when a custom `validators.xml` cannot be found in the classpath.

Here's the sample code, given in Listing 9.1, for `validators.xml` file that includes all the bundled validators:

Listing 9.1: `com/opensymphony/xwork2/validator/validators/default.xml` file

```
<validators>
<validator name="required"
class="com.opensymphony.xwork2.validator.validators.RequiredFieldValidator"/>

<validator name="requiredstring"
class="com.opensymphony.xwork2.validator.validators.RequiredStringValidator"/>

<validator name="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"/>

<validator name="double"
class="com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"/>

<validator name="date"
class="com.opensymphony.xwork2.validator.validators.DateRangeFieldValidator"/>

<validator name="expression"
class="com.opensymphony.xwork2.validator.validators.ExpressionValidator"/>

<validator name="fieldexpression"
class="com.opensymphony.xwork2.validator.validators.FieldExpressionValidator"/>
<validator name="email"
class="com.opensymphony.xwork2.validator.validators.EmailValidator"/>
<validator name="url"
class="com.opensymphony.xwork2.validator.validators.URLValidator"/>

<validator name="visitor"
class="com.opensymphony.xwork2.validator.validators.VisitorFieldValidator"/>

<validator name="conversion"
class="com.opensymphony.xwork2.validator.validators
.ConversionErrorFieldValidator"/>

<validator name="stringlength"
class="com.opensymphony.xwork2.validator.validators
```

```
.StringLengthFieldvalidator"/>
<validator name="regex"
class="com.opensymphony.xwork2.validator.validators.RegexFieldvalidator"/>
</validators>
```

Listing 9.1 shows the bundled validators with their name and description. All validators have their properties, like `defaultMessage`, `messageKey`, and `shortCircuit`, configured using built-in attributes in the XML file. In addition, all validators, except the Expression validator, support a `fieldName` property that is normally set by the Field validators inside a `<field>` element. Additional properties shall be discussed shortly.

Most of the validators listed in the Listing 9.1 are relatively simple. They do things like checking whether the input text field is empty or null, whether a value is within the specified range, or whether a String is in some defined format, like an email address, a URL, and so on.

Let's now discuss the additional properties of these validators—all the bundled validators for their functionalities and their implementations.

RequiredFieldValidator Class

The `RequiredFieldValidator` is used to check whether your input field is not null. When you use this validator, you have to pass one parameter, i.e. `fieldName`. This `fieldName` validator is simply your field name that is passed with `<param name>` to validate your input field. If you are using `field-validator` syntax then it is not needed, but if plain-validator syntax is used then it is used. Here's Listing 9.2 showing how you can use `RequiredFieldValidator`:

Listing 9.2: Sample configuration for RequiredFieldValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "required" >
        < param name = "fieldName" > username </param >
        < message >username must not be null < / message >
    < / validator >

    < ! - - Field Validator Syntax - - >
    < field name = "username" >
        < field-validator type = "required" >
            < message >username must not be null < /message >
        < /field-validator >
    < /field >
< / validators >
```

The code snippet, given in Listing 9.2, shows how you can check input field 'username' that contains null value or an empty text. For example, if we pass `username` text field to null or empty then it gives an error message 'username must not be null'.

NOTE

You can internationalize the message to be displayed on validation errors by using `<message key="some.key"/>` instead of `<message></message>`. The value for key `some.key` will be searched from the resource bundle according to the current locale. Every thing about Struts 2 support for internationalization has been discussed in chapter 10.

RequiredStringValidator Class

RequiredStringValidator checks whether a String field does not contain null value or has a length greater than 0. For example, the text field ‘username’ must contain some string of length more than zero. For using this validator, you have to pass two parameters—`fieldName` and `trim`. The `fieldName` parameter validates the name of the text field, and is needed to validate only if plain-validator syntax is used.

Here’s Listing 9.3 that shows the `trim` parameter, which determines whether it will trim the string before performing the length check and, if unspecified, the string will be trimmed

Listing 9.3: Sample configuration for RequiredStringValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "requiredstring" >
        < param name = "fieldName" > username </param >
        < param name = "trim" > true </param >
        < message > username is required < / message >
    < / validator >

    < ! - - Field Validator Syntax - - >
    < field name = "username" >
        < field-validator type = "requiredstring" >
            < param name = "trim" > true </param >
            < message > username is required < /message >
        < /field-validator >
    < /field >
< / validators >
```

In other words, Listing 9.3 shows how you can use `RequiredStringValidator` in your XML file to validate Struts2 application. This code shows how the input text ‘username’. For example, if we pass `username` text field to 61Sam or “”, it gives ‘username is required’ message.

IntRangeFieldValidator Class

This validator checks whether an Integer value entered in the field is within the specified range. For example, the field to take age of an employee should be validated for the range 15-60, as the age of employee should not be less than 15 and greater than 60. We can use this validator to validate a field for a given range. To use this validator you have to pass three parameters with `<param name>` of XML validation—`fieldname`, `min`, and `max`. The descriptions for each parameter are as follows:

- ❑ `fieldName`—It represents the field name this validator is validating. If you are using field-validator syntax then it is not needed but if plain-validator syntax is used then it is used.
- ❑ `min`—Integer property that represents the minimum allowable value. Defaults to null, which doesn’t check the minimum.
- ❑ `max`—Integer property that represents the maximum allowable value. Defaults to null, which doesn’t check the maximum.

Here’s Listing 9.4 showing how you can use `IntRangeFieldValidator` in your XML file to validate a field for the range of integer entered in the field, say `age`:

Listing 9.4: Sample configuration for IntRangeFieldValidator

```

< validators>
    < ! -- Plain validator Syntax -- >
    < validator type = "int" >
        < param name = "fieldName" > age </param >
        < param name = "min" > 15 </param >
        < param name = "max" > 60 </param >
        < message > Age needs to be between ${min} to ${max} < / message >
    < / validator >

    < ! -- Field validator Syntax -- >
    < field name = "age" >
        < field-validator type = "int" >
            < param name = "min" > 15 </param >
            < param name = "max" > 60 </param >
            < message > Age needs to be between ${min} to ${max} < /message >
        < /field-validator >
    < /field >
< / validators >

```

Listing 9.4 shows how the input field ‘age’ contains only integer values within the range ‘15’ to ‘60’. If we fill age text field with 61 or 14, it gives error message—‘Age needs to be between 15 to 60.’

DoubleRangeFieldValidator Class

This validator is same as IntRangeFieldValidator that checks whether the double or double value is in the specified range, e.g. room temperature should not be less than 18.33 and greater than 30.50. To use the DoubleRangeFieldValidator validator, pass three parameters with <param-name> in .xml validation file. Different parameters for this validator are as follows:

- ❑ **fieldName**—This parameter validates the name of the field of the application. If you are using field-validator syntax then it is not needed. But, if plain-validator syntax is used then it is used.
- ❑ **minInclusive**—This parameter checks the minimum inclusive value in `FloatValue` format specified by Java language. Defaults to null, which doesn’t check the minimum inclusive value.
- ❑ **maxInclusive**—This parameter checks the maximum inclusive value in `FloatValue` format specified by Java language. Defaults to null, which doesn’t check the maximum inclusive value.
- ❑ **minExclusive**—This parameter checks the minimum exclusive value in `FloatValue` format, specified by Java language. Defaults to null, which doesn’t check the minimum exclusive value.
- ❑ **maxExclusive**—This parameter checks the maximum exclusive value in `FloatValue` format, specified by Java language. Defaults to null, which doesn’t check the maximum exclusive value.

Here’s the code, given in Listing 9.5, using each parameter:

Listing 9.5: Sample configuration for DoubleRangeFieldValidator

```

< validators>
    < ! -- Plain validator Syntax -- >
    < validator type = "double" >
        < param name = "fieldName" > Percentage </param >
        < param name = "minInclusive" > 10.20 </param >

```

```
< param name = "maxInclusive" > 35.33 </param >
< message > Percentage needs to be between ${minInclusive} and ${maxInclusive}
to (Inclusive Message) < / message >
< / validator >

< ! - - Field Validator Syntax - - >
< field name = "Percentage" >
    < field-validator type = "double" >
        < param name = "minExclusive" > 33.50 </param >
        < param name = "maxExclusive" > 99.99 </param >
    < message > Percentage needs to be between ${minExclusive} and ${maxExclusive}
to (Exclusive Message) < / message >
    < /field-validator >
< /field >
< / validators >
```

Listing 9.5 shows how the input field percentage contains only double value with range from 33.50 to 99.99. If we fill the percentage text field with 32.20 or 100.00, it gives an error message – 'Percentage needs to be between 33.50 to 99.99'.

DateRangeFieldValidator Class

Date is one of the most important fields of any Web application. For developing a well designed Web application software, each input field must be in a specified range before submitting the form on to the server. Struts 2 provides the DateRangeFieldValidator validator that is used to check whether your date is within the specified range. DateRangeFieldValidator checks to make sure that the Date field value is in a given range of dates. If no date converter is specified, XworkBasicConverter will automatically do the date conversion, which, by default, uses the Date.SHORT format using a programmatically specified locale, else it falls back to the system default locale.

Suppose you want to check whether the date of birth of a student is between 01/01/1980 to 01/01/1999. This can be done by using the DateRangeFieldValidator validator. Here you have to pass three parameters, which can be set using the <param-name> element. The parameters to be passed are as follows:

- ❑ **fieldName** – It represents the field name this validator is validating. If you are using field-validator syntax then it is not needed, but if plain-validator syntax is used then it is used.
- ❑ **min** – This is the Date property that represents the minimum allowable value. Defaults to null, which doesn't check the minimum.
- ❑ **max** – This is the Date property that represents the maximum allowable value. Defaults to null, which doesn't check the maximum.

Here's the code, given in Listing 9.6, that shows how the DateRangeFieldValidator is used in your validation XML file.

Listing 9.6: Sample configuration for DateRangeFieldValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "date" >
        < param name = "fieldName" > birthday </param >
        < param name = "min" > 01/01/1980 </param >
        < param name = "max" > 01/01/1999 </param >
    < message > Birthday needs to be between ${min} and ${max} < / message >
```

```

< / validator >

< ! -- Field Validator Syntax -- >
< field name = "birthday" >
    < field-validator type = "date" >
        < param name = "min" > 01/01/1980 </param >
        < param name = "max" > 01/01/1999 </param >
        < message > Birthday needs to be between ${min} and ${max}
            < /message >
        < /field-validator >
    < /field >
< / validators >

```

By following Listing 9.6, when we enter a value that is before 01/01/1980 or after 01/01/1999 in the student's 'birthday' field, then it will give an error message—'Birthday needs to be between 01-01-1980 and 01/01/1999'.

ExpressionValidator Class

It is a non-field validator that evaluates the Boolean value in the OGNL expression, e.g. `user.name != Ambrish`. These validators allow you to create powerful validations using just XML and your existing model. The Expression validator should be applied when validations are not specific to one field, because it adds action-level error messages. This validator is used to validate your expression and returns the Boolean value 'true' or 'false.' If the input field returns 'true' then the expression is correct, otherwise the expression is wrong.

Here's Listing 9.7 showing you how to use `ExpressionValidator` in your validation XML configuration file:

Listing 9.7: Sample configuration for `ExpressionValidator`

```

< validators>
    < validator type = "expression" >
        < param name = "expression" >..... </param >
        < message > Not an OGNL expression < / message >
    < / validator >
< / validators >

```

By following Listing 9.7, when we put a wrong expression, an error message—'Not an OGNL expression'—gets displayed.

FieldExpressionValidator Class

This validator is used to check whether your input field contains OGNL expression and returns a Boolean value. If the OGNL expression is valid, then the `FieldExpressionValidator` will return a true value, otherwise it returns a false value. A Field validator evaluates any OGNL expression used for evaluating a Boolean, e.g. `user.name != "Ambrish"`. These validators allow you to create powerful validations using just XML and your existing model. The Expression validator should be applied when validations are not specific to one field, because it adds action-level error messages. The `fieldexpressionvalidator` is a Field validator that adds error messages specific to that particular field. Except for the location where the error messages are stored, `ExpressionValidator` and `FieldExpressionValidator` are the same. To use the `FieldExpressionValidator` validator, you

have to pass two parameters with <param-name> of your validation file. These parameters are as follows:

- ❑ expression – An OGNL expression that evaluates to true or false. A value of true means the object or field is valid. A value of false or a non-Boolean value means that the object or field is invalid and causes an error message to be added.
- ❑ fieldName – It is the field name this validator is validating. If you are using field-validator syntax then it is not needed, but if plain validator syntax is used then it is used.

Here's Listing 9.8 showing how you can use the FieldExpressionValidator in your validation file:

Listing 9.8: Sample configuration for FieldExpressionValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "fieldexpression" >
        < param name = "fieldName" > birthday </param >
        < param name = "expression" > < ! [ CDATA[ #mybirthday
            <#myfatherbirthday ] ] > </param >
        < message > My Birthday needs to be less than my Father's
            Birthday</message>
    < / validator >

    < ! - - Field Validator Syntax - - >
    < field      name = "birthday" >
        < field-validator type = "fieldexpression" >
            < param name = "expression" > < ! [ CDATA[
                #mybirthday < #myfatherbirthday ] ] > </param >
            < message > My Birthday needs to be less than my
                Father's Birthday</message>
        < /field-validator >
    < /field >
< / validators >
```

Listing 9.8 describes the FieldExpressionValidator that shows how you can use field expression. Suppose you put your birthday that is greater than your father's birthday. In that case, you'll get an error message saying 'My Birthday needs to be less than my Father's Birthday'.

EmailValidator Class

The EmailValidator checks whether a given String field is in a valid Email address format. For example, it checks whether the @ character, and .com, .org, etc. exists in your input text field.

This validator requires only one parameter, i.e. thefieldname that this validator is validating. If you are using field-validator syntax then it is not needed, but if plain-validator syntax is used then it is used.

Here's Listing 9.9 showing the usage of EmailValidator:

Listing 9.9: Sample configuration for EmailValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "email" >
        < param name = "fieldName" > Enter Email </param >
        < message > Provide a Valid Email Address < / message >
```

```
< / validator >

< ! -- Field Validator Syntax -- >
< field name = "EnterEmail" >
    < field-validator type = "email" >
        < message > Provide a Valid Email Address < /message >
        < /field-validator >
    < /field >
< / validators >
```

In Listing 9.9, when you put a wrong email address, e.g. `yourname@yahoo.xyz`, an error message—'Provide a valid Email Address'—gets displayed.

URL Validator Class

This `URLValidator` simply checks whether your input text field contains valid URL or not. To use this validator you need only one parameter, i.e. `fieldName`, which is passed with `<param name>` of your validation file. The `fieldName` represents the name of the field which you wish to validate in your application. If you are using the `field-validator` syntax, then it is not needed. However, if plain-validator syntax is used then it is used.

Here's the code, given in Listing 9.10, to understand how you can validate an input text for accepting a valid URL only:

Listing 9.10: Sample configuration for `URLValidator`

```
< validators>
    < ! -- Plain Validator Syntax -- >
    < validator type = "url" >
        < param name = "fieldName" > homepage </param >
        < message > Invalid HomePage URL < / message >
    < / validator >

    < ! -- Field Validator Syntax -- >
    < field name = "homepage" >
        < field-validator type = "url" >
            < message > Invalid HomePage URL < /message >
        < /field-validator >
    < /field >
< / validators >
```

VisitorFieldValidator Class

`VisitorFieldValidator` allows the validation to be run against the value of the field to which this validator is applied. Suppose you have an action `CreateUser` and a class `User`, then if a visitor Field validator is applied to the `user` object of the `CreateUser` action, it applies the validations mapped for the `User` class. The framework uses the validations defined in `*-validation.xml` files for the object types of the property value. It means that the validator allows you to forward the validation to object properties of your action using the object's own validation files. This allows you to use the Model-Driven development pattern and manage your validations for your models in one place, where they belong, next to your model classes. The `VisitorFieldValidator` can handle simple Object

properties, Collections of Objects, and simple arrays. To use the URLValidator you have to pass three parameters with `<param name>`. These parameters are as follows:

- ❑ `fieldName` – It represents the field name this validator is validating. If you are using the field-validator syntax, then it is not needed. However, if plain-validator syntax is used, then it is used.
- ❑ `context` – It allows you to specify a different context under which to validate this object. By default, the context is the alias name of other action whose validation mapping is being applied here. It is an optional parameter.
- ❑ `appendPrefix` – This parameter tells the visitor Field validator whether to append the name of this property to the full property name when adding error messages. For example, if the visitor Field validator is added to the `User` property, and the `appendPrefix` is set to true, then the field error messages for the `firstName` property of the `User` adds field error messages for `user.firstName` to the action. By default, this is true. For Model-Driven actions where the visitor Field validator is applied to the model property, this should be set to false. This parameter is optional.

Listing 9.11 shows how you can use the `VisitorFieldValidator`:

Listing 9.11: Sample configuration for `VisitorFieldValidator`

```
< validators>
    < ! -- Plain Validator Syntax -- >
    < validator type = "visitor" >
        < param name = "fieldName" > user </param >
        < param name = "context" > myContext </param >
        < param name = "appendPrefix" > true </param >
    < / validator >

    < ! -- Field Validator Syntax -- >
    < field name = "user" >
        < field-validator type = "visitor" >
            < param name = "context" > myContext </param >
            < param name = "appendPrefix" > true </param >
        < /field-validator >
    < /field >
< / validators >
```

ConversionErrorFieldValidator Class

`ConversionErrorFieldValidator` checks if any conversion error had occurred for this field. The validator checks whether a type conversion error had occurred when setting the value on this field and uses the type-conversion framework to create the correct field error message to be added for this field. This Field validator should only be used if the `conversionError` interceptor is not applied and you want to handle the setting conversion error messages on a per-field basis. When you use this validator, you have to pass only one parameter, i.e. `fieldName` and `<param name>`.

The `fieldName` parameter is used for validating any conversion error. When you are using field-validator syntax then it is not needed. However, if plain-validator syntax is used, then it is used.

Here's Listing 9.12 showing you how use this validator:

Listing 9.12: Sample configuration for ConversionErrorFieldValidator

```

< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "conversion" >
        < param name = "fieldName" > username </param >
        < message > Conversion Error Occurred < / message >
    < / validator >

    < ! - - Field Validator Syntax - - >
    < field name = "username" >
        < field-validator type = "conversion" >
            < message > Conversion Error Occurred < /message >
        < /field-validator >
    < /field >
< / validators >

```

In Listing 9.12 (ConversionErrorFieldValidator) when you put wrong conversion username then it gives an error message – 'Conversion Error Occurred'.

StringLengthFieldValidator Class

The `StringLengthFieldValidator` validator is used to validate a string for the number of character. This validator checks and makes sure that the length of a `String` property's value is within a specified range. If the `minLength` parameter is specified, it will make sure that the string has at least that many characters. If the `maxLength` parameter is specified, it will make sure that the `String` has at the most that many characters. The `trim` parameter checks whether to trim the `string` before checking the length of the string. For example, the length of password should not be less than 6 characters. Before using the `StringLengthFieldValidator` validator, you have to pass some parameter with `<param name>` of your validation file. These parameters are as follows:

- ❑ `fieldName` – It specifies the field name this validator is validating. If you are using field-validator syntax, then it is not needed. However, if plain-validator syntax is used, then it is used.
- ❑ `minLength` – The value set for this property represents the minimum allowable length. Defaults to null, which doesn't check the minimum.
- ❑ `maxLength` – The value set for this property represents the maximum allowable length. Defaults to null, which doesn't check the maximum.
- ❑ `trim` – It trims the field name value before validating. It is a Boolean property that tells the validator whether to call `trim()` to remove extra whitespace before checking the `String`'s value. It's default value is `true`.

Here's Listing 9.13 showing you how to use `StringLengthFieldValidator`:

Listing 9.13: Sample configuration for StringLengthFieldValidator

```

< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "stringlength" >
        < param name = "fieldName" > username </param >
        < param name = "minLength" > 5 </param >
        < param name = "maxLength" > 10 </param >
        < param name = "trim" > true </param >

```

```
< message > Username needs to be 10 characters long < / message >
< / validator >

< ! - - Field Validator Syntax - - >
< field name = "username" >
    < field-validator type = "stringlength" >
        < param name = "fieldName" > username </param >
        < param name = "minLength" > 5 </param >
        < param name = "maxLength" > 10 </param >
        < param name = "trim" > true </param >
        < message > Username needs to be 10 characters long
        < / message >

    < /field-validator >
< /field >
< / validators >
```

In Listing 9.13, when you enter a ‘username’ having less than 5 characters or greater than 10 characters, then it gives an error message – ‘Username needs to be 10 characters long’.

RegexFieldValidator Class

The RegexFieldValidator validates a String field using a regular expression. This validator checks the value of a field against the given regular expression. The parameters used in RegexFieldValidator are as follows:

- ❑ **expression** – It is a regular expression that evaluates to true or false. A value of true means the field is valid. A value of false or a non-Boolean value means the field is invalid and causes an error message to be added.
- ❑ **fieldName** – It specifies the field name this validator is validating. If you are using field-validator syntax, then it is not needed. However, if plain-validator syntax is used, then it is used.
- ❑ **caseSensitive** – A Boolean value that sets whether the expression should be matched for its case-sensitivity. Its default value is true.

Here’s Listing 9.14 showing you how RegexFieldValidator is used:

Listing 9.14: Sample configuration for RegexFieldValidator

```
< validators>
    < ! - - Plain Validator Syntax - - - >
    < validator type = "regex" >
        < param name = "fieldName" > MyCode </param >
        < param name = "expression" > < ! [ CDATA[[aAbBcCcDdEe]
[12345] [fFgGhH] [4567]]] > </param >
    < / validator >
    < ! - - Field Validator Syntax - - >
    < field name = "MyCode" >
        < field-validator type = "regex" >
            < param name = "expression" > < ! [ CDATA[[aAbBcCcDdEe]
[12345] [fFgGhH] [4567]]] > </param >
        < /field-validator >
    < /field >
< / validators >
```

Defining Validators Scopes

Validator scopes indicate whether a validator can act on a single field accessible through an action or on the full action context which involves more than one field. In Struts 2, there are two types of Validators scope—Field validator and Non-field validator.

Field Validator

The Field validators act on single fields accessible through an action. A validator is more generic and can do validations in full action context, involving more than one field in validation rule. Most validations can be defined on per field basis. This should be preferred over non-field validation wherever possible because the Field validator messages are bound to the related field and will be presented next to the corresponding input element in the respective view.

Here's Listing 9.15 showing the usage of Field validator:

Listing 9.15: Sample configuration for Field validator

```

< validators >
< ! -- Field Validator Syntax for RequiredFieldValidator -- >
    < field name = "username" >
        < field-validator type = "required" >
            < message > username must not be null < /message >
        < /field-validator >
    < /field >

    < ! -- Field Validator Syntax for RequiredStringValidator-- -- - >
    < field name = "username" >
        < field-validator type = "requiredstring" >
            < param name = "trim" > true </param >
            < message > username is required < /message >
        < /field-validator >
    < /field >

    < ! -- Field Validator Syntax for IntRangeFieldvalidator -- >
    < field name = "age" >
        < field-validator type = "int" >
            < param name = "min" > 15 </param >
            < param name = "max" > 35 </param >
            < message > Age needs to be between ${min} and
                ${max} < /message >
        < /field-validator >
    < /field >

    < ! -- Field Validator Syntax for DoubleRangeFieldValidator -- >
    < field name = "Percentage" >
        < field-validator type = "double" >
            < param name = "minExclusive" > 05.20 </param >
            < param name = "maxExclusive" > 85.33 </param >
            < message > Percentage needs to be between
                ${minExclusive} and ${maxExclusive}
                (Exclusive Message) < / message >
        < /field-validator >
    < /field >

```

```
< ! -- Field Validator Syntax for FieldExpressionvalidator -- >
< field name = "birthday" >
    < field-validator type = "fieldexpression" >
        < param name = "expression" > < ! [ CDATA[
            #mybirthday < #myfatherbirthday ] ] >
        </param >
        < message > My Birthday needs to be less than my
            Father's Birthday</message>
    < /field-validator >
< /field >

< ! -- Field Validator Syntax for Emailvalidator -- >
< field name = "Enter Email" >
    < field-validator type = "email" >
        < message > Provide a Valid Email Address < /message >
    < /field-validator >
< /field >

< ! -- Field Validator Syntax for URLValidator -- >
< field name = "HomePage" >
    < field-validator type = "url" >
        < message > Invalid HomePage URL < /message >
    < /field-validator >
< /field >

< ! -- Field Validator Syntax for VisitorFieldValidator -- >
< field name = "user" >
    < field-validator type = "visitor" >
        < param name = "context" > myContext </param >
        < param name = "appendPrefix" > true </param >
    < /field-validator >
< /field >

< ! -- Field Validator Syntax for StringLengthFieldvalidator -- >
< field name = "username" >
    < field-validator type = "stringlength" >
        < param name = "fieldName" > username </param >
        < param name = "minLength" > 5 </param >
        < param name = "maxLength" > 10 </param >
        < param name = "trim" > true </param >
        < message > Username needs to be 10 characters long
        < / message >
    < /field-validator >
< /field >
```

Non-Field Validator

Non-field validators only add action level messages. These validators are mostly domain specific and therefore often need custom implementations. Expression Validator is an example of Non-field validator

Here's Listing 9.16 showing how you can use Non-field validator:

Listing 9.16: Sample configuration for Non-field validators

```
< validators>
    < validator type = "expression" >
        < param name = "expression" >..... </param >
        < message > Not an OGNL expression < / message >
    < / validator >
< / validators >
```

Registering Validators

Validators must be registered with the `com.opensymphony.xwork2.validator.ValidatorFactory`. This may be done programmatically using the `registerValidator` static method of the `ValidatorFactory` class. The signature for `registerValidator ()` method is `Public static void registerValidator (String valname, String classname)`. This method registers the given validator to the existing map of validators. In this method, the parameter `valname` is the name of the validator to be added and `classname` is the fully qualified classname of the validator.

The simplest way to register a validator is to add a file name `validators.xml` in the root of the classpath (/ WEB-INF/classes) that declares all the validators you mean to use.

Here's Listing 9.17 showing a `validators.xml` file, which includes all the bundled validators:

Listing 9.17: A sample `validators.xml` file

```
< validators >
<validator name ="required"
class="com.opensymphony.xwork2.validator.validators.RequiredFieldValidator" />
<validator name ="requiredstring"
class="com.opensymphony.xwork2.validator.validators.RequiredStringValidator"/>
<validator name ="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator" />
<validator name ="double"
class="com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"/>
<validator name ="date"
class="com.opensymphony.xwork2.validator.validators.DateRangeFieldValidator" />
<validator name ="expression"
class="com.opensymphony.xwork2.validator.validators.ExpressionValidator" />
<validator name ="fieldexpression"
class="com.opensymphony.xwork2.validator.validators.FieldExpressionValidator" />
<validator name ="email"
class="com.opensymphony.xwork2.validator.validators.EmailValidator" />
<validator name ="url"
class="com.opensymphony.xwork2.validator.validators.URLValidator" />
<validator name ="visitor"
class="com.opensymphony.xwork2.validator.validators.VisitorFieldValidator" />
<validator name ="conversion"
class="com.opensymphony.xwork2.validator.validators.
ConversionErrorFieldValidator" />
<validator name ="stringlength"
```

```
class="com.opensymphony.xwork2.validator.validators
.StringLengthFieldValidator"/>
<validator name ="regex"
class="com.opensymphony.xwork2.validator.validators.RegexFieldValidator" />
</validators>
```

Creating Custom Validators

In addition of using bundled Struts 2 validators, you can build your own Custom validators for different and more complex applications. The Validation Framework allows Custom validators to be built and applied in the same declarative fashion as other bundled validators. Implementing a Custom validator is as simple as creating a class that extends the `com.opensymphony.xwork2.validator.validators.ValidatorSupport` (for Global validator) or `com.opensymphony.xwork2.validator.validators.FieldValidatorSupport` (for FieldValidator).

Here's Listing 9.18 showing the source code for building a Custom validator named `MyStringLengthFieldValidator`:

Listing 9.18: Source code of `MyStringLengthFieldValidator` class

```
package com.kogent.validators;

import com.opensymphony.xwork2.validator.ValidationException;
import com.opensymphony.xwork2.validator.validators.FieldValidatorSupport;

public class MyStringLengthFieldValidator extends FieldValidatorSupport{
    private int maxLength = -1;
    private int minLength = -1;
    private boolean dotrim = true;

    public void setMinLength ( int minLength )
    {
        this.minLength = minLength;
    }
    public void setMaxLength ( int maxLength )
    {
        this.maxLength = maxLength;
    }
    public void setTrim ( boolean trim )
    {
        dotrim = trim;
    }
    public int getMinLength ( )
    {
        return minLength;
    }
    public int getMaxLength ( )
    {
        return maxLength;
    }
    public boolean getTrim ( )
```

```

{
    return dotrim;
}
public void validate ( Object obj ) throws validationException
{
    String fieldName = getFieldName( );
    String val = ( String ) getFieldValue( fieldName , obj ) ;
    if ( dotrim )
    {
        val = val.trim ( );
    }
    if (( minLength > -1 ) && ( val.length ( ) <  minLength ) )
    {
        addFieldError ( fieldName , obj );
    }
    else if (( maxLength > -1 ) && ( val.length ( ) >  maxLength ) )
    {
        addFieldError ( fieldName , obj );
    }
}
}

```

The `MyStringLengthFieldValidator` adds the properties `maxLength`, `minLength`, and `trim` and uses them to check against the length of the String. The `getFieldName()`, `getFieldValue()`, and `addFieldError()` methods are implemented in the abstract base class and make it trivial to write new validators. This validator works similar to the `StringLengthFieldValidator`.

Once you have created a new validator class, all that is left is to register it with the Validation Framework, to be able to use it as a named validator in your `*--validation.xml` files. All that has to be done is to add the following line in the `validators.xml` file which is to be stored in application's classpath:

```
<validator name="mystringlength"
class="com.kogent.validators.MyStringLengthValidator"/>
```

When implementing Custom validators and registering them using `validators.xml` file, all the required bundled validators should also be configured in the `validators.xml` file, as shown in Listing 9.17.

Defining Validation Rules

Validation rules for Struts 2 Framework can be defined in the following three different ways:

- Per Action class
- Per Action alias
- Inheritance hierarchy and interfaces implemented by action class.

Per Action Class

We can define validation rules per action class, i.e. validation rules for a single action class. This can be done by creating an XML file, named `ActionName-validation.xml`. This is known as Action-level validation. To create an Action-level validation, create a file `ActionClass-validation.xml` at the

same location where the action class itself lies. For example, if the action class is MyAction, the XML file would be named as MyAction-validation.xml. Observe the two different action mappings here:

```
<action name="myAlias" class="action.level.validation.MyAction">
.....
.....
</action>

<action name="myAnotherAlias" class="action.level.validation.MyAction"
method = "another">
.....
.....
</action>
```

In this example, both the actions, i.e. myAlias and myAnotherAlias, can be validated according to the same validation configuration file MyAction-validation.xml file.

Per Action Alias

We can also implement validation rules per action alias by creating a validation configuration file named ActionClassName-alias-validation.xml. This is known as Action Alias-level validation. To create an Action Alias-level validation, create a file ActionClassName-alias-validation.xml at the same location where the action class itself lies. For example, if the action class is MyAction with an alias myAlias, then the XML file would be named as MyAction-myAlias-validation.xml.

```
<action name="myAlias" class="action.level.validation.MyAction">
.....
.....
</action>
<action name="myAnotherAlias" class="action.level.validation.MyAction"
method = "another">
.....
.....
</action>
```

An Action Alias-level validation allows the validation to be applied to all MyAction action classes with alias myAlias only, hence limiting the scope. In the preceding example, the action myAlias will have the validation applied, whereas the action myAnotherAlias will not.

Short-Circuiting Validations

It is sometimes the case that when one validation fails then other validations are not required. For example, if an email field is required, and it is null, you do not need to check whether it is a valid email address. In order to enable this, XWork 1.0.1 added a short-circuit property to the Validation Framework. By using this validation, it is possible to short-circuit a stack of validators.

Here's Listing 9.19 showing how you can build a short-circuit validation:

Listing 9.19: Sample configuration for short-circuit validation

```

< validators >
< ! - - Field Validator Syntax for Emailvalidator - - >
    < field name = "EnterEmail" >
        < field-validator type = "requiredstring" short-circuit = "true" >
            < message > You must enter a value for Email Address
            < /message >
        < /field-validator >

        < field-validator type = "email" short-circuit = "true" >
            < message > Enter a valid Email Address < /message >
        < /field-validator >
    < /field >
< / validators >

```

Listing 9.19 describes that if the `Email` field is null or empty, the `Email` validator is not called because the short-circuit attribute is set to `true` for the required string validator.

NOTE

Validators should be written to be completely independent, so that a validator that checks the value of a property against some rule (for an example here we check that a string makes a valid email address) should not add an error message if the property is null. This is the job of Email validator.

The short-circuit option checks whether any error messages are added during the execution of that validator. For Field validators, it checks for errors added for this field.

Implementing Validation Annotations

The Validation annotations help in implementing validation rules on different fields without configuring them in some XML files. Different Validation annotations are provided here for corresponding validation rules. Let's see the available Validation annotations.

ConversionErrorFieldValidator Annotation

This annotation checks whether there are any conversion errors for a field and corrects them if there exists any. This validator must be applied at the method level. It has five parameters, such as `message`, `key`, `fieldName`, `shortCircuit` and `type`. Among these five parameters only the `message` and `type` are the required parameters, others are the optional. Here's an example of this annotation:

```
@ConversionErrorFieldValidator(message = "Default message",
                                key = "i18n.key", shortCircuit = true)
```

DateRangeFieldValidator Annotation

It checks the information regarding the date field, whether the date field has a value within a specified range. The `DateRangeFieldValidator` annotation has seven parameters:

- `message`
- `key`

- `fieldname`
- `shortCircuit`
- `type`
- `min`
- `max`

Among these parameters the `message` and `type` parameters are the necessary fields along with the `min` and `max` fields for the checking operations. All the other parameters are optional.

The use of this annotation is shown in the following code:

```
@DateRangeFieldValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true,
    min = "2005/01/01", max = "2005/12/31")
```

DoubleRangeFieldValidator Annotation

This validator checks whether a double field has a value within a specified range. So, here also, we have to provide `min` and `max` properties, otherwise it does nothing. The `DoubleRangeFieldValidator` annotation has the following parameters:

- `message`
- `key`
- `fieldName`
- `shortCircuit`
- `type`
- `minInclusive`
- `maxInclusive`
- `minExclusive`
- `maxExclusive`

The use of this annotation is shown in the following code:

```
@DoubleRangeFieldValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true,
    minInclusive = "0.123", maxInclusive = "99.987")
```

EmailValidator Annotation

This Validation annotation checks whether the field is containing a valid email address. It has the parameters similar to that of the `ConversionErrorFieldValidator` annotation.

The use of this annotation is shown in the following code:

```
@EmailValidator(message = "Default message",
    key = "i18n.key", shortCircuit = true)
```

ExpressionValidator Annotation

It validates an expression. This annotation must be applied at the method level. It has the following parameters:

- message
- key
- fieldName
- shortCircuit
- type
- expression

The use of this annotation is shown in the following code:

```
@ExpressionValidator(message = "Default message",
                      key = "i18n.key",
                      shortCircuit = true, expression = "an OGNL expression")
```

FieldExpressionValidator Annotation

This validator uses an OGNL expression to perform its validation. If the expression returns false at the time it is evaluated against the value stack, the error message will be added to the field. It has the same parameters like the ExpressionValidator:

- message
- key
- shortCircuit
- fieldName
- type
- expression

Among these parameters, only the message and type are the required fields. The others are optional. The use of this annotation is shown in the following code:

```
@FieldExpressionValidator(message = "Default message",
                           key = "i18n.key",
                           shortCircuit = true,
                           expression = "an OGNL expression" )
```

IntRangeFieldValidator Annotation

This annotation checks whether the numeric field has a value within a specified range or not. The IntRangeFieldValidator annotation has seven parameters:

- message
- key
- shortCircuit
- fieldName
- type

- min
- max

When you are using this annotation, you have to provide min and max values in such a way that 0 can also be considered as a possible value.

The following syntax shows how this annotation is used:

```
@IntRangeFieldValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true, min = "0", max = "42" )
```

RegexFieldValidator Annotation

It validates a string field using a regular expression. It is also applied in the method level. The RegexFieldValidator annotation has the following parameters:

- message
- key
- fieldName
- shortCircuit
- type

The use of this annotation is shown in the following code:

```
@RegexFieldValidator(key = "regex.field", expression = "your regexp")
```

RequiredFieldValidator Annotation

This annotation checks whether the field is not null. This must be applied in the method level. The RequiredFieldValidator annotation has the following parameters:

- message
- key
- fieldName
- shortCircuit
- type

The use of this annotation is shown in the following code:

```
@RequiredFieldValidator(message = "Default message",
    key = "i18n.key", shortCircuit = true)
```

RequiredStringValidator Annotation

It checks whether the string field is empty or not, i.e. it verifies whether the string's field length is greater than zero (0). The RequiredStringValidator annotation has the following parameters:

- message

- key
- fieldname
- shortCircuit
- type
- trim

Here, the trim property is a Boolean property, and determines whether the string is trimmed before performing the length check. The use of this annotation is shown in the following code:

```
@RequiredStringValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true, trim = true)
```

StringLengthFieldValidator Annotation

It checks the right length of the string field. Here you need to set the minLength and the maxLength; otherwise, you cannot utilize this annotation. The StringLengthFieldValidator annotation has the following parameters:

- message
- key
- fieldname
- shortCircuit
- type
- trim
- minLength
- maxLength

The use of this annotation is shown in the following code:

```
@StringLengthFieldValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true,
    trim = true,
    minLength = "5", maxLength = "12")
```

StringRegexValidator Annotation

This validator validates entered string against some configured regular expression. The StringRegexValidator annotation uses the following parameters:

- message
- key
- fieldname
- shortCircuit
- type
- regex

- `caseSensitive`

The parameter `caseSensitive` is to check whether the matching alpha characters in the expression should be checked keeping case-sensitivity in view or not. The use of this annotation is shown in the following code:

```
@StringRegexValidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true,
    regex = "a regular expression",
    caseSensitive = true)
```

UrlValidator Annotation

It checks for a valid URL. It must be applied at the method level. The `UrlValidator` annotation uses the following parameters:

- `message`
- `key`
- `fieldname`
- `shortCircuit`
- `type`

The use of this annotation is shown in the following code:

```
@UrlValidator (message = "Default message",
    key = "i18n.key", shortCircuit = true)
```

Validation Annotation

It is a Marker annotation for validation at Type level. Whenever you want to use annotation-based validation, you have to annotate the class or interface with `Validation` annotation. This `Validation` annotation must be used at Type level. The `Validation` annotation has the following parameters:

- `message`
- `type`
- `key`
- `fieldname`
- `shortCircuit`

Here's an example, given in Listing 9.20, that shows an annotated interface:

Listing 9.20: A sample annotated interface.

```
@Validation()
public interface AnnotationDataAware {

    void setBarObj(Bar b);

    Bar getBarobj();
```

```

@RequiredFieldValidator(message = "You must enter a value for data.")
@RequiredStringValidator(message = "You must enter a value for data.")
void setData(String data);

    String getData();
}

```

Here, you have to mark the interface with @Validation () as well as apply standard or custom annotations at the method level. Listing 9.21 shows a sample class using @Validation annotation:

Listing 9.21: A sample action class using @Validation()

```

@Validation()
public class SimpleAnnotationAction extends ActionSupport {

    @RequiredFieldValidator(type = ValidatorType.FIELD, message =
        "You must enter a value for bar.")
    @IntRangeFieldValidator(type = ValidatorType.FIELD, min = "6", max =
        "10", message = "bar must be between ${min} and ${max},
        current value is ${bar}.")
    public void setBar(int bar) {
        this.bar = bar;
    }

    public int getBar() {
        return bar;
    }

    public String execute() throws Exception {
        return SUCCESS;
    }
}

```

Validations Annotation

If you want to use several annotation of the same type, the annotations must be nested within the @validations () annotation. This is used at the method level. All the parameters, with their working functionality, are as follows:

- ❑ requiredFields – It adds the list of RequiredFieldValidators
- ❑ customValidator – It adds the list of CustomValidators
- ❑ conversionErrorFields – It adds the list of ConversionFieldErrorFieldValidators
- ❑ dateRangeFields – It adds the list of DateRangeFieldValidators
- ❑ Emails – It adds the list of EmailValidators
- ❑ fieldExpressions – It adds the list of FieldExpressionValidators
- ❑ intRangeFields – It adds list of IntRangeFieldValidators
- ❑ requiredStrings – It adds the list of RequiredStringValidators.
- ❑ stringLengthFields – It adds the list of StringLengthFieldValidators
- ❑ Urls – It adds the list of URLValidators

- ❑ visitorFields – It adds the list of VisitorFieldValidators
- ❑ stringRegex – It adds the list of StringRegexValidators
- ❑ regexFields – It adds the list of RegexFieldValidators
- ❑ expressions – It adds the list of ExpressionValidators

Here's an example, given in Listing 9.22, that shows the usage of Validations annotations:

Listing 9.22: A sample action class using @Validations()

```
@Validations(  
    requiredFields={  
        @RequiredFieldValidator(type = ValidatorType.SIMPLE,  
            fieldName = "customfield",  
            message = "You must enter a value for field.")  
    },  
    requiredStrings ={  
        @RequiredStringValidator(type = ValidatorType.SIMPLE,  
            fieldName = "stringisrequired",  
            message = "You must enter a value for string.")  
    },  
    emails = {  
        @EmailValidator(type = ValidatorType.SIMPLE,  
            fieldName = "emailaddress",  
            message = "You must enter a value for email.")  
    },  
)  
public String execute() throws Exception {  
    return SUCCESS;  
}
```

VisitorFieldValidator Annotation

This annotation allows you to forward the validator to object properties of your action using the objects' own validator files. This annotation lets you use the Model-Driven development pattern and handles your validation for your model. It can handle either simple Object properties, collection of properties or arrays.

The VisitorFieldValidator annotation has the following parameters:

- ❑ message
- ❑ key
- ❑ fieldname
- ❑ shortCircuit
- ❑ type
- ❑ context
- ❑ appendPrefix

Here the work of the appendPrefix is to determine whether the field name of this Field validator should be prepended to the field name of the visited field to determine the full field name when an error occurs. Suppose that the bean being validated has a name property. If this appendPrefix is to be true, then the field error will be stored under the field bean.name, otherwise the field error will be stored under the field name.

The use of this annotation is shown in the following code:

```
@visitorFieldvalidator(message = "Default message",
    key = "i18n.key",
    shortCircuit = true,
    context = "action alias", appendPrefix = true)
```

CustomValidator Annotation

This annotation can be used for Custom validators. You need to use the ValidationParameter annotation to provide additional parameters. The CustomValidator annotation has the following parameters:

- message
- type
- key
- fieldname
- shortCircuit

The use of this annotation is shown in the following code:

```
@Customvalidator(type = "customvalidatorName", fieldname = "myField")
```

After discussing about the various bundled validators provided by Struts 2 Validation Framework and describing different techniques to implement them on various action class fields, we can now move on to the “Immediate Solutions” section. This section discusses how validation is handled by using Struts 2 Validation Framework and the implementation of different validators by creating a Struts 2 based application implementing all support provided by the framework. We’ll also go through the implementation of annotations using two action classes

Immediate Solutions

Struts 2 comes with its own set of bundled validators, which can be used to validate a given data like checking an empty string, validating whether an integer is in a given range, checking an email id for its validity, etc. The Struts 2 Validation Framework provides an easy way to configure validation rules for different fields in different XML files. The Validation annotations have also been used in the application created here, which further eliminates the need of the XML file to configure validation rules to be applied on different fields.

The directory structure of the `Struts2Validation` application, created in this section and shown in Figure 9.1, is similar to other Struts 2 based applications. Create a directory structure similar to Figure 9.1 with project folder, `Struts2Validation`, created in `D:\Code\Chapter 9` folder on your machine.

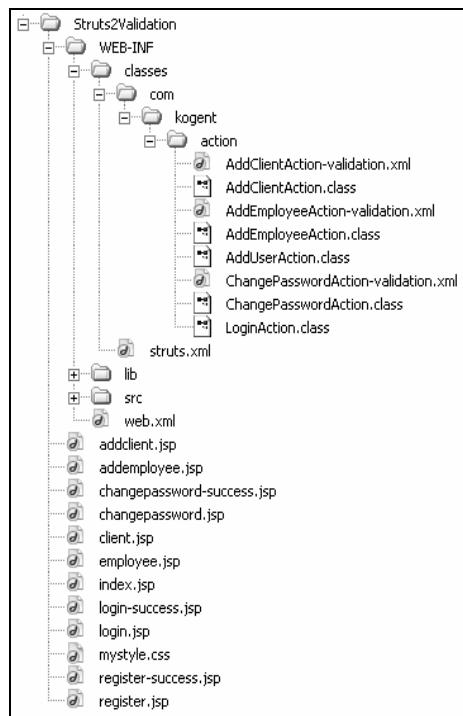


Figure 9.1: Directory structure of Struts2Validation application.

After creating various components (yet to be discussed) for the application, you need to deploy and run the application on your Web server as done earlier with other Struts 2 applications.

The first JSP page created here is `index.jsp`, which provides links to execute different examples implemented in this application.

Here's the code, given in Listing 9.23, for index.jsp file (you can find index.jsp file in Code\Chapter 9\Struts2Validation folder in CD):

Listing 9.23: index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Struts 2 Validation</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h2>Struts 2 Validation</h2>
<s:a href="addclient.jsp">A Basic Validation Example</s:a><br><br>
<s:a href="addemployee.jsp">Using Field Validators</s:a><br><br>
<s:a href="changepassword.jsp">Using Non-Field Validators</s:a><br><br>
<s:a href="login.jsp">Using Annotations for validation</s:a><br><br>
</body>
</html>
```

The Deployment Descriptor of the application is very much similar to other Struts 2 application, which provides a filter mapping and an optional welcome-file-list, also shown in Listing 9.24.

Here's the code, given in Listing 9.24, for creating your copy of web.xml with the configuration provided (you can find web.xml file in Code\Chapter 9\Struts2Validation\WEB-INF folder in CD):

Listing 9.24: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Struts 2 Validation</display-name>
        <filter>
            <filter-name>struts2</filter-name>
            <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
                </filter-class>
            </filter>
            <filter-mapping>
                <filter-name>struts2</filter-name>
                <url-pattern>/*</url-pattern>
            </filter-mapping>
            <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
            </welcome-file-list>
        </web-app>
```

Similarly, create your copy of Struts 2 configuration file, i.e. struts.xml files, with the basic structure shown in Listing 9.25.

Here's the code, given in Listing 9.25, where no action mapping is provided yet in struts.xml file (you can find struts.xml file in Code\Chapter 9\Struts2Validation\WEB-INF\classes folder in CD):

Listing 9.25: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="default" extends="struts-default">

</package>
</struts>
```

Copy different JARs in your WEB-INF/lib folder from the folder in CD. Now run your application to see the output of index.jsp page, as shown in Figure 9.2.

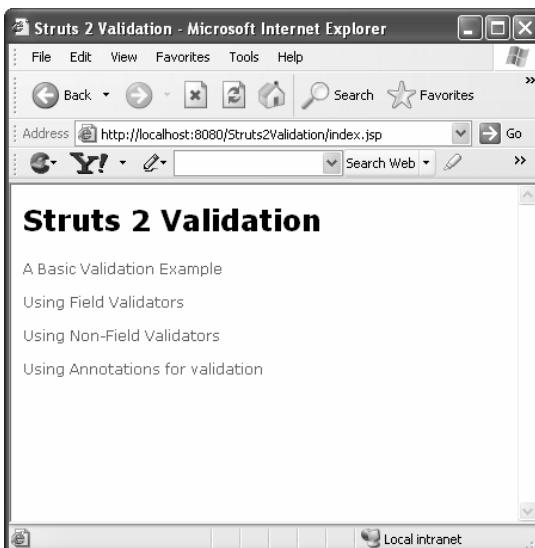


Figure 9.2: The index.jsp showing different hyperlinks

We can now implement different examples by creating a set of JSP pages, action classes and different XML files, and configuring different validators, which are to be applied on different fields of an action.

Creating Basic Validation Example

Here we are going to implement a small application using Struts 2 Validation Framework. In this example, we are not differentiating between Field and Non-field validators. We are creating a JSP page which prompts the user to enter some data, like 'Name', 'Age' and 'Nationality'. These fields will be validated for the data being entered. We just need to create a validation configuration file to validate the data being entered here.

Creating JSP Pages

The example uses two JSP pages here. The first JSP page is the `addclient.jsp` page, which provides form with input fields to be submitted for a new client. Here's the code, given in Listing 9.26, for `addclient.jsp` page (you can find `addclient.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.26: `addclient.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>A Basic Validation Example</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<h3>A Basic Validation Example</h3>
<b>Enter new client details:</b>
<s:form action="addClient">
    <s:textfield label="Name" name="name" required="true"/>
    <s:textfield label="Age" name="age" required="true"/>
    <s:textfield label="Nationality" name="nation" required="true"/>
    <s:submit value="Add Client"/>
</s:form>
    | <s:a href="index.jsp">Home</s:a> |
</html>
```

This JSP page provides a form with input fields to enter ‘Name’, ‘Age’, and ‘Nationality’ of the new client being added. The output screen for this JSP page is shown in Figure 9.3. Observe a ‘*’ displayed with all the field labels indicating that they are required. Just by putting a ‘*’ sign does not mean that the field will automatically validate; various things have to be implemented to validate these fields.

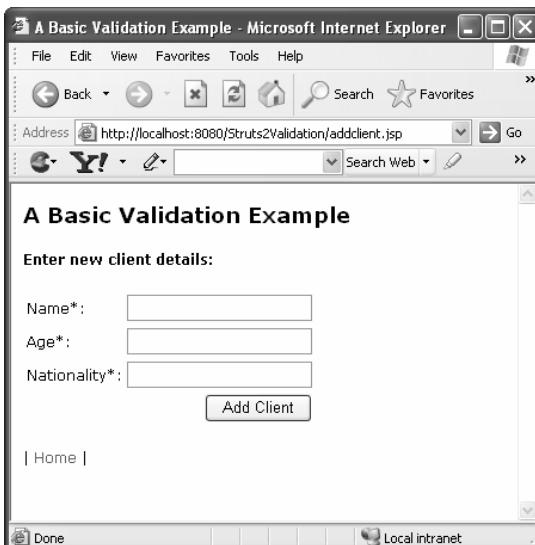


Figure 9.3: The `addclient.jsp` page.

Another JSP page used here is `client.jsp`, which simply displays the data being entered through `addclient.jsp` page. This page will be displayed only after the data being entered is validated and found valid.

Creating AddClientAction Class

The action class associated here is `AddClientAction`, which processes the data that is submitted from `addclient.jsp` page. The `AddClientAction` class provides three input properties matching three fields created in `addclient.jsp`. The class also provides getter and setter methods for these input properties in addition to its `execute()` method.

Here's the code, given in Listing 9.27, for `AddClientAction` class (you can find `AddClientAction.java` file in `Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action` folder in CD):

Listing 9.27: `AddClientAction.java`

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;

public class AddClientAction extends ActionSupport {
    String name;
    int age;
    String nation;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getNation() {
        return nation;
    }
    public void setNation(String nation) {
        this.nation = nation;
    }
    public String execute() throws Exception {

        //Logic for adding new client.
        return SUCCESS;
    }
}
```

Compile `AddClientAction.java` file and place the `.class` file created in `WEB-INF/classes/com/kogent/action` folder.

NOTE

We need the implementation of Validation interceptor to validate the action fields. The Validation interceptor works with ValidationAware actions only. So, make sure that your action class here implements the ValidationAware interface. You can extend ActionSupport class, which already implements the ValidationAware interface with some other interfaces for you.

Configuring **AddClientAction** Action

The form created in addclient.jsp page will request the URL <http://localhost:8080/Struts2Validation/addClient.action>. So, we need to provide an action mapping in struts.xml file. We have already created our struts.xml file. Now we just need to add the new action mapping in it.

Here the code, given in Listing 9.27, showing the new action mapping added:

Listing 9.27: struts.xml file with new action mapping.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="default" extends="struts-default">
<action name="addClient" class="com.kogent.action.AddClientAction">
    <result name="success">client.jsp</result>
    <result name="input">addclient.jsp</result>
</action>
</package>
</struts>
```

Note that we have not configured any interceptor here, but we are using the defaultStack as interceptor stack, which is the default for struts-default package defined in struts-default.xml file. We require the Validation interceptor, which is included in defaultStack interceptor stack to implement the Validation Framework. The Validation interceptor checks the action class against all validation rules defined in <ActionClassName>-validation.xml file and adds field-level and action-level error messages, given that the action class implements the com.opensymphony.xwork2.ValidationAware interface.

Creating **AddClientAction-validation.xml** File

For basic validation, we just need to define validation rules in <ActionClassName>-validation.xml file. The validation configuration file, created here for AddClientAction action class, is AddClientAction-validation.xml. Different validation rules to be implemented on different fields of AddClientAction action class can be defined in this validation configuration file.

Here's the validation configuration, given in Listing 9.28, done in AddClientAction-validation.xml file (you can find AddClientAction-validation.xml file in Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action folder in CD):

Listing 9.28: AddClientAction-validation.xml

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//xwork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="name">
        <field-validator type="requiredstring">
            <message>Name is required.</message>
        </field-validator>
    </field>
    <field name="age">
        <field-validator type="int">
            <param name="min">10</param>
            <param name="max">20</param>
            <message>Age must be between 10 and 20.</message>
        </field-validator>
    </field>
    <field name="nation">
        <field-validator type="requiredstring">
            <message>Nationality field is empty.</message>
        </field-validator>
    </field>
</validators>
```

The ‘name’ and ‘nation’ fields are being validated for empty string and ‘Age’ field is being validated for integer range. If the data in these fields is found invalid, the corresponding error messages, shown in Listing 9.28, are added as field errors. Now save AddClientAction-validation.xml file at the same location where the associated action class file is placed, i.e. WEB-INF/classes/com/kogent/action folder.

You can submit the form, created in Listing 9.26 and shown in Figure 9.3, with some invalid data to get the output similar to Figure 9.4.

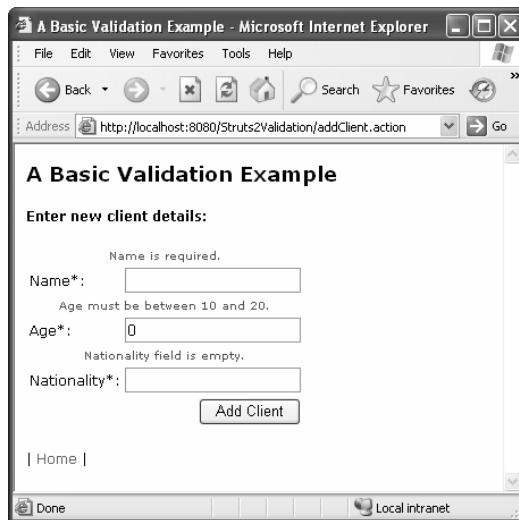


Figure 9.4: The addclient.jsp with validation errors.

Observe that the messages configured in `AddClientAction-validation.xml` file is being displayed here as field errors. The ‘Name’ and ‘Nationality’ fields are left blank and the value in ‘Age’ field is not in the specified range and this has resulted in the errors displayed shown in Figure 9.4. You can resubmit the form after adding valid set of data. The output of `client.jsp`, showing the submitted data for new client, is shown in Figure 9.5.

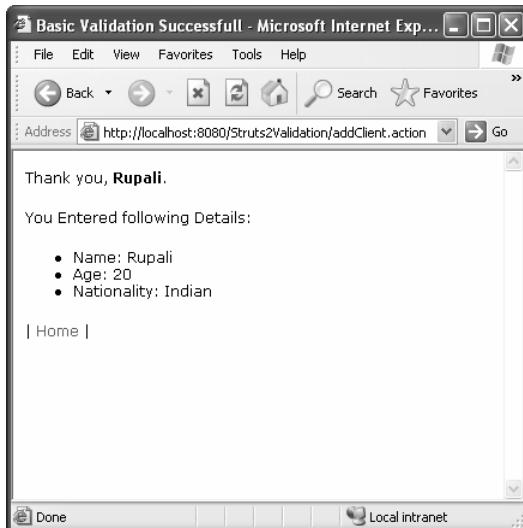


Figure 9.5: The client.jsp showing new client details added.

Using Field Validators

Field validators work on a single field of an action class. Most of the validations are defined on per field basis. We can define separate validation rules on separate fields of an action class. The Field validator messages are bound to the related field and will automatically be displayed besides the associated input field in the JSP page. This example uses two JSP pages, one action class and one validation configuration file associated with the action class to validate its different fields.

Creating JSP Pages

The first JSP page used in this example is `adddemployee.jsp`. This JSP page is similar to `addclient.jsp` page and provides a form with a number of input fields, like ‘Employee ID’, ‘Password’, ‘Employee Name’, ‘City’, ‘E-Mail’, etc

Here’s the code, given in Listing 9.29, for `adddemployee.jsp` page (you can find `adddemployee.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.29: `adddemployee.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
    <title>Using Field Validators-Adding New Employee</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
```

```
</head>
<body>
    <h3>Using Field Validators</h3>
    <b>Enter new employee details:</b>
    <s:form action="addEmployee">
        <s:textfield label="Employee ID" name="empid"/>
        <s:password Label="Password" name="password"/>
        <s:password label="Re-Enter Password" name="password1"/>
        <s:textfield label="Employee Name" name="empname"/>
        <s:textfield label="Date of Joining" name="doj"/>
        <s:textfield label="Age" name="age"/>
        <s:textfield label="City" name="city"/>
        <s:textfield label="E-Mail" name="email"/>
        <s:submit value="Add Employee"/>
    </s:form>
    <br><br>| <s:a href="index.jsp">Home</s:a> |
</body>
</html>
```

You can see different input fields created here in Listing 9.29. These input fields can be filled with the details of a new employee being added here. The form is to be submitted to some action mapped with name addEmployee, which is yet to be created and configured. All the input fields will be validated for the data being entered.

Another JSP page used in the example is employee.jsp page, which simply displays data submitted for the new employee to be added. The employee.jsp uses `<s:property/>` tag to display data from the action context.

Here's the code, given in Listing 9.30, for employee.jsp page (you can find employee.jsp file in Code\Chapter 9\Struts2Validation folder in CD):

Listing 9.30: employee.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Using Field Validators-Employee Added</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <h3>Employee added successfully</h3>
        <table cellspacing="5">
            <tr>
                <td>Employee ID: </td>
                <td><s:property value="empid"/></td></tr>
            <tr>
                <td>Password: </td>
                <td><s:property value="password"/></td></tr>
            <tr>
                <td>Employee Name: </td>
                <td><s:property value="empname"/></td></tr>
            <tr>
                <td>Date of Joining: </td>
                <td><s:property value="doj"/></td></tr>
            <tr>
```

```

        <td>Age: </td>
        <td><s:property value="age"/> years</td></tr>
    <tr>
        <td>City: </td>
        <td><s:property value="city"/></td></tr>
    <tr>
        <td>E-Mail: </td>
        <td><s:property value="email"/></td></tr>
    </table>
    <br><br>| <s:a href="index.jsp">Home</s:a> |
</body>
</html>
```

Now, we can create an action class to process the data being entered through addemployee.jsp page. We can create another validation configuration file to validate different fields of this action class too.

Creating AddEmployeeAction Class

The action class created here to implement the logic of adding a new employee is AddEmployeeAction. This is a simple action class created by extending the ActionSupport class. Using ActionSupport class here makes the AddEmployeeAction class ValidationAware also. This action class has input properties matching with the different input fields designed in addemployee.jsp page. The action class also defines a set of getter/setter methods for these input properties making them accessible. The execute() method would have implemented logic to add new employees in some database, but here we are emphasizing on validating data and the logic for adding employee has not been described here.

Here's the code, given in Listing 9.31, for AddEmployeeAction action class (you can find AddEmployeeAction.java file in Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action folder in CD):

Listing 9.31: AddEmployeeAction.java

```

package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
import java.sql.Date;
public class AddEmployeeAction extends ActionSupport{
    private String empid;
    private String password;
    private String password1;
    private String empname;
    private Date doj;
    private Integer age;
    private String city;
    private String email;

    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    public String getCity() {
        return city;
    }
```

```
    }
    public void setCity(String city) {
        this.city = city;
    }
    public Date getDoj() {
        return doj;
    }
    public void setDoj(Date doj) {
        this.doj = doj;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getEmpid() {
        return empid;
    }
    public void setEmpid(String empid) {
        this.empid = empid;
    }
    public String getEmpname() {
        return empname;
    }
    public void setEmpname(String empname) {
        this.empname = empname;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getPassword1() {
        return password1;
    }
    public void setPassword1(String password1) {
        this.password1 = password1;
    }
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

Now compile AddEmployeeAction.java and place the class file in WEB-INF/classes/com/kogent/action folder.

Configuring AddEmployeeAction Action

We need a new action mapping to be provided in struts.xml file for AddEmployeeAction action class. You can add the action mapped with name addEmployee in your copy of struts.xml file. Again, in this action mapping too, there is no interceptor configured and the defaultStack interceptor stack is being used by default.

Here's the new action mapping, given in Listing 9.32, that is added in struts.xml file along with the different results configured for this action:

Listing 9.32: struts.xml file with new action mapping

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="default" extends="struts-default">
<action name="addClient" class=". . . ">
    <result >. . . </result>
    <result >. . . </result>
</action>
<action name="addEmployee" class="com.kogent.action.AddEmployeeAction">
    <result name="success">employee.jsp</result>
    <result name="input">addemployee.jsp</result>
</action>
</package>
</struts>
```

Creating AddEmployeeAction-validation.xml File

The example is not ready to be executed, but the data is not going to be validated unless the Validator interceptor finds an <ActionClassName>-validation.xml file, where ActionClassName is the name of action class. Hence, to validate different fields of AddEmployeeAction action class, we are creating AddEmployeeAction-validation.xml file as validation configuration file here. The use of <field-validator> is preferred here to define different Field validators as they can inherit field name from the parent <field> element.

Here's Listing 9.33 showing the validation rules defined for different fields (you can find AddEmployeeAction-validation.xml file in Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action folder in CD):

Listing 9.33: AddEmployeeAction-validation.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//Xwork Validator 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>
    <field name="empid">
        <field-validator type="requiredstring">
            <message>Employee ID is required</message>
        </field-validator>
        <field-validator type="regex">
            <param name="expression">
                <![CDATA[([k][o][g][0-9][0-9][0-9])]]>
            </param>
            <message>Valid Employee ID required e.g. kog101</message>
        </field-validator>
    </field>
</validators>
```

```
</field-validator>
</field>

<field name="password">
    <field-validator type="requiredstring">
        <message>Password field is empty.</message>
    </field-validator>
    <field-validator type="stringlength">
        <param name="maxLength">10</param>
        <param name="minLength">4</param>
        <param name="trim">true</param>
        <message>Enter password 4-10 characters long.</message>
    </field-validator>
</field>

<field name="password1">
    <field-validator type="fieldexpression">
        <param name="expression">(password == password1)</param>
        <message>Password and Re-Enter Password must be same.</message>
    </field-validator>
</field>

<field name="empname">
    <field-validator type="requiredstring">
        <message>Employee Name is required.</message>
    </field-validator>
</field>

<field name="doj">
    <field-validator type="date">
        <param name="min">01/01/2006</param>
        <param name="max">01/01/2008</param>
        <message>Enter a date between ${min}and ${max}</message>
    </field-validator>
</field>

<field name="age">
    <field-validator type="int">
        <param name="min">18</param>
        <param name="max">45</param>
        <message>Enter between 18 and 45.</message>
    </field-validator>
</field>

<field name="email">
    <field-validator type="requiredstring">
        <message>E-Mail field is empty.</message>
    </field-validator>
    <field-validator type="email">
        <message>Enter a valid E-Mail Id.</message>
    </field-validator>
</field>
</validators>
```

Different validators used here are requiredstring, stringlength, regex, fieldexpression, date, int, and email. They all are discussed in detail in the “In Depth” section of this chapter.

NOTE

The simplest way to register all validators with ValidatorFactory is to add a file named validators.xml in WEB-INF/classes folder. However, if no Custom validator is being used, it is not required. If a Custom validator is being used and it is defined in validators.xml file, make sure that you have registered all bundled validators defined in com/opensymphony/xwork2/validator/validators/default.xml file in validators.xml file. Once the validators.xml file is detected in classpath, this default.xml file is not loaded and, hence, no bundled validator is registered.

Now, we ready to access the addemployee.jsp page by clicking over the ‘Using Field Validators’ hyperlink created in index.jsp (Listing 9.23) and shown in Figure 9.2. The output of addemployee.jsp page is shown in the Figure 9.6.

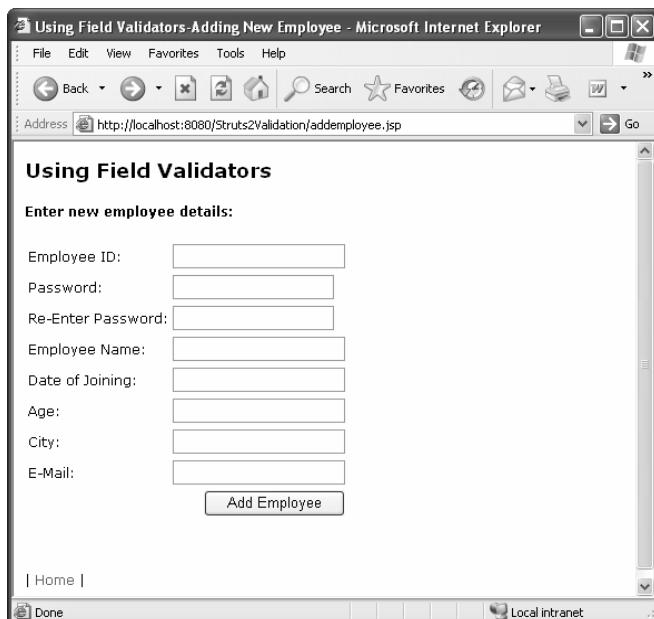


Figure 9.6: The addemployee.jsp page showing form with input fields.

You can click over ‘Add Employee’ button, shown in Figure 9.6, leaving all fields empty to see the number of field-level messages displayed in addemployee.jsp page. The output of addemployee.jsp page with different field errors is shown in Figure 9.7.

The screenshot shows the 'Using Field Validators-Adding New Employee' page in Microsoft Internet Explorer. The address bar shows the URL: <http://localhost:8080/Struts2Validation/addEmployee.action>. The page title is 'Using Field Validators'. The form is titled 'Enter new employee details:' and contains the following fields:

- Employee ID: [Text Box] - Error message: 'Employee ID is required.'
- Password: [Text Box] - Error message: 'Password field is empty.'
- Re-Enter Password: [Text Box]
- Employee Name: [Text Box]
- Date of Joining: [Text Box]
- Age: [Text Box]
- City: [Text Box]
- E-Mail: [Text Box] - Error message: 'E-Mail field is empty.'

At the bottom right is a 'Add Employee' button.

Figure 9.7: The addemployee.jsp page with some field errors.

We have applied more than one Field validators on few fields. For example, the `empid` field is validated with `requiredstring` and `regex` validator. If we enter some input to 'Employee ID' which does not match with the regular expression provided, then the error message displayed will be different this time. Similarly, some other fields, like 'Date of Joining', 'Age', 'Password' and 'E-mail' can be validated by different validators, if some data is provided (Figure 9.8).

The screenshot shows the same 'Using Field Validators-Adding New Employee' page in Microsoft Internet Explorer. The address bar shows the URL: <http://localhost:8080/Struts2Validation/addEmployee.action>. The page title is 'Using Field Validators'. The form is titled 'Enter new employee details:' and contains the following fields:

- Employee ID: [Text Box] - Error message: 'Valid Employee ID required e.g. kog101'
- Password: [Text Box] - Error message: 'Enter password 4-10 characters long.'
- Re-Enter Password: [Text Box] - Error message: 'Password and Re-Enter Password must be same.'
- Employee Name: [Text Box] - Value: 'Rupali'
- Date of Joining: [Text Box] - Error message: 'Enter a date between 1/1/06 and 1/1/08'
- Age: [Text Box] - Error message: 'Enter between 18 and 45.'
- City: [Text Box] - Value: 'Delhi'
- E-Mail: [Text Box] - Error message: 'Enter a valid E-Mail Id.'

At the bottom right is a 'Add Employee' button.

Figure 9.8: The addemployee.jsp page showing some other error messages.

If all the fields are filled with a valid set of data, the employee.jsp page is shown with the details submitted for the new employee being added. Figure 9.9 shows the output of employee.jsp.

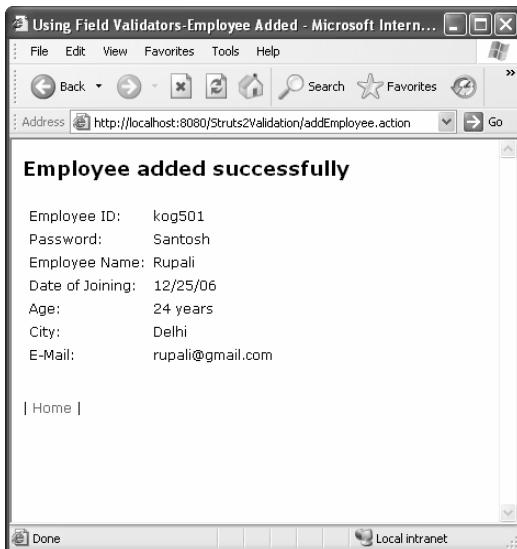


Figure 9.9: The employee.jsp page showing new employee details.

Using Non-Field Validators

The validators, which are not associated with a single field, can be defined as Non-field validators. A Field validator adds a field-level error message, while a non-field validator adds action-level messages. The most important Non-field validator is `ExpressionValidator`. Non-field validators always have higher priority over Field validators, irrespective of the order they are defined in validation configuration file. The example created here with the help of two JSPs, one action class, and one validation configuration file implements `ExpressionValidator`.

Creating JSP Pages

Let's now create an example that uses two JSP pages—`changepassword.jsp` and `changepassword-success.jsp`. The `changepassword.jsp` page appears like a form with three input fields. The page is being used as a console provided to change some user password. Out of these three input fields one is used to enter the current password and the rest of the two are used to enter new password. The new password entered in the two fields must be same. The condition stated in the last sentence is to be validated here before the associated action is executed.

Here's the code, given in Listing 9.34, for `changepassword.jsp` page (you can find `changepassword.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.34: `changepassword.jsp`

```
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Using Non-Field Validator-Changing Password</title>
```

```
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h3>Changing Password</h3>
    <s:actionerror/>
    <s:form action="changePassword" validate="true">
        <s:password name="password" label="Current Password" />
        <s:password name="password2" label="Enter New Password" />
        <s:password name="password3" label="Re-Enter New Password Again" />
        <s:submit label="Change Password"/>
    </s:form>
    <br>| <s:a href="index.jsp">Home</s:a> |
</body>
</html>
```

NOTE

You can observe the use of `validate="true"` with `<s:form/>` tag in Listing 9.34. This simply helps in implementing client-side validation without AJAX. There are some themes which do not support client-side validation. The setting of `validate` attribute of `<s:form/>` tag to `true` embeds the JavaScript in JSP page itself to validate different fields and in this way implements the client-side validation. You can see this JavaScript by viewing the source through your browser.

Another JSP page is `changepassword-success.jsp`, which intimates user for successful changing of password and displays the newly set password to the user.

Here's the code, given in Listing 9.35, for `changepassword-success.jsp` page (you can find `changepassword-success.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.35: `changepassword-success.jsp`

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Using Non-Field validators-Password Changed</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <h3>Password changed successfully</h3>
        You new Password is : <b><s:property value="password2"/></b>
        <br><br>| <s:a href="index.jsp">Home</s:a> |
    </body>
</html>
```

*Creating **ChangePasswordAction** Class*

The action class used here is `ChangePasswordAction`, which processes the data entered through `changepassword.jsp` page. The action class has three input properties `password`, `password2`, and `password3` with its getter/setter methods.

Here's the code, given in Listing 9.36, for `ChangePasswordAction` class (you can find `ChangePasswordAction.java` file in `Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action` folder in CD):

Listing 9.36: ChangePasswordAction.java

```

package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;

public class ChangePasswordAction extends ActionSupport{

    private String password;
    private String password2;
    private String password3;

    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getPassword2() {
        return password2;
    }
    public void setPassword2(String password2) {
        this.password2 = password2;
    }
    public String getPassword3() {
        return password3;
    }
    public void setPassword3(String password3) {
        this.password3 = password3;
    }
    public String execute() throws Exception {
        //Logic for changing Password.
        return SUCCESS;
    }
}

```

Compile ChangePasswordAction.java and place the class file in WEB-INF/classes/com/kogent/action folder.

*Configuring **ChangePasswordAction** Action*

The form created in changepassword.jsp page requires some action mapped in struts.xml file with the name changePassword. You can add the new action mapping with the name changePassword in your copy of struts.xml file.

Here's Listing 9.37 showing the new action mapping:

Listing 9.37: struts.xml file with new action mapping

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="default" extends="struts-default">

```

```
<action name="addClient" class=". . . . ">
    <result>. . . . </result>
    <result>. . . . </result>
</action>

<action name="addEmployee" class=". . . . ">
    <result>. . . . </result>
    <result>. . . . </result>
</action>

<action name="changePassword" class="com.kogent.action.ChangePasswordAction" >
    <result name="input">changepassword.jsp</result>
    <result>changepassword-success.jsp</result>
</action>

</package>
</struts>
```

Observe the action class and the two results configured for this action mapped in struts.xml file.

Creating ChangePasswordAction-validation.xml File

Similar to other examples created in the section, we are creating one more validation configuration file to the Define validator used here. There is no particular field associated with this validator. This validator takes an OGNL expression as the expression parameter. To validate the field this OGNL expression is evaluated, and if it fails an action level message is added by the validator. While entering a new password, the string entered twice as new password in different fields should be the same. This is checked here by the Expression validator. We have also used a Field validator here for password and password2 fields also.

Here's the code, given in Listing 9.38, that shows the Expression validator configured in ChangePasswordAction-validation.xml file (you can find ChangePasswordAction-validation.xml file in Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action folder in CD):

Listing 9.38: ChangePasswordAction-validation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//Xwork validator 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="password">
        <field-validator type="requiredstring">
            <message>Current password is required.</message>
        </field-validator>
    </field>

    <field name="password2">
        <field-validator type="requiredstring">
            <message>New password is required.</message>
        </field-validator>
    </field>
    <validator type="expression">
```

```

<param name="expression"><![CDATA[((password2 == password3))]]></param>
<message>
<![CDATA[ New Password and Re-Enter New Password field should be same.]]>
</message>
</validator>
</validators>

```

Now, click on the ‘Using Non-Field Validators’ hyperlink created in Listing 9.23 and shown in Figure 9.2 to display the output of changepassword.jsp, as shown in Figure 9.10.



Figure 9.10: The changepassword.jsp page showing form with three fields.

If you click on the ‘Change Password’ button (Figure 9.10), after filling all the fields with different strings, you’ll see an action level message added by Expression validator in the last two fields for the new password. This is shown in Figure 9.11.



Figure 9.11: The changepassword.jsp showing action level message added by Expression validator.

Performing Validation Using Annotation

Now we are going to develop a complete validation application using annotation-based validation. Use of annotation here eliminates the user of validation configuration files in the application. The Struts 2 provides a set of annotations to be used to specify different validators. We have directly defined validators for different fields in the action class itself using annotations. We have already discussed different Validation annotations in the “In Depth” section. This example here describes the implementation of annotations using two action classes.

This example interacts with a table in the database. Table 9.1 shows the structure of table, USER_TAB.

NOTE

Before running this example, make sure that you have access to an Oracle database sever and the table USER_TAB is created with some record inserted.

Table 9.1: Structure of USER_TAB table

Name	Type
USERNAME	VARCHAR2(30)
PASSWORD	VARCHAR2(30)
NAME	VARCHAR2(30)
AGE	NUMBER(10)
EMAIL	VARCHAR2(30)
NATIONALITY	VARCHAR2(30)
PHONE	NUMBER(15)

Creating JSP Pages

The example created here to implement validation using annotation uses different JSP pages—login.jsp, login-success.jsp, register.jsp, and register-success.jsp. The login.jsp gives a login form with two input fields for username and password. A user can login through this form by entering a valid set of username and password.

Here's the code, given in Listing 9.39, for login.jsp page (you can find login.jsp file in Code\Chapter 9\Struts2Validation folder in CD):

Listing 9.39: login.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
    <head>
        <title>Validation Using Annotation - Login</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <h3>Validation Using Annotation</h3>
```

```

<table border="1" bgcolor="#fcf3e2">
<tr><td align="center">Please Login</td></tr>
<tr><td><s:actionerror/></td></tr>
<tr><td>
<s:form action="login" validate="true" >
<s:textfield label="User Name" name="username"/>
<s:password label="Password" name="password"/>
<s:submit/>
</s:form>
</td></tr>
</table>
<br>
<b>Not Registered?</b>
Click here to <s:a href="register.jsp"><b>Register</b></s:a>
<br><br> | <s:a href="index.jsp">Home</s:a> |
</body>
</html>

```

Here, the action attribute of the `<s:form>` tag used is set to ‘login’ and requires an action to be mapped in `struts.xml` file with the same name. After a successful login the user gets a message displayed on the `login-success.jsp` page, a simple JSP page.

Here’s the code, given in Listing 9.40, for `login-success.jsp` (you can find `login-success.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.40: `login-success.jsp`

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>validation Using Annotation - Login Successful!</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h3>Login Successful!</h3>
    Thanks, <b><s:property value="username"/></b> .
    <br><br>
    <a href="register.jsp">Register as New User </a>
</body>
</html>

```

We can now register a new user in the database here. The `register.jsp` page again provides a form with a set of input fields to enter new user details. All the fields here again will be validated for the entered data. The form is submitted to an action mapped with the name `addUser`. The form includes fields for ‘Employee ID’, ‘Password’, ‘Employee name’, ‘E-mail’, ‘Age’, ‘Nationality’, etc.

Here’s the code, given in Listing 9.41, for `register.jsp` file (you can find `register.jsp` file in `Code\Chapter 9\Struts2Validation` folder in CD):

Listing 9.41: `register.jsp`

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>

```

```
<head>
<title>validation Using Annotation - Registering New User!</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<h3>Please Enter New User Details</h3>
<s:actionerror/>
<s:form action="addUser" validate="true" >
    <s:textfield label="Username" name="username"/>
    <s:password label="Password" name="password"/>
    <s:textfield label="Name" name="name"/>
    <s:textfield label="Age" name="age"/>
    <s:textfield label="Email" name="email"/>
    <s:textfield label="Nationality" name="nation"/>
    <s:textfield label="Phone" name="phone"/>
    <s:submit value="Add User"/>
</s:form>
| <s:a href="login.jsp">Login</s:a> | <s:a href="index.jsp">Home</s:a> |
</body>
</html>
```

After successfully adding a new user, the register-success.jsp appears displaying details of a new user added in the database. The page simply uses `<s:property/>` tag to show data.

Here's the code, given in Listing 9.42, for register-success.jsp page (you can find register-success.jsp file in Code\Chapter 9\Struts2Validation folder in CD):

Listing 9.42: register-success.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>validation Using Annotation - User Registered Successfully!</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h3>User successfully added!</h3><br>
    <b>The new user details are:</b>
    <br><br>User ID : <s:property value="username"/>
    <br><br>Password : <s:property value="password"/>
    <br><br>Name : <s:property value="name"/>
    <br><br>Age : <s:property value="age"/>
    <br><br>E-Mail ID : <s:property value="email"/>
    <br><br>Country : <s:property value="nation"/>
    <br><br>Phone : <s:property value="phone"/> <br><br>
    Click to <a href= "login.jsp">Login</a>
</body>
</html>
```

The forms created in login.jsp and register.jsp when submitted will look out for some actions configured in struts.xml. Let's create the required action classes and configure them in struts.xml file.

Creating Action Classes

The first action class created here is `LoginAction`, which processes the data entered through the form created in `login.jsp` page. The values entered for the fields of username and password are to be validated, which can easily be implemented using Validation annotations. We do not need to create any `<ActionClassName>-validation.xml` file to validate different fields now. Use of annotation to validate has eliminated the need for validation configuration file.

Here's the code, given in Listing 9.43, of `@RequiredFieldValidator`, which describes the full code of `LoginAction` action class (you can find `LoginAction.java` file in `Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action` folder in CD):

Listing 9.43: `LoginAction.java`

```

package com.kogent.action;

import com.opensymphony.xwork2.validator.annotations.*;
import com.opensymphony.xwork2.ActionSupport;
import java.sql.*;

public class LoginAction extends ActionSupport{
    private String username;
    private String password;

    @RequiredFieldValidator(type=validatorType.SIMPLE,
        fieldName = "username",
        message = "User Name field is empty.")

    public String getUsername(){
        return username;
    }
    public void setUsername(String username){
        this.username = username;
    }

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "password",
        message = "Password field is empty.")
    public String getPassword(){
        return password;
    }
    public void setPassword(String password){
        this.password = password;
    }

    public String execute()throws Exception {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        String url   = "jdbc:oracle:thin:@kogent8:1521:Ambrish";
        String userid = "scott";
        String password = "tiger";
        boolean flag=false;
        try{
            DriverManager.registerDriver(new

```

```
        oracle.jdbc.driver.oracleDriver());
conn=DriverManager.getConnection(url,userid,password);
stmt = conn.createStatement();
rs = stmt.executeQuery("select username, password from user_tab");
while(rs.next()){
    String usn=rs.getString(1);
    String pass = rs.getString(2);
    if((getUsername().equals(usn))&&(getPassword().equals(pass))){
        flag=true;
        break;
    }
}
//try close
catch (SQLException e ){
System.err.println(e.getMessage());
}
finally{
rs.close();
stmt.close();
conn.close();
}
if(flag){
    return SUCCESS;
}
else{
    this.addActionError("Invalid User Name or Password");
    return ERROR;
}
//execute close
}
//class close
```

NOTE

You can change the url=" jdbc:oracle:thin:@kogent8:1521:Ambrish" for different drivers, in case you are using different databases. The kogent8 is the Oracle server machine name and Ambrish is the database name here.

If the user with a given username and password exists, LoginAction action class returns ‘success’ as result code, otherwise it returns ‘error’ as result code after adding an action error.

Another action that we need to process data entered through register.jsp is AddUserAction. This action class is responsible for adding a new user record in the USER_TAB table in the database. In addition, all the fields are validated here by using the different Validation annotations provided by Struts 2. Different Validation annotations used in AddUserAction action class are @RequiredFieldValidator, @EmailValidator, and @IntRangeFieldValidator. The parameters, like type, fieldname, and message, for different annotations, can be set to customize the behavior of Validators being used.

Here’s the full code, given in Listing 9.44, for AddUserAction action class. (you can find AddUserAction.java file in Code\Chapter 9\Struts2Validation\WEB-INF\src\com\kogent\action folder in CD):

Listing 9.44: AddUserAction.java

```

package com.kogent.action;

import com.opensymphony.xwork2.validator.annotations.*;
import com.opensymphony.xwork2.ActionSupport;
import java.sql.*;

public class AddUserAction extends ActionSupport{
    private String username;
    private String password;
    private String name;
    private int age;
    private String nation;
    private String email;
    private long phone;

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "username",
        message = "User Name field is empty.")
    public String getUsername(){
        return username;
    }
    public void setUsername(String username){
        this.username = username;
    }

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "password",
        message = "Password field is empty.")
    public String getPassword(){
        return password;
    }
    public void setPassword(String password){
        this.password = password;
    }

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "name",
        message = "Name field is empty.")
    public String getName(){
        return name;
    }
    public void setName(string name){
        this.name = name;
    }

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "age",
        message = "Age field is empty.")
    @IntRangeFieldValidator(type = ValidatorType.FIELD,
        fieldName = "age",
        min="18", max="40",
        message = "Enter age between 18 and 40.")
    public int getAge(){

```

```
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }
    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "nation",
        message = "Nationality field is empty.")
    public void setNation(String nation){
        this.nation = nation;
    }
    public String getNation(){
        return nation;
    }
    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "email",
        message = "E-Mail is required.")
    @EmailValidator(type = ValidatorType.FIELD,
        fieldName = "email",
        message = "Enter a valid E-Mail ID.")

    public String getEmail(){
        return email;
    }
    public void setEmail(String email){
        this.email = email;
    }

    @RequiredFieldValidator(type = ValidatorType.SIMPLE,
        fieldName = "phone",
        message = "Please Enter your Phone Number")
    public long getPhone(){
        return phone;
    }
    public void setPhone(long phone){
        this.phone = phone;
    }
    public String execute()throws Exception{
        Connection conn = null;
        PreparedStatement ps = null;

        String url  = "jdbc:oracle:thin:@kogent8:1521:Ambrish";
        String user = "scott";
        String pwd = "tiger";
        int result=0;
        try
        {
            DriverManager.registerDriver(new
                oracle.jdbc.driver.OracleDriver());
            conn=DriverManager.getConnection(url,user,pwd);
            System.out.println("connection successful");

            ps =
            conn.prepareStatement("insert into user_tab
                values(?,?,?,?,?,?)");
            ps.setString(1,username);

```

```

        ps.setString(2,password);
        ps.setString(3,name);
        ps.setInt(4,age);
        ps.setString(5,email);
        ps.setString(6,nation);
        ps.setLong(7,phone);
        result=ps.executeUpdate();
        ps.close();
        conn.close();
    }catch ( SQLException e ){
        System.err.println(e.getMessage());
    }
    if(result == 1){
        return SUCCESS;
    }else{
        this.addActionError("Some exception occurred.");
        return ERROR;
    }
}
//execute close
}//class close

```

The AddUserAction action class can return error as result code in case an exception occurs during the execution of an action, otherwise it returns success. Compile LoginAction.java and AddUserAction.java and place the class files in WEB-INF/classes/com/kogent/action folder.

Configuring Actions in struts.xml

To make things functional, we need to provide action mapping for LoginAciton and AddUserAction classes. Add these action mappings in your copy of struts.xml file.

Here's the two action mappings, given in Listing 9.45, that are added in struts.xml file:

Listing 9.45: struts.xml file with new action mappings

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>

<package name="default" extends="struts-default">
<action name="addClient" class=". . . . ">
    <result >. . . . </result>
    <result >. . . . </result>
</action>
<action name="addEmployee" class=". . . . ">
    <result >. . . . </result>
    <result >. . . . </result>
</action>

<action name="changePassword" class=". . . . ">
    <result >. . . . </result>
    <result >. . . . </result>
</action>
<action name="login" class="com.kogent.action.LoginAction">

```

```
<result name="input">login.jsp</result>
<result name="error">login.jsp</result>
<result name="success">login-success.jsp</result>
</action>

<action name="adduser" class="com.kogent.action.AddUserAction">
    <result name="input">register.jsp</result>
    <result name="error">register.jsp</result>
    <result name="success">register-success.jsp</result>
</action>

</package>
</struts>
```

Observe the different results configured for the action mapped with the name, login and addUser. The defaultStack interceptor stack is the default interceptor stack working here. If you are configuring your own set of interceptors here, make sure that the Validator interceptor is included in the interceptor stack to enable validation through Validation Framework.

Let's run this final example created in *Struts2Validation* here, which implements validation using annotations without creating any validation configuration files. You can access the login.jsp by clicking on 'Using Annotations for validation' hyperlink created in Listing 9.23 for index.jsp and shown in Figure 9.2. This output of login.jsp page is shown in Figure 9.12.

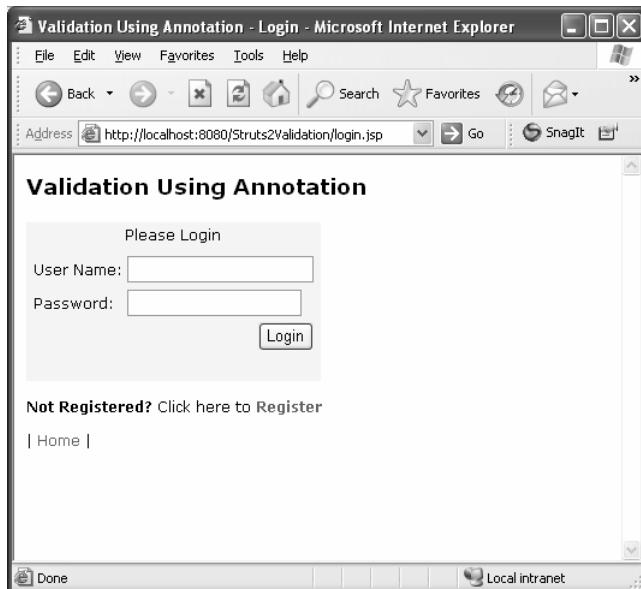


Figure 9.12: The login.jsp page showing login form.

The login form, created here using `<s:form>` tag, has its `validate` attribute set to `true` (see Listing 9.39). Hence, the JavaScript to validate these two fields are embedded here with the HTML for this page. Click on the 'Login' button leaving all fields blank to see the error messages, as shown in Figure 9.13. The error message is created here using JavaScript and this indicates that the action class is still not invoked. You can see the address bar in Figure 9.13, which is still showing `login.jsp`, instead of `login.action`.

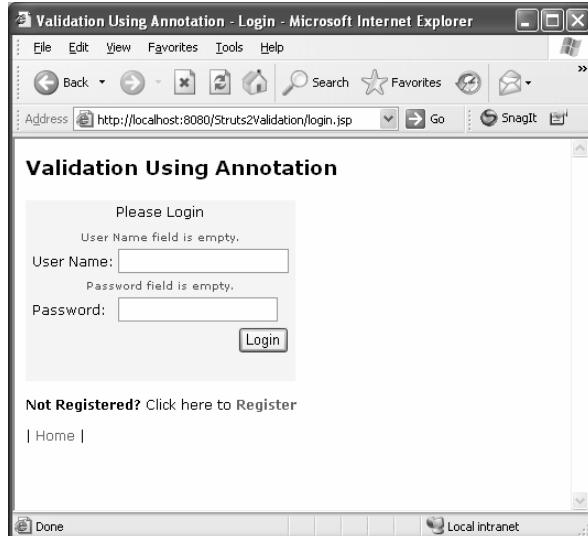


Figure 9.13: The login.jsp showing validation error messages.

You can enter the username and password of an existing user to see the output of login-success.jsp page. When entering username and password, make sure that you enter the associated record in the table USER_TAB. The output of login-success.jsp page is shown in Figure 9.14, which displays the username used to login.



Figure 9.14: The login-success.jsp page intimating successful login.

You can add a new user in the database by entering new user details in the form created in register.jsp page. You can access the register.jsp page by clicking on 'Register' hyperlink, shown in Figure 9.13, and 'Register as New User' hyperlink, shown in Figure 9.14. The new user registration form is shown in Figure 9.15, which is the output of register.jsp page.



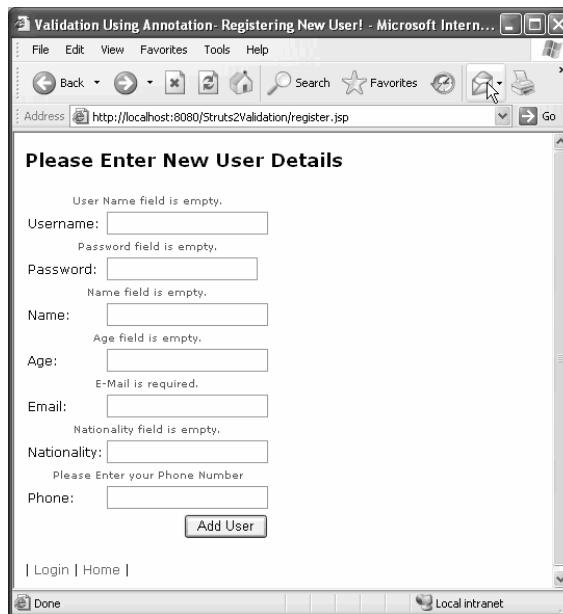
The screenshot shows a Microsoft Internet Explorer browser window with the title "Validation Using Annotation- Registering New User! - Microsoft Internet Explorer". The address bar shows the URL "http://localhost:8080/Struts2Validation/register.jsp". The main content area displays a form titled "Please Enter New User Details". The form contains the following fields:

- Username: [empty input field]
- Password: [empty input field]
- Name: [empty input field]
- Age: [empty input field]
- Email: [empty input field]
- Nationality: [empty input field]
- Phone: [empty input field]

Below the fields is a "Add User" button. At the bottom of the page, there are links for "Login" and "Home". The status bar at the bottom of the browser window shows "Done" and "Local intranet".

Figure 9.15: The register.jsp page showing new user registration form.

When the form is submitted by clicking over ‘Add User’ button, as shown in Figure 9.15, it uses AddUserAction as its action class. This action class uses different annotations to validate the data. We have again implemented client-side validation by setting the validate attribute of <s:form/> tag to true (see Listing 9.41). According to the different Validation annotations used in AddUserAction action class, the JavaScript is embedded with HTML generated for register.jsp page. Click on ‘Add User’ button leaving all fields blank to see the output similar to what is being shown in Figure 9.16.



The screenshot shows the same Microsoft Internet Explorer browser window as Figure 9.15, but with validation error messages displayed above each corresponding input field. The errors are:

- User Name field is empty.
- Password field is empty.
- Name field is empty.
- Age field is empty.
- E-Mail is required.
- Nationality field is empty.
- Please Enter your Phone Number

The rest of the form and layout are identical to Figure 9.15.

Figure 9.16: The register.jsp showing different validation error messages.

You can see different messages displayed in Figure 9.17 with the associated input components. All these messages are present because of the `@RequiredFieldValidator` annotation used in `AddUserAction` action class. We have used some more annotations too which can be seen, if the some invalid values in these fields are entered (Figure 9.17).

The screenshot shows a Microsoft Internet Explorer window with the title "Validation Using Annotation- Registering New User! - Microsoft Internet Explorer". The address bar shows the URL `http://localhost:8080/Struts2Validation/register.jsp`. The main content is a form titled "Please Enter New User Details". The form fields and their current values are:

- Username: santosh
- Password: *****
- Name: Santosh Kr. Jena
- Age: 45 (with validation message "Enter age between 18 and 40.")
- Email: santosh@g (with validation message "Enter a valid E-Mail ID.")
- Nationality: Indian
- Phone: 9873123351

At the bottom right of the form is a "Add User" button.

Figure 9.17: The register.jsp page showing some more validation error messages.

You can see messages displayed for 'Age' and 'Email' fields, which have been added here because of the use of `@IntRangeFieldValidator` and `@EmailValidator` annotations, respectively (See Listing 9.44). If all the fields are filled with valid data, then on the submission of the form it adds a new user in the database, and consequently the `register-success.jsp` is shown to the user with the new user details added. The output of `register-success.jsp` is shown in Figure 9.18.

The screenshot shows a Microsoft Internet Explorer window with the title "Validation Using Annotation- User Registered Successfully! - Microsoft Internet Explorer". The address bar shows the URL `http://localhost:8080/Struts2Validation/addUser.action`. The main content displays a success message and the new user details:

User successfully added!

The new user details are:

- User ID : santosh
- Password : rupalis
- Name : Santosh Kr. Jena
- Age : 26
- E-Mail ID : santosh@kogentindia.com
- Country : Indian
- Phone : 9873123351

At the bottom left is a "Click to Login" link.

Figure 9.18: The register-success.jsp page showing new user details.

So, we have implemented various ways of using Validation Framework provided by Struts 2. This chapter focused on the fact that the Validation Framework helps in validating user input in an easy and efficient manner, as it just needs the configuration of bundled Struts 2 validators in some validation configuration file. Even different input fields of an action class can be validated by providing the Validation annotations without creating any extra XML file for it. We have also gone through bundled validators in detail along with their functionality and implementations.

In the next chapter, we'll concentrate on Internationalization of Struts 2 based Web applications with all support available with Struts 2 APIs for localization of messages, and other locale specific formats.



10

Performing Internationalization in Struts 2

If you need an immediate solution to:

Developing the Struts18nApp Application

See page:

395

In Depth

Different regions of the world have their own culture. They have their own language, different from other languages. When two people from different language-speaking regions want to communicate with each other, they have to use a third language, which is known to both of them. Since English is an international language, it is mostly used as the third language for communication. The English language acts as a bridge between the people of linguistically different regions to share their views with one another. Due to this worldwide acceptance of English language, Web applications are generally written in English language. The use of English language in Web applications makes the applications understandable to people all over the world.

Although English language is supported by different regions of the world, there are many implications that raise question on the use of English language. Firstly, people feel more comfortable in their regional language. Therefore, people like the application to be represented in their own language. Moreover, for people who do not understand English, have no use of such an application. Language is the biggest constraint that limits the international scope of the application and keeps it away from the reach of other corners.

Secondly, besides language, there are other cultural dissimilarities that English language is unable to deal with. People in different countries may have their own different currencies, like Rupee in India, Dollar in USA, Yen in Japan, and so on. There is no common currency accepted worldwide, which we can use in our application for representing currency. Similarly, different regions on the globe have different time-zones. In, while it is morning in India, it will be night in America. So, we cannot rely on the time of our own region to develop an application.

Last, but not the least, the application should provide translation of other non-textual elements that are attached with the application, like pictures and audios, etc. Sometimes, the application may represent images that too are region-sensitive. In such a case, different images are presented in different regions for the same application. For example, an application may represent the Indian flag when accessed from India, the American flag when accessed from America, the British flag when accessed from Britain and so on. The application should replace the flag in different countries with their own national flag. In the same way, the application should provide support for the translation of other region-dependent elements, like audio and video files, that are attached within the application.

In order to make the Web application usable to everyone, all these implications must be resolved. This resolution is achieved by using Struts 2 support for Internationalization. Internationalization provides the translation of messages in the application in a user's own language. It also provides other culture-sensitive data according to the user's region, like time, money and number system representation, etc. Not only this, Internationalization also provides the conversion of various non-textual elements, embedded into the page, in a form that is relevant to the user. In this chapter, we'll study Internationalization and explore deeply the way Internationalization is implemented in Struts 2.

Understanding Internationalization

Struts 2 provides a feature that allows a Web application eligible for global usage. By utilizing these features the applications can be meant for international audiences, without caring for their national

dissimilarities and without affecting the functionalities provided by the application. Two terms—Internationalization and Localization—are used to describe these features of Struts. These terms have been widely used in Java. Internationalization is often abbreviated as `i18n`, because there are 18 letters between the first letter ‘T’ and the last one ‘n’. For the same reason, Localization is sometimes abbreviated as `l10n`. Read on to understand a detailed discussion on Internationalization and Localization features.

Internationalization is defined as the process of designing an application so that it can be adapted to various languages and regions without engineering changes.

In this definition of Internationalization two terms need to be paid due attention to—*adapt* and *engineering changes*. The first term *adapt* refers to the capability of the application to mould itself according to the user’s culture by representing all messages in user’s language. So, to internationalize the application, it should be adapted perfectly according to the user’s regional details.

The second term *engineering changes* refers to the changes in the code. The regional details are not hard-coded in the application. The control logic of the application remains the same and only the view is changed. This change in the view takes place for different locations and not for the same location. In one type of location, the same view is represented to all the users, but in different locations different views are presented.

The main theory behind the implementation of Internationalization feature involves using a key, wherever there is any region-specific data. This key is like an indicator that indicates that a region-sensitive data is used here. When that page is displayed to the user, the key is replaced by its value. The value of the key depends on the language of the user. Therefore there are different values of a key for different languages. All the possible values for this key are available in a file called *Resource Bundle* that is stored outside the source code.

When a session begins, the user’s browser provides local details of the user to the Web server. According to these local details of the user, a corresponding resource bundle is chosen. Wherever there is a key in the page, an appropriate value of the key is retrieved from the chosen resource bundle. This value is inserted in place of that key. The retrieval of key-value and its insertion is dynamic, i.e. it takes place during the run time of the application. This provides a Web application to the user in a format that the user is most likely aware of.

It is to be noted that the local details provided by the browser is stored in the server as an object in the `ActionContext`. This detail is saved temporarily for a user session only and not for the entire application. For different user sessions, these user-details are saved as different `ActionContext`. The objects in `ActionContext` are unique for every session. We’ll read more about `ActionContext` in the “`Locale`” sub-topic of the current topic.

To internationalize an application there are some areas that need to be addressed. These areas are as follows:

- ❑ **Input**—The format of the data coming from the user to the application depends on the browser settings of the user. The application should be able to understand that data and process it as usual. This involves the translation of data from user’s machine-dependent format to the application-dependent format. For example, a user may enter numeric values separated by commas, like 1,000,000, but the application is designed to accept the data in simple form without any commas, like 1000000. In such a scenario, the application must be capable enough to discard the commas in order to make the value interpretable by the application; otherwise, the commas will be also processed.
- ❑ **Output**—The output of the application seen by the user should be in the user’s language. Therefore, the application should be capable to translate the messages appearing in dialog boxes, prompts, button labels, etc. into the user’s language without altering the functionality of the application. Like

input, the messages in output should also be translated into user's format. Suppose that the user is sitting in 'India' and the application is running on a Web server located in 'California'. If the application includes displaying the current time, then it should display the current time of user's country, i.e. India and not the country where the server is located, i.e. California.

- ❑ **Validation** – The new language may use different formats for the representation of currency, date and time, etc. The application should be capable enough to validate the data in all the possible formats without generating any error. For example, the user in different regions may have different digit telephone numbers. The application should provide validation to all the types of phone numbers.
- ❑ **Database** – The data in the database is stored in a pre-defined format. When the user accesses the database, he/she may alter the data in another format, creating inconsistency in the database. To maintain consistency in the database, while transferring the data from one place to another, data should be stored in a standard format. The application should be capable of transforming the user data into a format supported by the database and vice versa. Generally, the access to database is provided by database-access logic.

As mentioned earlier, to avoid inconsistency in the database its format should not be altered. Moreover, the database is a crucial element of an application and should be protected from the outside world. Therefore, the database is not included in the Internationalization feature. Struts provides Internationalization by meeting the first three issues.

Localization

When we say that an application is internationalized, this means that the application is capable enough to adapt itself according to the user's language. The application contains all the formats for all the languages that it supports in the resource bundle. Now the question arises on how the application finds the user's regional-details and delivers the target user its intended data. The answer is that the application does not find the details itself; it is the responsibility of the user's machine to provide these details in its HTTP request. These details help the application to extract the application contents from the resource bundles. The application then creates a page that embeds messages adapted in the user's language, and then sends it back to the user. When the application is adapted accordingly, it is said to be localized in the user's language.

More precisely, Localization can be defined as the process of adapting software for a specific region or language by adding locale-specific components and translating text. Here, in the definition, the term to be focused is *locale-specific*, which emphasizes on those elements of the application that are dependent on the local settings of the user's machine. Locale is an object representing the regional settings of the user's machine.

NOTE

We'll see more about locales in the "Locale" sub-topic of the current topic.

The main purpose of Localization is to translate the region-specific information in the application into corresponding language and data format. This includes changing the language used along with changing currency, time and date, etc. into the local terms specified by user's machine.

At the first glance, the two terms, i.e. Internationalization and Localization, may seem to specify the same thing, but the two are different from each other. The relation between the two can be understood by looking at Internationalization as the developer-side effort and Localization as the user-side effort. Internationalization is like equipping the application with the abilities to be adapted itself according to user's environment. Localization is like molding the application in user's environment by utilizing the

adapting capability provided by Internationalization. To internationalize an application, it should be localized. In other words, not any application can be localized, only an internationalized application can be localized. The two terms complement each other and not the same thing.

ISO Language and Country Codes

Every language and country existing on this planet are assigned a unique code by ISO (International Organization for Standardization). ISO is a network of the national standard institutes of various countries that define a common standard for all the member nations. A technology related product, developed in a country, follows its own national standards, which may not be compatible to the standards of other nations. The member countries follow the specifications set by ISO, so that any product developed in one country can be used in other countries too. ISO has assigned a two-letter code to recognize all the languages and countries. These codes are specified under the ISO 639-1 (for language) and ISO-3166 (for countries) standard names. The language codes are in small alphabetic characters, whereas the country codes are in capital ones. Table 10.1 specifies the ISO language and country codes for some countries.

Table 10.1 Various ISO country and language codes

Country name	ISO 3166 country code	Language name	ISO 639-1 language code
Italy	IT	Italian	it
China	CN	Chinese	zh
France	FR	French	fr
Germany	DE	German	de
India	IN	Hindi	hi
Japan	JP	Japanese	ja
Spain	ES	Spanish	es
United States	US	English	en

These ISO country and language codes are used in those parameters of the HTTP request that specify the regional details of the user. The application reads these codes from the parameter and uses it to determine the right resource bundle.

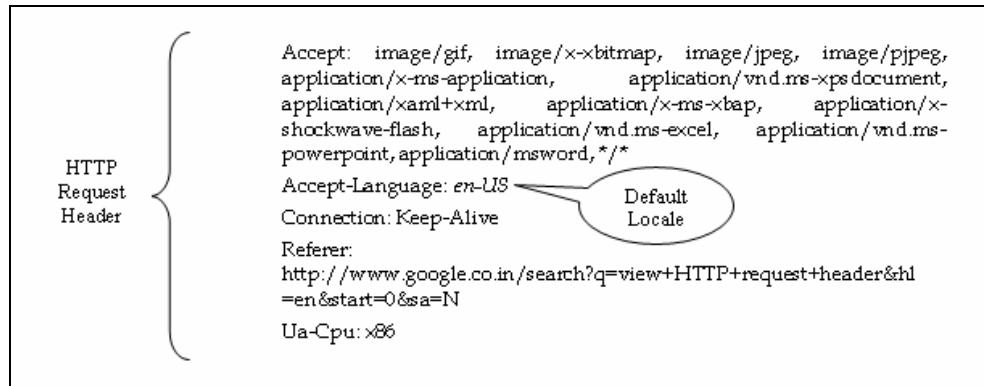


Figure 10.1: HTTP request header containing Default Locale

Figure 10.1 shows the HTTP request header from the author's machine, and the locale mentioned against the 'Accept Language' heading. Here the accepted language is mentioned as en-US that is read by the application to set the locale for the session.

Locale

To localize a Web application, it should be aware of the local variant of the user. Locale provides a way to do so. A locale is an identifier and represents the local details of the user to the application. As we saw in the previous topic, the web browser sends locale as a parameter in the HTTP header to the server. The locale uses ISO country and language codes to represent the user's country and language. If the application provides support for that locale, it responds to the request by sending all the pages in the respective language and data format. The elements in the application that are affected by the locale are called *locale-sensitive*.

The locale is a Java object that can be created by constructor provided by the `java.util.Locale` API. The constructor can be passed two arguments, one for the language and the second for the country.

For example:

```
Locale localeObject = new Locale ("hi", "IN");
```

In this example, a `Locale` object named `localeObject` is created and it is passed two arguments, `hi` and `IN`, to its constructor. Here, `hi` is the ISO language code for the Hindi language and `IN` is the ISO code for India. Thus, the `localeObject` represents a locale specifying the country as India and the language as Hindi. The object can also be created by passing one argument to the constructor. In that case the argument is taken as the language. The `Locale` object, thus created, will specify only the language and so can translate the messages only. Since no country code is provided, no specific regional details can be obtained. This is because one language can be spoken in different regions. The constructor does not support the country code as the only parameter, because there are multi-lingual countries. Therefore, the application needs to be mentioned along with the language for translating the messages.

When the user session begins, the `Locale` object is sent in the HTTP request as a parameter. This object is saved in the `ActionContext` of that session and remains there for the entire session. Therefore, there is no need to specify the user locale for every request in the session and only the first request for the session has to specify the locale. However, when a new locale is required in the running session, it must be specified in the new request. The new locale will replace the previous locale in the `ActionContext`.

Every session has its own ActionContext that contains the default locale for its respective user session.

The default locale of the user can be set in two ways. One way is to change the browser settings and another way is by the application itself. To change the default locale of the web browser, follow these steps:

- ❑ Click on the ‘Tools’ button on the toolbar of your web browser.
- ❑ In the drop-down menu that appears, click on ‘Internet Options’. A dialog box gets displayed.
- ❑ Now click on the ‘Languages’ button. This again displays a dialog box.
- ❑ Click on the ‘Add’ button to display a list of countries and their respective languages.
- ❑ Select the desired country and click ‘OK’. Once you click OK on the country, the country will be added to the Language preference list of the browser.

The new country selected by you is given the least preference in the list. The first preference is given to the previous language of the browser. To give your selected language the first preference, select that language in the preference list. Then click on the ‘Move up’ button. The selected language moves one step above in the list. If there are more than two languages, keep clicking on the ‘Move up’ button, until the selected language reaches at the top in the preference list. Figure 10.2 shows the way to change the locale using the web browser.

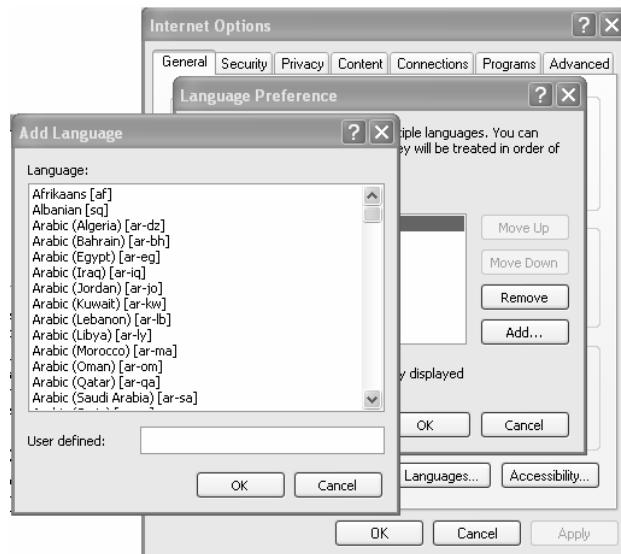


Figure 10.2: Changing the default locale using web browser.

Changing the locale by the web browser will permanently set the default locale to the selected one. To change the locale to a different one requires the user to follow the same steps again. The second method to change the locale is easier to use. In this method, the locale is not set permanently; it is set temporarily for the running session only. After the session is closed, the locale is set to the previous settings as provided by the web browser. In this method, the user may be asked to click a link on the browser that asks for a desired locale to select. For example, let’s consider a link displaying a locale as:

```
<a href="/anyAction.action?request_locale=fr">Click here for French language</a>
```

The preceding line in the page will display a message ‘Click here to select the French language’. When you click the message, the control will be passed to an action class.

When a session is closed, the original default value is again restored. In a new session, to change the default locale to the new locale, click on that link again. In Struts 2, this approach of changing locale dynamically is handled by `I18nInterceptor` interceptor class, which has been discussed in detail later in this chapter.

NOTE

We have already introduced `I18nInterceptor` in Chapter 4.

Resource Bundles

A resource bundle is a file containing the key-value pairs for a particular language. Different resource bundles are required for different languages. A resource bundle file is created in a text editor and saved with the `.properties` extension. Therefore, resource bundles are also known as property files. For example:

`ResourceBundleName.properties`

A property file for a language is distinguished from other property files by specifying a two-letter ISO language code in the file name. This code is appended with the file name before the file extension.

For example, let’s consider property files for different languages with entry for three keywords as follows:

- **ResourceBundle_fr.properties** – Properties file for French language:

```
app.greeting= Bonjour
app.username= usager nom
app.submit= s'assujettir
```

- **ResourceBundle_es.properties** – Properties file for Spanish language:

```
app.greeting=Hola
app.username= Nombre de Usuario
app.submit=somete
```

- **ResourceBundle.properties** – Properties file for default language:

```
app.greeting=welcome
app.username=Username
app.submit=SUBMIT
```

These entries tell Struts that when the user has a locale that uses the French language, and the key `app.greeting` is encountered in the code, then the value `Bonjour` will be substituted at that place, and when `app.username` is encountered, the `usager nom` will be substituted. Similarly, for the Spanish language Locale, `Hola` and `Nombre de Usuario` will be substituted in place of `app.greeting` and `app.username` keys, respectively. It is up to the developer to decide how many property files he wants to include in the application. A default property file is provided, for the users, whose locale is not supported by the application. In this example, the default property file contains values of the keys in

the English language. Therefore, Welcome and Username will be substituted for every occurrence of app.greeting and app.username keywords in the code, for the users other than French and Spanish. Once the property files are created, the next step is to deploy these files in the framework. In Struts 2, the deployment of property files is comparatively easier than Struts 1. Here the property files are stored at ApplicationName/WEB-INF/classes/ folder. To make Struts 2 aware of this file, a special file, called struts.properties, is used. This file contains the location of the properties file. For example:

```
struts.custom.i18n.resources=NameOfPropertyFile
```

This code line shows an entry in the struts.property file. The line describes the global resource for the application. If the property file is placed in the same folder as the struts.properties file, i.e. at ApplicationName/WEB-INF/class folder, then merely specifying the file name will do. However, if the file is placed at any different location, then the complete path to the folder must be specified.

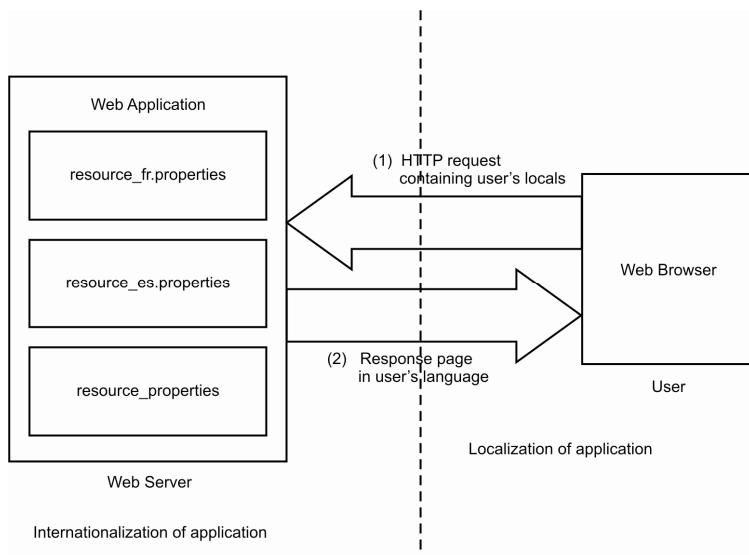


Figure 10.3: Internationalization using properties files.

Figure 10.3 shows the control flow involved in localizing an internationalized application. The process of Localization begins with the user's request. The web browser sends an HTTP request to the Web server. The HTTP request contains the locale of the user's machine containing the details of various local variant. The server reads the user's locale from the HTTP request and passes it to the Web application running in the container. The application receives the user's locale and uses it to determine the required properties file. All the locale-sensitive data is retrieved from the property file and substituted in the response page. In this way all the messages are delivered to the users in their local language, without any change in the functionality provided by the application.

Global Resource

We have seen earlier that to deploy a resource bundle in the framework, it is required to make the framework aware of it. Struts 2 allows these resource bundles to be associated with all the classes in the application. Whenever any locale-sensitive information is present, Struts 2 automatically looks for the resource bundles for that. These resource bundles are available to all the classes in the application. Since

resource bundles are global to all classes, these are also called as *Global Resource Bundles*. A global resource bundle is specified in the struts.properties file, which is available at ApplicationName/WEB-INF/classes folder. A global resource bundle can be specified in the struts.properties file in the following way:

```
struts.custom.i18n.resources=GlobalResourceBundle
```

It is also possible to specify more than one global resource bundles by separating the filenames using commas, as shown here:

```
struts.custom.i18n.resources= File_1, File_2
```

By mentioning more than one resource bundles, the action class will be associated with all the specified files and look for the required key-value in both the resource bundles.

Implementing Tags for Localization

The Internationalization features provided by Struts Framework not only reduces the complexity-level, but also provides advanced features. Localization supporting tags, provided by the framework, is responsible for providing Internationalization features to the application. We'll describe in detail the discussion on the mechanism of localizing an application and implementing the Localization supporting tag.

Struts 2 can internationalize an application at two places—at the UI tags and at the action/field error messages, along with validating the messages. Hence, we need to localize our application at these two places. Prior to Struts 1, there were a number of locale-sensitive tags provided by the frameworks to support Localization. However, Struts 2 provided only two types of tags that were required to localize the application. These tags are also known as message-aware tags.

Other tags that are not message-aware can use the `%{getText()}` expression. The internationalized elements can be made accessible to any part of the program with the help of the `getText()` method. The method fetches the internationalized message and places it at the point where the method was called. The complete format for the `getText()` method is as follows:

```
String getText (String SomeKey)
```

This method is used to get the value of `SomeKey` key in the resource bundle. If the value is found in the resource bundle, it will be returned as a String, otherwise a NULL will be returned as a default value.

Different tags available with Struts 2 which supports internationalization are `<s:text>`, `<s:i18n>`, `<s:property>` and more. These tags have been discussed here:

<s:text> Tag

This tag is used for rendering the locale-sensitive text-messages. These text-messages are stored in the resource bundle. These messages are fetched from the resource bundles and are placed at the point specified by the `<s:text>` tag.

The internationalized message is displayed, by mentioning its key-name against the name attribute provided by the `<s:text>` tag. For example:

```
<s:text name="SomeKey"/>
```

In this example, whenever the key, `SomeKey`, is encountered, its value will be retrieved from the associated resource bundle, and will be substituted in place of this tag. The tag, besides retrieving values for the keys, can also be made to display a default message whenever it is called in the application. For example:

```
<s:text name="SomeKey">
    Default Message to be displayed is placed here.
</s:text>
```

The message between `<s:text ...>` and `</s:text>` tags will be displayed along with the message corresponding to the key. It is also possible to assign a value to a key by passing the message as parameter to the resource bundle, as shown here:

```
<s:text name="SomeKey">
    <:param>Message</:param>
</s:text>
```

In this example, the value of the key, `SomeKey`, will be assigned `Message`, which is passed as a parameter. When the framework does not find any message corresponding to the key, it uses the body of the tag as default message. If no body is used then the key corresponding to the message will be used as default message.

<s:i18n> Tag

A resource bundle is associated with a particular action in the application. Sometimes, the action may require assistance from the resource bundles that are not associated to the action. To access a non-associated resource bundle, it should be placed on the value stack. With the use of `<s:i18n>` tag the resource bundle becomes visible and comes in the access scope of the running action.

The `<s:i18n>` tag gets a resource bundle and pushes it on the value stack. Any tag that falls between the `<s:i18n>` and `</s:i18n>` tags can access messages from that resource bundle. Hence, the bundles that are not associated with the current action class can also be made available to that action. For example:

```
<s:i18n name="someBundle">
    <s:property value="text('someKey')"/>
</s:i18n>
<s:i18n name="someBundle">
    <s:text name="someKey"/>
</s:i18n>
```

Here, the framework will search for the key, `someKey`, in the resource bundle `someBundle`. It is not necessary for the resource bundle to be associated with the current context. It will be associated with the current context automatically, whenever the `<s:i18n>` tag is used.

<s:property> Tag

It has been mentioned earlier that Struts 2 supports only two message-aware tags. Here we'll see how we can make other tags message-aware by using the `getText()` method.

Property tag is used to display the value of a property, say `SomeProperty`, which is set by the `setProperty()` method in the action class. This value of `SomeProperty` is retrieved by calling the `getProperty()` method in the action class. If no value is available, the top of the stack is referred as the default one. The property tag to work requires the property file of the concerned property to be placed on the value stack. The tag is generally used to print the current value of the Iterator. For example:

```
<s:iterator value="days">
    <p>day is: <s:property/></p>
</s:iterator>
```

This example retrieves the value of the `getDays()` method of the current object on the value stack. The `iterator` tag then retrieves that object from the `ActionContext` and finally calls the `getDays()` method. The property tag can be made to print the internationalized elements, if the properties file is on the value stack. Here's an example:

```
<s:property value="SomeProperty"/>
```

To display the value of an internationalized text, the same property tag can be used. The tag can be made to display the value corresponding to a key in the resource bundle by simply using the `getText()` method. For example:

```
<s:property value="getText('SomeKey')"/>
```

Here, the value for the `SomeKey` will be retrieved dynamically from the resource bundle and displayed where the tag is used.

<s:textfield> Tag

This tag is used to create a dynamic input field in a form. The input given by the user is saved as a value of the property that is already defined in the action class. For example:

```
<s:textfield name="SomeProperty" label="LabelName" />
```

If the property `SomeProperty` is available on the value stack, its value will be set to the value provided by the user in the input field created by the tag. The text field will be assigned a label `LabelName`, while displaying the text.

To display an internationalized message as the label, we can use the `#{getText()}` expression as explained earlier. For example:

```
<s:textfield name="SomeProperty" label="#{getText('SomeKey')}"/>
```

In this statement shown here, the `getText()` method is being invoked to search the value of `SomeKey` in the resource bundle and then assign it as the label. The framework provides a simpler way to display the internationalized message in the label without using the `"#{getText('SomeKey')}"` expression. This is done by simply mentioning the key-name against the `key` attribute provided by the `<s:textfield>` tag, as shown here:

```
<s:textfield name="SomeProperty" key="'SomeKey" />
```

Here, in this example, the value of the key `SomeKey` in the resource bundle will be assigned as a label to the input text field provided by the tag.

<s:submit> Tag

The `<s:submit>` tag is used to submit a form created by the `<s:form>` tag. The tag will show a button which when clicked, will submit the data entered in the input form fields. The message on the button can be displayed by the `value` attribute. For example:

```
<s:submit value= "SUBMIT" />
```

This example will display the ‘SUBMIT’ message on the submit button. This `submit` tag also supports the display of the Internationalization components in the application. Thus, the label on the submit button can display a message in the desired language. The message is internationalized by creating resource bundles for the desired languages. Here’s an example:

```
<s:submit value="#{getText('app.submit')}"/>
```

These internationalized messages can be displayed by placing their key names against the `key` attribute provided by the `<s:submit>` tag. For example:

```
<s:submit key="app.submit" />
```

The `submit` tag here will display the message for submitting the form in the language supported by the user’s locale. The value of the `app.submit` key will be retrieved from the associated resource bundle and will then be displayed as the message on the button. This tag only handles the functionality of the button and has no concern with the value of data submitted by the form.

<s:reset> Tag

This tag is used to reset the input field data in the form created by the `<s:form>` tag. It displays a button showing a message indicating it to be a reset button. When the user clicks the button, all the values entered in the input fields are reset to the default one. The tag also supports displaying Internationalization components in the application. The message on the button can be displayed by mentioning it against the `value` attribute. For example:

```
<s:reset value="RESET" />
```

This example will display ‘RESET’ message on the reset button. The message is internationalized by creating resource bundles for the desired languages. For example:

```
<s:reset value="#{getText('app.reset')}"/>
```

These internationalized messages can be displayed by placing their key names against the key attribute provided by the `<s:reset>` tag. Here’s an example:

```
<s:reset key="app.reset"/>
```

This `reset` tag will display the message for resetting the form in the language supported by the user’s locale. The value of the `app.reset` key will be retrieved from the associated resource bundle and be displayed as the message on the button.

Implementing I18n Interceptor

Interceptors are objects which dynamically intercept action invocations. These are used to allow the application developer to define codes that are required to be executed before or after the execution of an action. Interceptors can also be used to keep an action from getting executed.

When a request is received by the application, it is not sent directly for processing. Rather it has to pass through a series of interceptors (interceptor stack). Similarly, when the response is to be sent to the user, it is not sent directly but passed through the same interceptor.

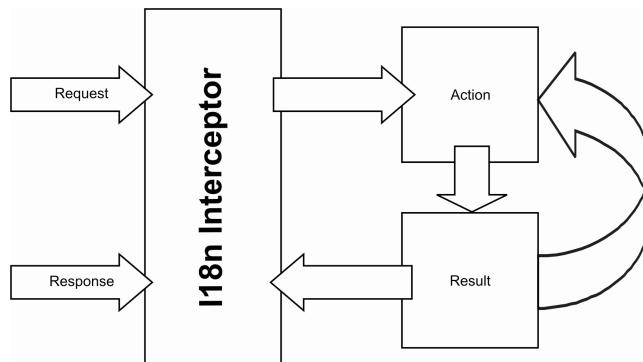


Figure10.4: Flow diagram showing processing of a Request.

Figure 10.4 show the control flow in processing an internationalized application. Here all requests are being intercepted by different interceptors including I18n interceptor. The I18n interceptor is used to remember the locale selected for a user’s session. With every HTTP request, a new thread is created, and the interceptor pushes the locale into the `ActionContext` associated to the thread. All the locale-sensitive methods utilize this locale.

The class associated with i18n inceptor is `com.opensymphony.xwork2.interceptor.I18nInterceptor`. This class is derived directly from `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class, which implements all methods of `com.opensymphony.xwork2.interceptor.Interceptor` interface, like `init()`, `destroy()`, and `intercept()`.

Whenever an HTTP request is received, the interceptor looks for the parameters in the request header and sets the locale according to the parameter. This locale remains the same for the entire session of the request. However, it is also possible to change the locale for a session dynamically, which allows the user to select a language of his/her choice at any point in the entire session. The locale parameter is removed from the HTTP request, while the interceptor is getting executed.

The interceptor can be passed two parameters—the `parameterName` and the `attributeName`. Both these parameters are optional:

- ❑ `parameterName`—This parameter tells the interceptor the name of the parameter in the HTTP request, which specifies the new locale. When no parameter is present, the default one is `request_locale`.
- ❑ `attributeName`—This parameter tells the interceptor the name of the session key to store the selected locale for the entire session. If no such parameter is present, the default one is `WW_TRANS_I18N_LOCALE`.

Let's suppose the current locale for a user's session is for the default language. A request by the browser `someAction?request_locale=en_EN` will cause the locale for the session to be set for the English language.

Here's an example of configuring i18n interceptor for the action:

```
<package name="some-default" extends="struts-default">
    <action name="someAction" class="examples.SomeAction">
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="basicStack"/>
        <result name="success">secondPage.jsp</result>
    </action>
</package>
```

The next example shows the setting of a parameter to customize i18n interceptor. Now, we can pass a request parameter named `newlocale`, which will be taken care of by the i18n interceptor to set the new locale according to the value of this parameter:

```
<package name="some-default" extends="struts-default">
    <action name="someAction" class="examples.SomeAction">

        <interceptor-ref name="i18n">
            <param name="parameterName">newlocale</param>
        </interceptor-ref >

        <interceptor-ref name="basicStack"/>
        <result name="success">secondPage.jsp</result>
    </action>
</package>
```

I18n Interceptor Methods

To work with the interceptors, the `i18nInterceptor` class provides the following methods:

- ❑ `intercept()`
- ❑ `saveLocale()`

- `init()`
- `destroy()`

Let's understand these methods in detail.

intercept() Method

The interceptor is executed by calling the `intercept()` method. The method is provided in the `Interceptor` interface. The argument passed to the method is an `ActionInvocation` interface that represents the execution state of an action. The method returns a code as String, which can be generated either from the `ActionInvocation.invoke()` method, or from the interceptor itself. The exception thrown can be any system-level error, as defined in the `ActionInvocation.invoke()` method. The derived class extending the `I18nInterceptor` class is required to override the `intercept()` method to handle interception. Here's the signature of `intercept()` method:

```
public String intercept (ActionInvocation invocation) throws Exception
{ }
```

saveLocale() Method:

Any class implementing the `ActionInvocation` interface is used to hold the Interceptors and the Action instance for a user session. When the locale is changed during a session, it is required to save the new locale to the object of the class implementing the `ActionInvocation`. This is done using the `saveLocale()` method:

```
protected void saveLocale(ActionInvocation invocation, Locale locale)
{ }
```

The method is not provided in the `Interceptor` interface, rather it is defined in the `I18nInterceptor` class. Two arguments are passed to the method. The first argument is an instance of the class implementing the `ActionInvocation`, and the second one is the `Locale` object that is to be saved to the `ActionInvocation`.

init() Method

The `init()` method is provided by the `com.opensymphony.xwork2.interceptor.Interceptor` interface and is inherited by the `I18nInterceptor` class from the `AbstractInterceptor` class, which implements the `Interceptor` interface. This method is called after an interceptor is created, but before any requests are processed by the `intercept()` method. This allows the interceptor to initialize the resources that are required during the interception.

destroy() Method

This method is called after the `intercept()` method. This allows the interceptor to clean up all the resources after the interception, which were allocated by the `init()` method before the interception. This method too is provided in the `Interceptor` interface.

It was mentioned in the beginning of the topic that the interceptor can be passed two parameters—`parameterName` and `attributeName`. These parameters are set by calling the following two methods provided of the `I18nInterceptor` interceptor:

□ `setParameterName()`

```
public void setParameterName(String parameterName){  
}
```

□ `setAttributeName()`

```
public void setAttributeName(String attributeName){  
}
```

In the “Immediate Solutions” section, we’ll discuss an application in order to understand the process of Internationalization of any Struts 2 application.

By now, in our journey to internationalize an application we have covered all the topics that are required to internationalize an application. At this point of time we are very well acquainted with the changes that are required to be made in the application in order to internationalize it. It is now the correct time to utilize the knowledge we have gained in this journey, by creating an internationalized application with a very simple functionality in the next section.

Immediate Solutions

Developing Strutsi18nApp Application

We are now going to create a simple application—the Strutsi18nApp. This application consists of two JSP pages, two actions, four properties files, and the configuration files required for any Struts 2 application. The view components comprise of the two JSPs, namely `index.jsp` and `success.jsp`. The two actions are `LoginAction` and `IndexAction`. The four properties files are `ResourceBundle_en.properties`, `ResourceBundle_fr.properties`, `resourceBundle_de.properties`, and `struts.properties`. The application does not do much of the processing, but simply explains the process of internationalizing an application using Struts 2. The user interface is a page, which has a login form along with the text which fills the rest of the portion of the page. The text is put on the page, only to explain the changes in the page on Internationalization. The user enters the username and password in the login form and submits it. If the username and password are correct, then we see the `success.jsp` page. In addition to this, the page has three hyperlinks for ‘German’, ‘English’, and ‘French’ languages. When the user clicks on any of the hyperlinks, the `index.jsp` page and the `success.jsp` page would show the contents in that locale only. The aim of this application is to explain the process behind this change.

We have divided the whole application into three main parts—“Usage of Struts 2 Tags”, “Preparing Resource Bundles”, and “Configuration Settings” of i18n interceptor along with the other required configurations. So let’s start with the usage of the Struts 2 tags in our application.

Using Internationalization Tags

We have already discussed the various Struts 2 tags, which support Internationalization. These tags are mainly used for developing the JSPs for the Web applications. In order to explain the importance of these tags in the Internationalization process, we’ll first give the code for the JSPs used in our application and then explain the various Struts 2 tags used in those pages. The very first page of our application is the `index.jsp` page. As mentioned here, this page has a text component, a login form, and three hyperlinks.

Here’s the code, given in Listing 10.1, for the `index.jsp` page (you can find `index.jsp` file in `Code\Chapter 10\Strutsi18nApp` folder in CD):

Listing 10.1: `index.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title><s:text name="app.title"/></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
```

```

<table align="center" width=600>
<tr><td colspan="3">
<h2 align="center"><s:text name="app.title"/></h2>
<div align="right"><s:text name="app.change"/> |
    <a href="index.action?request_locale=de">German</a> |
    <a href="index.action?request_locale=en">English</a> |
    <a href="index.action?request_locale=fr">French</a> |
</div>
<br><br>
<hr>
</td></tr>
<tr>
<td>

</td>
<td>

<table cellpadding="20">
<tr><td class="smalltext">
<s:text name="app.welcome.note"/>
</td></tr>
</table>

</td>
<td width="150">

<table bgcolor="#e8e7f8">
<tr><td colspan="2"><div class="boldred"><s:actionerror/></div></td></tr>
<tr><td align="center" colspan="2">Login!</td></tr>
<tr><td >
    <s:form action="login" method="post">
        <s:textfield name="username" key="app.username" size="15"
labelposition="top" />
        <s:password name="password" key="app.password" size="15"
labelposition="top"/>
        <s:submit key="app.submit"/>
    </s:form>
</td>
</tr>
</table>
</td>
</tr>
<tr><td colspan="3" align="center"><br><br>
    <hr>
    <s:text name="app.footer"/></td></tr>
</table>
</body>
</html>

```

In the Listing 10.1, if a request to `index.action?request_locale=de` is made, then the locale for the German language is saved in the user's session and will be used for all the future requests. Now, we'll explain the various Struts 2 specific tags, used in the `index.jsp` page one by one. Not all the tags used in this JSP page provide support for the Internationalization. We'll mainly explain only those tags which support Internationalization. The first tag to be explained here is the `<s:text>` tag. This tag is

first mentioned inside the `<title>` tag. The `<s:text>` tag has one required attribute name. This attribute gives the name of the resource property to be fetched from the properties file. In the `<s:text>` tag, the name has been assigned the value `app.title`. This is the resource property to be fetched from the properties file. The language of this text will depend on the option, which we choose by clicking the hyperlinks present on this page.

The next Struts Internationalization supporting tag used in Listing 10.1 is `<s:textfield>`. This tag has several attributes, but we have used only two of the attributes here namely name and key. The value assigned to the key attribute is `app.username` at the first instance of the tag. This value is used to fetch the property from the properties file. The other two tags used in `index.jsp` are `<s:submit>` and `<s:reset>` tags. The key attribute is used here and its value is used to fetch the property from the properties file. In addition to these tags, there are some other tags also which support Internationalization. These are `<s:property>`, `<s:i18n>` tags, and a method `getText()`. The details about `getText()` method will be provided to you when we would discuss about the Actions present in the application. But before that, we are giving you the code for the second JSP page present in our application.

Here's the code, given in Listing 10.2, for the `success.jsp` (you can find `success.jsp` file in `Code\Chapter 10\Strutsi18nApp` folder in CD):

Listing 10.2: success.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<div align="center">
<h2><s:text name="app.success_login"/></h2>
<a href="index.action"><s:text name="app.logout"/></a></div>
</body>
</html:html>
```

In Listing 10.2, we have used the tags, which we have already discussed for the `index.jsp` page. So we are not explaining them here again. You can find the details of these tags in the discussion for the `index.jsp` page.

Next, we'll discuss the actions present in our application. The first action to be mentioned here is the `LoginAction`. The `LoginAction` handles the processing of the login form present on the `index.jsp` page.

Here's the code for the `LoginAction.java` is given in the Listing 10.3 (you can find `LoginAction.java` file in `Code\Chapter 10\Strutsi18nApp\WEB-INF\src\com\kogent` folder in CD):

Listing 10.3: LoginAction.java

```
package com.kogent;
import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {
```

```
private String username;
private String password;

public String getUsername() {
    return username;
}
public void setUsername(String username) {
    this.username = username;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
public String execute() throws Exception {
    if(username.equals(password))
        return SUCCESS;
    else{
        this.addActionError(getText("app.invalid"));
        return ERROR;
    }
}
public void validate() {
    if ( (username == null ) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
    }
    if ( (password == null ) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
    }
}
}
```

The code for `LoginAction` given in the Listing 10.3 has getter and setters for the `username` and `password` fields of the `index.jsp` page. The `LoginAction` has two methods—`execute()` and `validate()`. The `execute()` method returns ‘`SUCCESS`’, if the `username` and `password` are same and returns error if they are not same. An error message is printed if there is an error. The following line of code fetches the error message from the properties file and gets it printed on the screen:

```
this.addActionError(getText("app.invalid"));
```

The method `getText("app.invalid")` needs some special attention. This method has been passed a parameter `app.invalid`. This parameter is a key, which is used to fetch the message from the properties file. This message gets printed at the place where this method is called. In our case, this method fetches an error message from the properties file and gets it printed at the required place. Similarly, `getText()` method is used in the `validate()` method for fetching the messages from the different properties files. The `validate()` method checks whether the user has filled the `username` and `password` fields present in the login form. It also checks whether any field doesn’t remain empty.

In addition to this action, our application has one more action, `IndexAction`. Here’s the code, given in the Listing 10.4, for `IndexAction.java` (you can find `IndexAction.java` file in `Code\Chapter 10\Struts18nApp\WEB-INF\src\com\kogent` folder in CD):

Listing 10.4: IndexAction.java

```

package com.kogent;
import com.opensymphony.xwork2.ActionSupport;
public class IndexAction extends ActionSupport{
    public String execute() throws Exception{
        return SUCCESS;
    }
}

```

The `IndexAction.java` has a very simple structure. This action doesn't do any processing, but its `execute()` method returns the message, 'SUCCESS'.

Next, we are going to explain you the Resource Bundles used in our application.

Preparing Resource Bundles

As mentioned earlier, the resource bundles are also called properties files. These files contain the key value pairs for different properties. A separate file is used for each language. The files are differentiated from each other by the two word ISO language code extension given at the end of each file. The values given in the properties files are fetched directly by the JSPs and Actions with the help of the keys. The application `Strutsi18nApp` contains three properties files one each for the 'English', 'French', and 'German' languages, respectively. One more file `struts.properties` is also present in our application. The properties files present in our application are as follows:

- ❑ `ResourceBundle_en.properties`
- ❑ `ResourceBundle_de.properties`
- ❑ `ResourceBundle_fr.properties`
- ❑ `struts.properties`

Here's the code, given in Listing 10.5, for `ResourceBundle_en.properties` (you can find these properties files in `Code\Chapter 10\Strutsi18nApp\WEB-INF\classes\com\kogent` folder in CD):

Listing 10.5: ResourceBundle_en.properties (properties file for English locale)

```

# Resources for parameter 'com.kogent.ResourceBundle'
# Project Strutsi18nApp
app.title=Struts 2 Internationalization
app.username=User Name
app.password=Password
app.success_login=Welcome! You have logged in successfully.
app.username.blank=User Name is Required
app.password.blank=Password is Required
app.submit=Login
app.logout=Logout
app.invalid=Invalid User Name or Password.
app.welcome.note=<p align="justify">Struts 2 provides a feature that allows web
application for global usage. By utilizing these features applications can be meant
for international audiences, without caring for there national dissimilarities and
without affecting the functionalities provided by the application.</p>
app.footer>All rights are reserved with ABC Pvt. Ltd.
app.change=Change Language

```

The properties file in Listing 10.5 shows the various key-value pairs for the English locale. The values will be fetched whenever the user selects the English locale. Here's the code, given in Listing 10.6, for ResourceBundle_de.properties (you can find these properties files in Code\Chapter 10\Strutsi18nApp\WEB-INF\classes\com\kogent folder in CD):

Listing 10.6: ResourceBundle_de.properties (properties file for German locale)

```
# Resources for parameter 'com.kogent.ResourceBundle'
# Project Strutsi18nApp
app.title=Struts 2 Internationalisierung
app.username=Benutzer Name
app.password=Kennwort
app.success_login=Willkommen! Du hast erfolgreich angemeldet.
app.username.blank=Benutzer-Name wird angefordert
app.password.blank=Kennwort wird angefordert
app.invalid=Unzulässiger Benutzer-Name oder Kennwort.
app.submit=Login
app.logout=Abmelden
app.welcome.note=<p align="justify">Struts 2 liefert eine Eigenschaft, die Netzanwendung für globalen Verbrauch erlaubt. Indem man verwendet, können diese Eigenschaften Anwendungen bedeuten werden für internationale Publikum, ohne für dort nationale Verschiedenartigkeit sich zu interessieren und ohne die Funktionalitäten zu beeinflussen, die von der Anwendung bereitgestellt werden.</p>
app.footer=Alle Rechte vorbehalten mit ABC Pvt. Ltd
app.change=Sprache ändern
```

The properties file given in the Listing 10.6 gives the same values for the German locale.

Here's the code, given in Listing 10.7, for ResourceBundle_fr.properties (you can find these properties files in Code\Chapter 10\Strutsi18nApp\WEB-INF\classes\com\kogent folder in CD):

Listing 10.7: ResourceBundle_fr.properties (properties file for the French locale)

```
# Resources for parameter 'com.kogent.ResourceBundle'
# Project Strutsi18nApp
app.title=Struts 2 Internationalisation
app.username=Nom d'utilisateur
app.password=Mot de passe
app.success_login=Bienvenue ! Vous avez entré avec succès.
app.username.blank=Le nom d'utilisateur est exigé
app.password.blank=Le mot de passe est exigé
app.invalid=Nom ou mot de passe inadmissible d'utilisateur
app.submit=Ouverture
app.logout=Déconnexion
app.welcome.note=<p align="justify">Les Struts 2 fournissent un dispositif qui permet la demande de web d'utilisation globale. Par l'utilisation ces applications de dispositifs peuvent être signifiées pour les assistances internationales, sans s'inquiéter des dissimilarités nationales et sans affecter les fonctionnalités fournies par l'application.</p>
app.footer=Tous les droits sont réservés avec ABC Pvt. Ltd
app.change=Choisir une autre langue
```

The properties file given in the Listing 10.7 shows the various properties for the French locale.

Here's the code, given in Listing 10.8, for struts.properties which has been configured for using com.kogent.ResourceBundle as the base name for resource bundles containing localized messages (you can find struts.properties file in Code\Chapter 10\Strutsi18nApp\WEB-INF\classes folder in CD):

Listing 10.8: struts.properties

```
struts.custom.i18n.resources=com.kogent.ResourceBundle
```

The property mentioned in the struts.properties file loads the custom resource bundle for our application.

Next, we would discuss the configuration required for the Strutsi18nApp application along with the configuration for the i18n interceptor.

Configuring **Strutsi18nApp** Application

The configuration settings for the Internationalization of an application mainly involves the configuration of the i18n interceptor. In our application, we have used the default interceptor, so that we need not configure it separately. But if you use the i18n tag in your application then in such a case you need to configure the i18n interceptor using the i18n interceptor stack in struts.xml file.

The configuration file for our application, i.e. struts.xml, configures two actions present in our application. Here's the code, given in Listing 10.9, for struts.xml (you can find struts.xml file in Code\Chapter 10\Strutsi18nApp\WEB-INF\classes folder in CD):

Listing 10.9: struts.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="login-default" extends="struts-default">
        <action name="login" class="com.kogent.LoginAction">
            <result name="success">/success.jsp</result>
            <result name="error">/index.jsp</result>
            <result name="input">/index.jsp</result>
        </action>
        <action name="index" class="com.kogent.IndexAction">
            <result name="success">/index.jsp</result>
        </action>
    </package>
</struts>
```

In the struts.xml file, shown in the Listing 10.9, first the LoginAction is being configured. There are three Results 'success', 'error', and 'input', respectively, for this action. The respective JSP pages will be shown to the user depending on the results. The second configuration involves the IndexAction. It has only one result, success, and the JSP page, which is shown to the user is index.jsp. The other configuration file is the usual web.xml file required for the deployment of the application.

Here's the code, given in Listing 10.10, for web.xml file (you can also find web.xml file in Code\Chapter 10\Strutsi18nApp\WEB-INF folder in CD):

Listing 10.10: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Struts 2 Internationalization</display-name>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

The web.xml file has been explained many times in the previous chapters and need not be discussed here.

In order to run this application, place the various resources according to the directory structure, discussed in previous chapters. After placing all the resources at their respective positions, start the Tomcat server and access the application using the URL <http://localhost:8080/StrutsI18nApp/>. The first page to be displayed is the index.jsp page and this is shown in Figure 10.5.

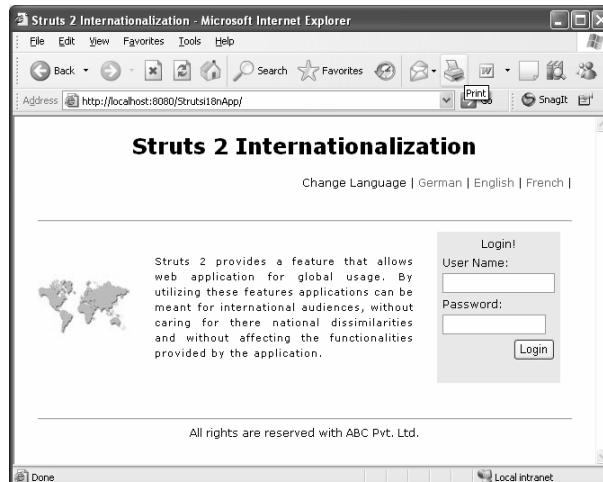


Figure10.5: The index.jsp with English (en) as default locale.

Figure 10.5 shows the default view of the `index.jsp` page. The contents of this page are shown using the 'English' locale. If the user clicks on some other locale present on the page, then the contents of the page will be shown in that locale. For example, if you click on 'German', the contents will look like the one shown in Figure 10.6.

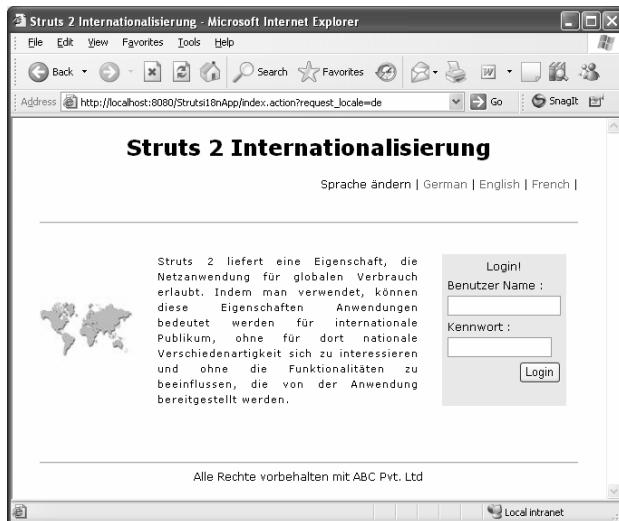


Figure 10.6: The index.jsp with German (de) as locale.

The contents shown in the Figure 10.6 are same as that of the contents shown in the Figure 10.5, but in a different language according to the new locale set by the `i18n` interceptor. Now, if you login successfully, the `success.jsp` page will be displayed on the screen. The locale set for the next page will remain the same, i.e. German(de).

The two different outputs of `success.jsp` page, supporting two different locales, are shown in Figures 10.7 and 10.8. Figure 10.7 shows the `success.jsp` page (in the 'English' locale).



Figure 10.7: The success.jsp with English locale.

The page, shown in the Figure 10.8, shows the same success .jsp page with ‘German’ locale.



Figure 10.8: The success.jsp page with German locale.

Thus, we have developed a very simple application that provides support for three different languages. Our example application only internationalizes the text messages. The application can be made to internationalize various other elements in the page, like images, music and videos, etc. The procedure to involve these non-text messages is the same as the text-messages. But it will require the usage of elements in the prescribed formats and with the associated tags. This completes our discussion on Internationalization of Struts 2 application.

In this chapter, we discussed Internationalization and Localization in detail. We also discussed the various Struts 2 tags that provide support for the Internationalization of the application. Next, we learnt about the `i18n` interceptor and Global Resource. Taking this thought process to the “Immediate Solutions” section, we developed an Internationalized application showing the usage of various Struts tags that provide support for Internationalization. We also learnt about the Resource Bundles and the configuration settings required for the Internationalization of Struts 2 application.

In the next chapter, we'll discuss about the Struts 2 Plugins, which helps in extending the features of Struts 2 along with its integration with other technologies.



11

Using Plugins in Struts 2

If you need an immediate solution to:

See page:

Implementing Codebehind Plugin	437
Implementing Struts 1 Plugin	442
Implementing JFreeChart Plugin	444
Implementing Config Browser Plugin	447

In Depth

The term *plugin* itself suggests that it is something, which can be plugged into the framework to introduce some extension points to the framework. This extension can be like adding some new functionality, new classes, result types, interceptors and some packages into the existing framework. The plugin approach separates different extension points creation, which can optionally be implemented when required. Different plugins also provide integration of Struts 2 Framework with other frameworks, like Spring, JSF, and other languages, like Groovy. Struts 2 provides various plugins which are provided with Struts 2 APIs. Hence, these plugins are also known as bundled plugins that are used to add extra functionality, add function on demand and add component to create your Web application more reliable. Open Application Programming Interfaces (APIs) provide a standard interface, allowing third parties to create plugins that interact with the main application. A stable API allows third-party plugins to function as the original version changes and to extend the life-cycle of obsolete applications. In this chapter, we'll discuss in detail each of the plugins bundled with Struts, and will also know how to configure these third party components into your Struts Web application.

Understanding Plugin

A *plugin* is a computer program which interacts with a main or host application, for example with a web browser or an email program, to provide a certain, usually very specific, function on demand. In Struts 2, a plugin is a simple JAR file, which is extended, replaced, or added to the existing Struts Framework in order to improve its functionality. A plugin can simply be installed by adding its JAR files into the applications' class path. The JAR file for a plugin may contain a `struts-plugin.xml` file providing configuration for the plugin. This `struts-plugin.xml` file follows the same pattern as that of the `struts.xml` file.

Since a plugin contains the `struts-plugin.xml` file, it has the ability to do the following tasks:

- It defines new packages with results, interceptors, and/or actions
- It overrides framework constants
- It provides new extension points implemented by different classes

The popular and optional features can be bundled and distributed as plugins. A particular application may retain all of the plugins provided with the framework, or may just include only those which are necessary for the implementation of the application. Plugins can also be used to organize application code or to distribute the code to third-parties.

All the plugins can be loaded in any order as there are no dependencies among different plugins. The plugins can use the Struts core classes and, hence, may depend on these classes, but they do not depend on the classes of other plugins.

In Struts 2, different configuration files are loaded in a particular order. The order in which the different configuration files are loaded is as follows:

- `struts-default.xml` – This file is bundled in the Core JAR.
- `struts-plugin.xml` – These files can be many in number. A single file is present in every JAR.

- ❑ `struts.xml` – This file is provided by the application.

The `struts.xml` file is loaded in the end to assure that it can use all the resources provided by other plugins added as JAR in the application.

There are several types of plugins available in the Struts 2 Framework. Some of the plugins are bundled with the distribution of the Struts 2 Framework, while some are available as third party plugins. We'll discuss both these types in detail.

Implementing Plugins in Struts 2

As discussed in the previous section, there are various plugins that come along with the distribution of the Struts 2 Framework. These plugins are also known as Struts 2 bundled plugin, because they come bundled with the framework distribution.

Table 11.1 shows the plugins that come bundled with the Struts 2 Framework.

Table 11.1 : Bundled plugins with description	
PLUGIN	DESCRIPTION
Codebehind	It reduces the required configuration for the action class and the results by adding the 'Page Controller' conventions
Config Browser	This plugin is a simple tool bundled with Struts 2 Framework, which helps in viewing the configuration details, including action mapping, exception mapping, result mappings, etc. at runtime
JasperReports	This plugin enables the actions to generate reports through JasperReports
JFreeChart	It helps the actions to return some charts and graphs based on some data
JSF	This plugin is used to provide support for Java Server Faces components and that is without any additional configuration
Pell Multipart	This plugin enables Struts 2 Framework to use the Jason Pell's multipart parser. This is used in order to process the file uploads
Plexus	The creation and injection of Actions, Interceptors, and Result by Plexus is enabled by this plugin. Plexus is another dependency injection framework like Spring
SiteGraph	This plugin creates a graphical representation of the flow in the Web application with diagrams showing actions, JSP, etc
SiteMesh	It is used to provide a common look and feel to the pages of the Web application using automatic wrapping of the simple page with the elements like headers and menu bars
Spring	The spring plugin allows the Actions, Interceptors, and

Table 11.1 : Bundled plugins with description	
PLUGIN	DESCRIPTION
	Results to be created and/or autowired by Spring
Struts 1	This plugin allows us to use Struts 1 Actions and ActionForms in the Web application, which is based on Struts 2 Framework
Tiles	It is an alternative to SiteMesh plugin. This gives a common look to the pages in the Web application by dividing pages into different fragments known as ‘tiles’

Let's now describe these plugins in detail.

The Codebehind Plugin

The Codebehind plugin is an experimental plugin bundled with Struts 2 Framework. This plugin can be used to provide some default implementation for action mapping and results configuration. The two common situations in which this plugin uses some conventions over configurations are as follows:

- ❑ **Default mappings**—Sometimes we do not require any action class for the execution of any logic before a page is shown to the user. One example of such a page is `index.jsp`, `index.ftl`, or `index.vm`. The pages using JSP tags and JSF components do not require any mappings.
- ❑ **Default Results**—Most of the action classes execute and prepare some data to be presented on a specific page. The list of different results is defined for an action. The Codebehind plugin helps in removing this result declaration for an action with some naming convention to be followed.

The Codebehind plugin encourages a page-based development, which links actions with pages and different pages for different results for the action. So, the Codebehind plugin provides the following two features:

- ❑ It provides the default mappings for pages that do not need any action to be executed.
- ❑ It provides default results by automatically searching the appropriate pages.

Using Codebehind Plugin

To use this plugin, we need to copy `struts2-codebehind-plugin-2.0.6.jar` file into the classpath. This plugin is used to find default mappings and default results. Read on to understand more about them.

Default Mappings

This plugin provides code-behind development approach by detecting the condition when the request has not been defined in any of Struts action mapping, but the corresponding page exists. In this case, the Codebehind plugin creates a dummy action mapping, referencing the default action class, which is usually the `ActionSupport` class. This will allow the page to be displayed normally. Additionally, the default interceptor stack for the configured package will be applied; this will bring the workflow benefits of interceptor stacks to the simple pages.

If there is no explicitly configured action for a given request, the plugin searches a likely page and the pattern used here is `/namespacename/action.jsp|action.vm|action.ftl`. This means, if there is no action mapping provided in `struts.xml` file with the name="login" and the request is

`http://localhost:8080/projectname/namespace/login.action`, then the plugin looks for the pages in the following order.

- /namespace/login.jsp
- /namespace/login.vm
- /namespace/login.ftl

If any of the aforementioned pages are found, then the plugin will create an `ActionConfig` object on the fly. The plugin will use the `Actionsupport` class for the Action and a single Result, which will point to the discovered page. The `ActionConfig` will be put in the configured package, meaning that it will be inheriting the default Interceptor stack for that package. The default package for the `Codebehind` package is `codebehind-default`; however, it can be configured in any configuration file via the `struts.codebehind.defaultPackage` constant.

Default Results

The `Codebehind` plugin tries to guess appropriate results for the action, if there is no result configured explicitly. This works for any result code returned from the action class. When zero configuration style is combined with this approach, the configuration details next to nothing. If no result is found configured for the result code retuned by the action, the plugin searches for the matching page and the following patterns are used:

```
/namespace/action-resultcode.(jsp|vm|ftl)
/namespace/action.(jsp|vm|ftl)
```

The patterns mentioned here are searched for all the default extensions, i.e. `.jsp`, `.vm`, `.ftl`. For example, for a request `http://localhost:8080/projectname/namespace/login.action` and the result code ‘success’, the plugin will search different pages in the following order:

```
/namespace/login-success.jsp
/namespace/login.jsp
/namespace/login-success.vm
/namespace/login.vm
/namespace/login-success.ftl
/namespace/login.ftl
```

The `struts-plugin.xml` file bundled with the JAR for this plugin file is shown here:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<bean type="com.opensymphony.xwork2.UnknownHandler"
class="org.apache.struts2.codebehind.CodebehindUnknownHandler" />
<constant name="struts.codebehind.pathPrefix" value="/" />
<constant name="struts.codebehind.defaultPackage" value="codebehind-default" />
<constant name="struts.mapper.alwaysSelectFullNamespace" value="true" />
<package name="codebehind-default" extends="struts-default">
</package>
</struts>
```

The Config Browser Plugin

The Config Browser plugin is a simple tool, which helps in viewing an application's configuration at runtime. This plugin is very useful when debugging problems that could be related to the configuration issues. The features of Config Browser plugin are as follows:

- It provides the browsable view of the loaded configuration.
- It also shows all the accessible action URLs.

In order to use this plugin, you just have to copy the JAR file into your Web application. Once it is installed, you can access this plugin by opening to the action named index in the Config-browser namespace. This plugin doesn't provide any customizable setting.

The configuration settings for Config Browser plugin are provided in the struts-plugin.xml file bundled in its JAR, i.e. struts2-config-browser-plugin-2.0.6.jar, which is as follows:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>

<package name="config-browser" extends="struts-default" namespace=
"/config-browser">
<interceptors>
    <interceptor-stack name="config-browser-default">
        <interceptor-ref name="validationWorkflowStack"/>
    </interceptor-stack>
</interceptors>
<default-interceptor-ref name="config-browser-default"/>

<global-results>
    <result name="error" type="freemarker">
        /config-browser/error.ftl
    </result>
    <result name="input" type="freemarker">
        /config-browser/error.ftl
    </result>
</global-results>

<action name="index">
    <result type="redirect-action">actionNames</result>
</action>

<action name="actionNames"
       class="org.apache.struts2.config_browser.ActionNamesAction">
    <result type="freemarker" name="success">
        /config-browser/actionNames.ftl
    </result>
</action>

<action name="showConfig"
       class="org.apache.struts2.config_browser.ShowConfigAction">
    <result type="freemarker" name="success">
        /config-browser/showConfig.ftl
    </result>
</action>

```

```
</result>
</action>

<action name="showValidators"
       class="org.apache.struts2.config_browser.ListValidatorsAction">
    <result name="error" type="freemarker">
        /config-browser/simple-error.ftl
    </result>
    <result name="input" type="freemarker">
        /config-browser/simple-error.ftl
    </result>
    <result type="freemarker" name="success">
        /config-browser/showvalidators.ftl
    </result>
</action>

<action name="validatorDetails"
       class="org.apache.struts2.config_browser.ShowValidatorAction">
    <result type="freemarker" name="success">
        /config-browser/validatorDetails.ftl
    </result>
</action>
</package>
</struts>
```

The JasperReports Plugin

The JasperReports plugin enables the actions to create high quality reports as results. Before studying about the JasperReports plugin, we'll first give you a brief introduction to JasperReports. JasperReports is a powerful open source reporting tool, which has the ability to deliver rich content to the screen, printer, or into PDF (Portable Document Format), HTML (Hyper Text Markup Language), XLS, CSV (Comma Separated Values) and XML (Extensible Markup Language) files. JasperReports are written entirely in Java and can be used in a variety of Java-enabled applications for generating dynamic content. The main objective of JasperReports is to generate page-oriented, ready-to-print documents in a simple and flexible way.

JasperReports organizes the data retrieved from the relational database using JDBC. This data is organized according to the design defined in an XML file. The report-design is first compiled and then the data is entered into the reports in order to fill the reports with the requisite data.

The compilation of the XML file, which represents the report-design, is performed using the `compileReport()` method of the `net.sf.jasperreports.engine.JasperManager` class. After compilation, the report design is loaded into a report-design object, which is then serialized and is stored on the disk (`net.sf.jasperreports.engine.JasperReport`). This serialized object is used to fill the specific report-design with the data as and when the application needs to do so. When the report-design is compiled, the Java code present in the XML file is also compiled along with the XML code. Various verifications are made at the compilation time for checking the report-design consistency. The result is a ready-to-fill report-design, which will generate documents on various sets of the data.

The `fillReportXXX()` method of the `net.sf.jasperreports.engine.JasperManager` class is used to fill a report-design. The report-design object or the file representing the specified report-design object is received as parameters for the `fillReportXXX()` method in a serialized form. The report is filled with data accessed from the database using JDBC connection.

The result is an object, which will represent a ready-to-print document (`net.sf.jasperreports.engine.JasperPrint`). This object can be stored on a disk in a serialized form for later use and can be delivered to the screen, or can be transformed into a PDF, HTML, CSV, or XML document.

The various features of JasperReports are as follows:

- ❑ It is embeddable, i.e. it helps to embed Java reporting library, and use the following features:
 - Enables embedding in any host application
 - No external reporting server required
 - Plug in Java and Groovy code
- ❑ It helps to handle complex reports with the following features:
 - Sub-reports easily handle highly complex layouts
 - Pixel-perfect page-oriented output for web or print
 - Report output in PDF, XML, HTML, CSV, XLS, RTF, TXT
- ❑ It helps in integrated charting with this feature:
 - Comprehensive set of chart types
- ❑ It supports Internationalization and Localization with the following features:
 - Multi-language Unicode and other native encodings
 - Dynamic text localization
 - Localized date, number, and currency formatting
- ❑ Due to scalability it helps to generate reports, which give excellent performance, with no limit to report size.
- ❑ Its extensible environment provides support for Groovy-based expressions.
- ❑ It is easy to use as it has :
 - iReport visual report designer
 - Other Eclipse- and Swing-based designers available
 - Built-in Swing viewer
 - Extensive code examples
- ❑ It provides reports written in Java and their definition in XML
- ❑ Its Flexible Data Access helps in the following:
 - Integrating multiple data sources of multiple kinds in one report
 - Developing application reports with Built-in support for JDBC, EJB, POJO, Hibernate, and XML

As mentioned earlier, the JasperReports plugin enables Actions to create high-quality reports in the form of results. This plugin allows the Actions to be rendered through JasperReports. In order to use this plugin, the packages that contain the target actions should extend the provided `jasperreports-default` package. This default package contains the Jasper result type. You can then use the result type in the desired actions. Table 11.2 shows the JasperReports plugin parameters.

Table 11.2 : JasperReports plugin parameters	
Parameter	Description
location	This is the default parameter. It provides the location where the compiled jasper report is present. This location is relative from the current URL
dataSource	This is the required parameter. The EL expression is used to retrieve the datasource from the value stack
parse	This parameter is true by default. If it is set to false, then the location parameter will not be parsed for expression language expressions
format	This parameter provides the format in which the report should be generated. Valid values can be found in JasperReportConstants. If there is no specified format, then the PDF format will be used
contentDisposition	It sets disposition which defaults to inline. The values are typically filename="document.pdf"
documentName	This parameter provides the name of the document
delimiter	This parameter is used when generating the CSV reports; by default, the character used is
imageServletUrl	This parameter provides the name of the url that, when prefixed with the context page, can return the images of the report

This follows the same set of rules from `StrutsResultSupport`. Specifically, all the parameters will be parsed if the `parse` parameter is not set to `false`. The JasperReports plugin can be used by just copying the requisite JAR file into `WEB-INF/lib` folder of your Web application.

The JasperReports plugin is used for displaying the results to the user in a predefined format. This is also visible in the configuration settings for this plugin, where only the `<result-types>` tags are configured for providing support for the JasperReports plugin. The configuration settings for JasperReports plugin is as follows:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<package name="jasperreports-default">
    <result-types>
        <result-type name="jasper"
        class="org.apache.struts2.views.jasperreports.JasperReportsResult"/>
    </result-types>
</package>
</struts>

```

The JFreeChart Plugin

The JFreeChart is a free Java chart library, which makes it easy for developers to display professional quality charts in their applications. JFreeChart have the following features:

- ❑ Consistent and well-documented API, supporting a wide range of chart types.
- ❑ Flexible design that is easy to extend, and targets both server-side and client-side applications.
- ❑ Support for many output types, including Swing components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS, and SVG).
- ❑ JFreeChart is an *open source* or, more specifically, a free software. It is distributed under the terms of the GNU Lesser General Public License (LGPL), which permits its use in proprietary applications.

The JFreeChart plugin allows the Actions to easily return the generated charts and graphs. Instead of streaming a generated chart directly to the HTTP response, the JFreeChart plugin provides a ChartResult, which handles the request. This feature allows you to generate the chart in one class and render it out in another class by effectively decoupling the view from the Actions. You can easily render it out to a file or some view other than a web HTTP response if you wish. The JFreeChart plugin has the following features:

- ❑ This plugin handles rendering charts to the HTTP response.
- ❑ This plugin can be used in other non-web contexts.

Currently the `chart` property is hardcoded. There should be a better way for transforming the data from Action to the Result, via some externally defined variable.

This plugin can be used by copying the requisite JAR file in the `lib` directory of the `WEB-INF` folder of your application. The JFreeChart plugin doesn't provide any global settings. The configuration settings for JFreeChart plugin, which is provided in `struts-plugin.xml` file bundled with its JAR, is shown here for you. This plugin is used for integrating the standard quality charts into your Web application. In this case also only the `<result-types>` tag is needed to be configured:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="jfreechart-default">
        <result-types>
            <result-type name="chart"
                class="org.apache.struts2.dispatcher.ChartResult">
                <param name="height">150</param>
                <param name="width">200</param>
            </result-type>
        </result-types>
    </package>
</struts>

```

The JSF Plugin

The addition of JSF plugin JAR files helps in providing support for the JavaServer Faces components and it is without any additional configurations. But before discussing the JSF plugin, we'll first give you a brief introduction to the JavaServer Faces technology.

JavaServer Faces technology includes a set of APIs, which represents the UI components and manage their state. They also handle the events and input validation, define the page navigation, and also support Internationalization and accessibility. In addition, a JSP custom tag library is there to express a JSP interface within some JSP page.

JavaServer Faces is a flexible technology, i.e. it is very easy to develop applications using JSF. JSF enhances the standard UI and web-tier concepts. This technology allows the user to use any particular markup language, protocol, or device. The UI component which are included with JSF, encapsulate only the functionality of the components, not the client-specific presentation, and thus enables the JSF UI components to be rendered by various client devices. Developers can construct custom tags to a particular client device. This can be done by combining the UI component functionality with custom renderers. These renderers define the rendering attributes for the specific UI component. JavaServer Faces technology provides accustom renderer and a JSP custom tag library for rendering an HTML client, which allows the Java Enterprise Edition (Java EE) developers to use JSF technology in their applications.

The primary goal of the JSF technology is the ease-of-use. The architecture clearly defines a separation between the application logic and the presentation logic. This clear separation between both the logics makes it easy to connect the presentation layer to the application code. This form of design also helps the members of the development team to focus on his or her piece of work. It also provides a simple programming model for linking the pieces together. For example, web page developers with no programming expertise can use the JavaServer Faces UI component tags to link to the application code from within a web page without writing any scripts.

JavaServer Faces technology establishes the standard for developing the server-side user interfaces. The various expert groups have well contributed in designing the JavaServer Faces APIs. These APIs are leveraged by the tools, which makes developing a Web application easier.

Now we can discuss our JSP plugin, which provides supports for JSF components. The result easily helps in incorporating component-driven pages, as required. The JSF life-cycle classes are broken into Struts Interceptors. The inclusion of `jsfStack` interceptor stack ensures that the all the phases of JSF page are executed correctly. Struts action is executed after the completion of all JSF life-cycle phases. The JSF `render` phase is transformed into a Result.

A concept of action is also supported by JSF. The JSF actions are handled as usual using `jsfStack` stack. The String result code is treated as Struts result code and they are not applied against the JSF navigation rule. This assures that the navigation is controlled by Struts.

Other JSF functionalities, like PhaseListeners, components, multiple backing beans, are preserved. The features of JSF plugin are as follows:

- This plugin allows the JSF components to be present on the normal Struts pages.
- JSF plugin doesn't require any additional configuration.
- It allows customization of JSF life-cycle.
- Most of the JSF Framework features are preserved.

Using JSF Plugin

The JSF support works by breaking up the JSF life cycle class into Struts Interceptors, one for each JSF phase. When you include the `jsfStack` stack, you are ensuring that the JSF page has its phases executed correctly. The Struts action is executed at the end of all phases. The JSF `render` phase has been transformed into a Result. The things required to use a page with JSF components are as follows:

- Configure `jsfStack` interceptor stack for the action, which can be implemented by extending package `jsf-default`.

- Set result types as jsf.

Other interceptors and results, which are not using JSF, can also be included. Regular Struts results are suggested to be used to handle any navigation to avoid common JSF application problems. The additional advantage added here is that every page has an action to execute page setup code and the same action instance is available in the JSF page's expression language as action.

The common page logics, like interacting with database, can remain in your action. The availability of action instance in JSF components makes the need of configuration file optional. The backing bean definitions and navigation rules are placed in `faces-config.xml` file. The backing bean definitions and navigation rules can be handled by Struts.

To use this plugin, simply place the `struts2-jsf-plugin-2.0.6.jar` into `WEB-INF/lib` folder and extend your package from `jsf-default` package, which is defined in `struts-plugin.xml` file in the JAR of this plugin. Here's the action mapping with `jsfStack` as interceptor stack and a `jsf` type result configured:

```
<action name="employee" class=
    "org.apache.struts.action2.showcase.jsf.EmployeeAction">
    <interceptor-ref name="basicStack"/>
    <interceptor-ref name="jsfStack"/>
    <result name="success" type="jsf" />
    <result name="index" type="redirect-action">index</result>
</action>
```

In this mapping, we can use the page with JSF components and can have complete access to the JSF life-cycle. The JSF plugin doesn't have any customizable settings. The configuration settings for JSF plugin in the `struts-plugin.xml` file is contained in the JAR of the JSF plugin .The `<result-types>` and `<interceptors>` tags are needed to be configured for using the JSF plugin in your Web application:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="jsf-default" extends="struts-default">

        <result-types>
            <result-type name="jsf" class="org.apache.struts2.jsf.FacesResult" />
        </result-types>
        <interceptors>
            <interceptor class="org.apache.struts2.jsf.FacesSetupInterceptor"
name="jsfSetup" />
            <interceptor class="org.apache.struts2.jsf.RestoreViewInterceptor"
name="jsfRestore" />
            <interceptor
class="org.apache.struts2.jsf.ApplyRequestvaluesInterceptor"
name="jsfApply" />
            <interceptor
class="org.apache.struts2.jsf.ProcessValidationsInterceptor"
name="jsfValidate" />
            <interceptor
```

```
class="org.apache.struts2.jsf.UpdateModelValuesInterceptor"
name="jsfupdate" />
<interceptor
class="org.apache.struts2.jsf.InvokeApplicationInterceptor"
name="jsfInvoke" />

<interceptor-stack name="jsfStack">
    <interceptor-ref name="jsfSetup">
        <param name="variableResolver">
            org.apache.struts2.jsf.StrutsVariableResolver
        </param>
        <param name="navigationHandler">
            org.apache.struts2.jsf.StrutsNavigationHandler
        </param>
    </interceptor-ref>
    <interceptor-ref name="jsfRestore" />
    <interceptor-ref name="jsfApply" />
    <interceptor-ref name="jsfValidate" />
    <interceptor-ref name="jsfUpdate" />
    <interceptor-ref name="jsfInvoke" />
</interceptor-stack>
</interceptors>
<default-interceptor-ref name="jsfStack"/>

</package>
</struts>
```

The Pell Multipart Plugin

The Pell Multipart plugin gives instruction to Struts to use the Jason Pell's Multipart parser for processing the file uploads. Before going into the details of the Pell Multipart plugin, let's first discuss file uploading in the Struts 2 Framework. The Struts 2 Framework comes with built-in support for file uploading.

When the `FilterDispatcher` receives a request, it checks the request to find whether the request contains multipart content. If the request contains multipart content, then the dispatcher creates a `MultipartWrapperRequest`. This wrapper handles the receiving and saving of the file on to the disk. The programmer should check if any error has occurred during processing. There are three properties that affect file uploading and that can be set by putting a `struts.properties` file in the `WEB-INF/classes`. Any property found in the properties file will override the default value. These three properties are mentioned as follows:

- ❑ `struts.multipart.parser`—This property should be set to a class, which extends `MultiPartRequest`. The Struts 2 Framework comes up with the Jakarta `FileUpload` implementation.
- ❑ `struts.multipart.saveDir`—This property sets the directory where the uploaded files will be placed. The default value for this property is `javax.servlet.context.tempdir`.
- ❑ `struts.multipart.maxSize`—This property gives the maximum file size in bytes, which is allowed for upload. The default value for this property is two Gigabytes. If we are uploading more than one file then the `maxSize` applies to the combined total, not the individual sizes of the files.

While you can set these properties to new values at runtime, the `MultipartWrapper` is created and the file handled before the Action code is called. So, in case you want to change the values, you must do so before this Action. Here's the code, given in Listing 11.1, showing the JSP page for file uploading:

Listing 11.1: File uploading.jsp

```
<%@ taglib uri="action2" prefix="saf" %>

<html>
<head>
<title>File Upload Test</title>
</head>
<body>
<h1>File Upload</h1>
<form action="FileUpload.action" method="POST" enctype="multipart/form-data">
<center>
<table width="350" border="0" cellpadding="3" cellspacing="0">
<tr>
<td colspan="2"><input type="file" name="fileName" value="Browse..." size="50"/></td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="Submit">
</td>
</tr>
</table>
</center>
</form>
</body>
</html>
```

In Listing 11.1, the user first selects a file to upload and after pressing the 'Submit' button, the file is uploaded successfully. No coding is required, as the file will be placed in the default directory. However, that leaves us with no error checking among other things. So let's add some more code to the Action.

Continuing our discussion on the Pell Multipart plugin, one of the important features of this plugin is that it uses the Jason Pell's Multipart parser. If you have installed multiple plugins, which provides multipart parsers, you can configure the Pell parser in your `struts.properties` file as follows:

```
struts.multipart.parser=pell
```

Table 11.3 mentions the settings that can be customized.

Table 11.3 : Pell Multipart plugin settings

Setting	Description	Default	Possible values
<code>struts.multipart.parser</code>	It sets which parser to use.	pell	pell, jakarta, or any other class name that extends MultiPartRequest

Table 11.3 : Pell Multipart plugin settings			
Setting	Description	Default	Possible values
struts.multipart.saveDir	The directory where the uploaded files will be placed.	Value from javax.servlet.context.tempdir	Any path
struts.multipart.maxSize	The maximum file size in bytes to be allowed for upload.	2 Gigabytes	Any number in kilobytes

The Pell Multipart plugin can be used by just copying the struts2-pell-multipart-plugin-2.0.6.jar file into the lib directory of the WEB-INF folder of your Web application. The Pell Multipart library is needed to be downloaded and included in your Web application as well. Due to its LGPL license, the library cannot be included in the Struts distribution. The configuration settings for the Pell Multipart plugin in struts-plugin.xml file are as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <beantype="org.apache.struts2.dispatcher.multipart.MultipartRequest"
        name="pell"
        class="org.apache.struts2.dispatcher.multipart.PellMultiPartRequest"
    />
</struts>
```

The SiteGraph Plugin

The SiteGraph plugin is used to generate the graphical diagrams, which represent the flow of your Web application. SiteGraph plugin works by parsing your configuration files, action classes, and view files such as JSP, Velocity, and FreeMarker. After parsing these files, this plugin displays a visual map.

SiteGraph is a tool that renders out GraphViz (Graph Visualization Software) -generated images. These images depict the flow of your Struts-powered Web applications. The SiteGraph plugin requires that the GraphViz should to be installed on your computer and the *dot* executable should be in your command path.

GraphViz (Graph Visualization Software) is a way of representing the structural information in the form of diagrams comprising of the abstract graphs and networks. Automatic graph drawing is used in many areas, such as software engineering, database and web design, networking, and in visual interface for many other domains.

GraphViz is an open source software for representing graph visualization. It has several inbuilt main graph layouts and consists of web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings.

The Graphviz layout programs take the descriptions of graphs in a simple text language and then make diagrams in several useful formats, such as images and SVG for web pages, Postscript for inclusion in PDF documents or to display in an interactive graph browser. Graphviz has several useful features for the development of concrete diagrams, for example options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

These graphs are usually generated using the data from some external sources, but they can also be created and edited manually in the form of raw text files or within a graphical editor.

There are several important things that must be noticed, when looking at the output from the SiteGraph:

- Boxes**—The actions are represented by boxes shaded in red and the actions shaded in green color indicate a view file (JSP, etc).
- Links**—These are also of two types. Links are represented by using arrows. The green colored arrows indicate that there is no new HTTP request being made, while black colored arrows indicate a new HTTP request.
- Link labels**—Labels may sometimes contain additional useful information. For example, a label of href means that the link behavior is that of a hyper-text reference. The complete label behavior is mentioned as follows:
 - href—A view file references an action by name (generally ending with the extension .action)
 - action—A view file makes a call to the Action tag.
 - form—A view file is linked to an action using the Form tag.
 - redirect—An action is redirected to another view or an action.
 - ! notation—A link to an action overrides the method to be invoked.

SiteGraph requires that your view files be structured in a specific way. Since the SiteGraph has to read these files, only certain styles are supported by the SiteGraph plugin. The requirements for the SiteGraph are as follows:

- The JSP tags must use the s namespace.
 - In JSP, use the <s :xxxx /> tag.
 - In FreeMarker, use the <@s .xxx /> tag.
 - In Velocity: Not applicable
- Use of the Form and Action tag must link directly to the action name.

One can use the SiteGraph plugin using the following command:

```
java -cp ... -jar struts2-sitegraph-plugin-x.x.x.jar
      -config CONFIG_DIR
      -views VIEWS_DIRS
      -output OUTPUT
      [-ns NAMESPACE]
```

Here,

- CONFIG_DIR**—This is the directory which contains the struts.xml file.
- VIEWS_DIRS**—This is the comma separated list of directories containing JSPs, VMs, etc.

- OUTPUT – This is the directory where the output should go.
- NAMESPACE – This is the namespace path restriction (/ , /foo, etc.)

NOTE

One must either supply the correct classpath when invoking the SiteGraph tool or place the SiteGraph plugin in the same directory as that of the dependent JARs. Specifically, the ZWork2 jar, Struts jar and there dependencies must be included in the class path. Also, you must include action class files referenced in struts.xml. If the class path entries are not proper, the SiteGraph plugin will not function properly.

Once you run the SiteGraph, check the directory specified in the output argument (OUTPUT). In that directory you'll find two files – out.dot and out.gif. One can open the out.gif and can have a view of the Web application flow. However, you may also wish to run the out.dot file using a different GraphViz layout engine, so the original dot file is provided as well.

Some advanced users may also think of executing the SiteGraph from within the application. This would be required, if you are developing an application, which supports the plugin capabilities. In case, you wish to use the SiteGraph plugin through its API, instead of command line, you can do that also. All you need to do is to create a new instance of the SiteGraph and specify the Writer to output the dot content to and then call #prepare().

The command-line version of the SiteGraph does exactly this, but doesn't override the Writer as shown here:

```
SiteGraph siteGraph = new SiteGraph(configDir, views, output, namespace);
siteGraph.prepare();
siteGraph.render();
```

The SiteGraph plugin doesn't need any global settings. In order to run this plugin all you need is to copy the requisite JAR file into the lib directory of the WEB-INF folder of your Web application. However, SiteGraph also requires the dot package provided by the GraphViz. Download the latest version of the GraphViz and make sure that the dot executable is in your class path. If you are using Windows, the GraphViz will automatically install the dot.exe into your class path.

SiteMesh Plugin

The SiteMesh plugin enables some templates to access the Struts information. SiteMesh itself is a framework which is used as a web page layout designer. This helps in creating a large Web application with a large number of web pages with a consistent look and feel throughout the application. The SiteMesh plugin provides a similar layout to all pages with matching navigation scheme.

The approach taken by SiteMesh plugin is based on the GangOfFour Decorator design pattern, which parses the requested page from the Web server and obtains its properties and data to generate a final page with modifications.

SiteMesh plugin helps in building Portal type of web sites where some HTML page is included as a Panel within another page creating a visual window within a page. The technologies used in SiteMesh are Servlet, JSP, and XML which make it ideal to be used with other J2EE applications. SiteMesh is very extensible and is designed in such a way that it is easy to extend for custom needs.

There are many Web application platforms that are in common use, and even more frameworks. Typically, a web application is built for a specific framework on a specific platform (e.g. using custom JSP tag-libraries on the J2EE platform, a custom Perl collaboration API using CGI, or a pre-written application using PHP). A website may usually consist of many Web applications built with a variety of

technologies. However, while individually each of these Web applications may look and perform as expected, they are often hard to integrate with the page layout of the existing site and can often end up isolated, attached to the site merely by a hyperlink. The most common reason for this is that sometimes sites can be a mixture of custom developed pages/applications and pre-written *out-of-the-box* software. This software may not be configurable enough to give the desired look and feel similar to the rest of the site which has been developed using a different technology to the rest of the site, restricting the developer from using fragments of the application in other pages of the site.

For example, how many times have you been to a site that has a very consistent feel all the way through, only to get to the ‘Search-Results’ page and wonder if you were still on the same site? Or how many times have you seen a framework or Web application that you would love to use on your site, but it does not integrate with what you already have?

The solution is to work with the HTML that is generated by the various web-apps, instead of hacking at the underlying code (although SiteMesh is primarily geared towards HTML-based sites, it is built in an extensible manner allowing it to be easily adapted to other mediums, such as XML, WML, PDF, etc.).

Each application should produce a plain vanilla version of the HTML content it wants to display. How this is achieved does not matter—it could be a static HTML page, XML/XSL transformation, Servlet, CGI script, etc.

The HTML content is intercepted on its way back to the web browser where it is parsed. The contents of the `<head>` tag and the `<body>` tag are extracted as along with the meta-data properties of the page (such as `title`, `<meta>` tags and attributes of the `<html>` and `<body>` tags).

From the extracted meta-data, an appropriate decorator is determined. A decorator can be thought of as a skin for a page—it wraps a plain looking page with a consistent look and feel. This decorator then overlays the original page.

To use SiteMesh, it needs to be installed, configured, and have some decorators created. This process can be completed in minutes, and no coding experience is required. No changes need to be made to your existing web- pages/applications.

Now, we’ll continue our discussion on the SiteMesh plugin. SiteMesh plugin allows the SiteMesh templates to access the framework resources. The SiteMesh Framework stores all its value stack information in the form of request attributes. This means that if you wish to display the data, which is present on the stack, you can do so by using the normal tag libraries that are inbuilt in the framework. The SiteMesh plugin can use the Struts 2 tags in the Sitemesh decorator templates.

Using SiteMesh Plugin

In SiteMesh Framework’s architecture, the standard filter-chain starts with the `ActionContextCleanUp` filter, followed by other desired filters. In the end, the `FilterDispatcher` handles the request, usually by passing the request on to the `ActionMapper`. The primary purpose of the `ActionContextCleanUp` is to provide the SiteMesh integration. The dispatcher filter removes the obsolete objects from the request as and when the clean-up filter tells to do so. Otherwise, the `ActionContext` may also get removed before the decorator’s attempts to access it.

`ActionContextCleanUp` is a special filter designed to work with `FilterDispatcher` that allows the integration with the SiteMesh easily. Normally, ordering of the filters is such that the SiteMesh filter will go first, followed by the `FilterDispatcher`. This ordering is perfectly fine in most cases. However, in some cases, you may wish to access the Struts 2 features, including the value stack, from within the SiteMesh decorators. Since the `FilterDispatcher` cleans up the `ActionContext`, your decorator won’t have access to the data you want. By adding the `ActionContextCleanUp` filter, the `FilterDispatcher` will know not to clean up and instead defer cleanup to this filter. In that case the ordering of the filters should be as follows:

- ActionContextCleanUp filter
- SiteMesh filter
- FilterDispatcher

NOTE

If the ActionContext access is required within the decorators, the ActionContextCleanUp filter must be placed at the start of the filter-chain.

FreeMarker and Velocity Decorators

The SiteMesh plugin provides an extension of the SiteMesh PageFilter in order to help with the integration with Velocity and FreeMarker. The filters provide the standard variables and Struts tags, which are used to create the views in the template languages.

FreeMarker

The FreeMarkerPageFilter extends the SiteMesh PageFilter to allow direct access to the framework variables, such as \$stack and \$request.

The following variables are available for decorating the FreeMarker page:

- \${title} – This variable provides the content of <title> tag in the decorated page.
- \${head} – This variable provides the content of <head> tag in the decorated page.
- \${body} – This variable provides the content of <body> tag in the decorated page.
- \${page.properties} – This variable provides the content of the page properties.

Velocity

The VelocityPageFilter extends the SiteMesh PageFilter in order to allow access to the framework variables, such as \$stack and \$request. While configuring the VelocityPageFilter in the web.xml file, place the VelocityPageFilter between the ActionContextCleanUp and FilterDispatcher.

Here's Listing 11.2 showing how the filter chains are configured in the web.xml file:

Listing 11.2: Configuration of velocity in web.xml

```
<web-app>
  . . .
  . . .
  <filter>
    <filter-name>struts-cleanup</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ActionContextCleanUp</filter-class>
  </filter>
  <filter>
    <filter-name>sitemesh</filter-name>
    <filter-class>org.apache.struts2.sitemesh.FreeMarkerPageFilter</filter-class>
  </filter>
  <filter>
    <filter-name>struts</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts-cleanup</filter-name>
    <url-pattern>/*</url-pattern>
```

```

</filter-mapping>
<filter-mapping>
<filter-name>sitemesh</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
<filter-name>struts</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
. . .
. . .
</web-app>

```

The configuration settings for the SiteMesh plugin provided in `struts-plugin.xml` file, which is contained in `struts2-sitemesh-plugin-2.0.6.jar` file is as follows:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<bean class="org.apache.struts2.sitemesh.FreeMarkerPageFilter" static="true"
optional="true"/>
<bean class="org.apache.struts2.sitemesh.velocityPageFilter" static="true"
optional="true"/>
</struts>

```

The Spring Plugin

Spring plugin allows the creation of actions, interceptors, and results by Spring, which is an open source framework. This framework is developed to address the complexity of enterprise-level applications. The layered architecture of Spring Framework allows developer to be selective about the components being used and provide cohesive framework for the development of J2EE applications. Spring is a unique framework for the following reasons:

- ❑ The business objects can be managed easily as Spring focuses on it
- ❑ The layered architecture makes this framework both comprehensive and modular
- ❑ The code written in Spring is always easy to test and this makes it an ideal choice for test driven projects

Spring provides a centralized and automated configuration with all wiring of different application objects. This is done using the concept of Dependency Injection. The creation of core framework objects is enhanced by Spring plugin, which overrides Struts `ObjectFactory`. The `class` attribute in the Struts configuration corresponding to the `id` attribute in the Spring configuration is used to create an object. If the `class` is not found in Struts, then the configuration is autowired by Spring. In case of Actions, Spring 2's bean scope feature can be used to scope an Action instance to the session, application, or a custom scope, providing the advanced customization above the default per-request scoping. The features of Spring plugin are as follows:

- ❑ It allows the creation of actions, interceptors, and results by Spring Framework.
- ❑ All objects, which are created using Struts can be autowired by Spring.

- ❑ Spring plugin provides two interceptors to autowire actions.

Using Spring Plugin

In order to use the Spring plugin, simply include the `struts2-spring-plugin-2.0.6.jar` in the `WEB-INF/lib` folder of your Web application. In case there is more than one object factory in use, then we need to set `struts.objectFactory` in `struts.properties` to `spring` like this:

```
struts.objectFactory = spring
```

Alternately, a constant can be configured in `struts.xml` for the same like this:

```
<struts>
    <constant name="struts.objectFactory" value="spring" />
    ...
</struts>
```

Autowiring

The term *autowiring* supported by Spring, by default, can be defined as looking for the objects defined in Spring having the name similar to the name of the object property. We can set the autowiring mode by setting `struts.objectFactory.spring.autoWire` property like this:

```
struts.objectFactory.spring.autowire = type
```

The autowiring modes can be `name`, `type`, `auto`, and `constructor`. These different modes are defined as follows:

- ❑ `name` – This is the default option. It autowires by matching the name of the bean in Spring with that of the name of the property in your action.
- ❑ `type` – This autowires by looking for a bean, which is registered with Spring of the same type as the property in your action. This requires you to have only one bean of this type registered with Spring.
- ❑ `auto` – If `autowire` property is set to `auto`, then the Spring will attempt to auto-detect the best method for autowiring your action.
- ❑ `constructor` – If `autowire` property is set to `constructor`, Spring will autowire the parameters of the bean's constructor.

Struts Framework will now try to use Spring to create all its objects and, if it is not created, the framework will create it on its own. The process of enabling Spring and other application objects integration includes two steps:

- ❑ Configuring the Spring listener – Spring listener can be configured in `web.xml` as shown here with an sample mapping:

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

- Register your objects using the Spring configuration—The registration of the objects is done in the `applicationContext.xml` file as follows:

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

<beans default-autowire="autodetect">
    <bean id="personManager" class="com.acme.PersonManager"/>
    ...
</beans>
```

Initialization of Actions using Spring

Normally, in `struts.xml`, you specify the class for each action. If the `SpringObjectFactory` is being used, the Spring is asked to create action. All dependencies specified by the default autowire behavior is wired up. Most of the times, the developer may want Spring to manage all beans. So all the beans are configured in the Spring `applicationContext.xml` file and change the `class` attribute of the action in `struts.xml` to use the bean name defined in Spring.

Here's Listing 11.3 showing the changed attributes of the action class in the `struts.xml` file:

Listing 11.3: Configuration of spring plugin in struts.xml

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>

    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <action name="foo" class="com.acme.Foo">
            <result>foo.ftl</result>
        </action>
    </package>

    <package name="secure" namespace="/secure" extends="default">
        <action name="bar" class="bar">
            <result>bar.ftl</result>
        </action>
    </package>
</struts>
```

Suppose, you have a Spring bean defined in your `applicationContext.xml` named `bar`. The `com.acme.Foo` Action needn't be changed. A typical spring configuration for `bar` in the `applicationContext.xml` will look like the one shown here:

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

<beans default-autowire="autodetect">
    <bean id="bar" class="com.my.BarClass" singleton="false"/>
```

```
....  
....  
</beans>
```

The Spring plugin can be used by just copying struts2-spring-plugin-2.0.6.jar into the lib directory of the WEB-INF folder of your Web application. The struts-plugin.xml file contained in this JAR provides some configuration settings for Spring plugin, which is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE struts PUBLIC  
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  
"http://struts.apache.org/dtds/struts-2.0.dtd">  
  
<struts>  
  
<bean type="com.opensymphony.xwork2.ObjectFactory"  
name="spring" class="org.apache.struts2.spring.StrutsSpringObjectFactory" />  
  
<!-- Make the Spring object factory the automatic default -->  
<constant name="struts.objectFactory" value="spring" />  
  
<package name="spring-default">  
<interceptors>  
  
<interceptor name="autowiring"  
class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor"/>  
  
<interceptor name="sessionAutowiring"  
class="org.apache.struts2.spring.interceptor.  
SessionContextAutowiringInterceptor"/>  
</interceptors>  
</package>  
</struts>
```

The Struts 1 Plugin

Sometimes we have some Struts 1 action classes and ActionForm classes and we may want to use these components built using Struts 1 API in our Web application based on Struts 2. The Struts 1 plugin integrates the existing Struts 1 Action and ActionForm classes with our Struts 2 application. A class is provided by Struts 1 plugin to provide a wrapping for Struts 1 Action. This wrapping class, org.apache.struts2.s1.Struts1Action, enables the execution of Struts 1 Actions and ActionForms by converting the incoming and outgoing objects into the expected forms. Several interceptors are also provided by Struts 1 plugin to emulate Struts 1 flow logic. These interceptors are as follows:

- org.apache.struts2.s1.ActionFormValidatorInterceptor** – It integrates the validation of ActionForms into the workflow of Struts 2.
- org.apache.struts2.s1.ActionFormResetInterceptor** – It calls the reset() method on any discovered ActionForms.

Using Struts 1 plugin, Struts 1 Actions and ActionForms can be used in Struts 2 application without any change and this supports validation enabled ActionForms also. The action mapping provided in

`struts.xml` file is similar to what should have been provided for some Struts 2 action. However, some additional implementations are required in order to run a Struts 1 Action and ActionForm.

- ❑ The package should extend the `struts1-default` package, which is defined in `struts-plugin.xml` file contained in JAR for this plugin. The `struts1-default` package contains several interceptors and a default interceptor stack that work the plugin into the Struts 2 request process.
- ❑ The `class` attribute of the `action` element always takes `org.apache.struts2.s1.Struts1Action` as the class to be executed.
- ❑ The Struts 1 Action to be executed is defined using a parameter named `className`.

In the simplest example of using a Struts 1 Action in Struts 2, we would configure a Struts 2 action using the wrapper. The following code shows a simple Struts 1 configuration:

```
<action name="myAction" class="org.apache.struts2.s1.Struts1Action">
<param name="className">com.mycompany.myapp.MyAction</param>
<result>myAction.jsp</result>
</action>
```

In most of the cases, you'll have an ActionForm that your Struts 1 Action expects. In order to use an ActionForm, you'll need an interceptor which manages the creation and scope of the ActionForm. Here's the code showing a Struts 1 Action with session-scoped ActionForm:

```
<package name="struts1_default" extends="struts1-default">

    <interceptors>
        <interceptor name="someForm"
            class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor">
            <param name="className">com.kogent.form.SomeForm</param>
            <param name="name">someForm</param>
            <param name="scope">session</param>
        </interceptor>
    </interceptors>

    <action name="someAction" class="org.apache.struts2.s1.Struts1Action">
        <param name="className">com.kogent.action.SomeAction</param>
        <param name="validate">true</param>

        <interceptor-ref name="someForm" />
        <interceptor-ref name="struts1Stack" />
        <result>success.jsp</result>
        <result name="error">error.jsp</result>
    </action>

</package>
```

This plugin doesn't support any global settings at all. In order to use the Struts 1 plugin in your Web application, you only need to copy the `struts2-struts1-plugin-2.0.6.jar` file into the `lib` directory of the `WEB-INF` folder of your Web application. The Web application will need Struts 1 JAR in order to function correctly. The `struts1-default` package, interceptors and default interceptor stacks

introduced by Struts 1 plugin is configured in `struts-plugin.xml` file contained in the JAR of this file.

Here's the code that shows the settings for the Struts 1 plugin in the `struts-plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="struts1-default" extends="struts-default">

        <interceptors>
            <interceptor name="actionForm-reset"
                class="org.apache.struts2.s1.ActionFormResetInterceptor"/>
            <interceptor name="actionForm-validation"
                class="org.apache.struts2.s1.ActionFormValidationInterceptor"/>
            <interceptor name="actionForm-commonsValidation"
                class="org.apache.struts2.s1.ActionFormValidationInterceptor">
                <param name="pathnames">
                    /org/apache/struts/validator/validator-rules.xml,
                    /WEB-INF/validation.xml
                </param>
            </interceptor>
            <interceptor-stack name="struts1Stack">
                <interceptor-ref name="static-params"/>
                <interceptor-ref name="scoped-model-driven"/>
                <interceptor-ref name="model-driven"/>
                <interceptor-ref name="actionForm-reset"/>
                <interceptor-ref name="basicStack"/>
                <interceptor-ref name="actionForm-validation"/>
                <interceptor-ref name="workflow"/>
            </interceptor-stack>
        </interceptors>

        <default-interceptor-ref name="struts1Stack"/>
    </package>

</struts>
```

The Tiles Plugin

The Tiles plugin allows the actions to return Tiles pages. Tiles are a templating framework designed to easily allow the creation of Web application pages with consistent look and feel. Tiles can be used for decorating and componentization of page. The Tiles plugin supports tiles in FreeMarker, JSP, and Velocity. The following steps should be taken in order to use tiles in the Struts 2-based applications:

- Include the `struts-tiles-plugin` as a dependency in your Web application.
- For using tiles listener, the listener will either be the standard tiles listener, i.e. `org.apache.tiles.listener.TilesListener` or the Struts 2 replacement `org.apache.struts2.tiles.TilesListener`. The latter provides tighter integration with Struts features, such as FreeMarker integration, like this:

```
<listener>
<listener-class>org.apache.struts2.tiles.StrutsTilesListener</listener-class>
</listener>
```

- All package definitions which require tiles support must either extend the tiles-default package or must register the Tiles Result type definition like this:

```
<result-types>
<result-type name="tiles" class="org.apache.struts2.views.tiles.TilesResult"/>
</result-types>
```

- Configure your actions to utilize a tiles definition:

```
<action name="sample" class="org.apache.struts2.tiles.example.SampleAction" >
<result name="success" type="tiles">tilesworks</result>
</action>
```

The following example shows a Tiles layout page using Struts 2 tags:

```
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib prefix="s" uri="/struts-tags" %>

<%-- Show usage; used in Header --%
<tiles:importAttribute name="title" scope="request"/>
<html>
<head><title><tiles:getAsString name="title"/></title></head>
<body>
    <tiles:attribute name="header"/>
    <p id="body">
        <tiles:attribute name="body"/>
    </p>
</body>
</html>
```

NOTE

Tiles plugin has been discussed in detail in Appendix C.

Now that you are aware of the plugins, let's discuss the third party plugins, which support the Struts 2 Framework.

Implementing Third Party Plugins

Third party plugins are those that don't come with the Struts 2 Framework installation. Even though, they support the Struts 2 Framework. They can be used for providing additional features to the Struts 2 Framework. The list of third party plugins, which supports the Struts 2 Framework are as follows:

- **Groovy Standalone plugin**—It enables the use of actions classes and interceptors written in Groovy.
- **GWT plugin**—It enables calling of Struts action methods using Google Web Toolkit.

- ❑ **JSON plugin** – It provides a result-type json, which serializes actions into JSON.
- ❑ **Scope plugin** – It implements JBoss Seam-style scoped bijection.
- ❑ **Spring MVC Plugin** – It enables Spring MVC controllers and interceptors execution in Struts 2 applications.
- ❑ **Spring Webflow plugin** – It integrates Spring Webflow (SWF) with Struts 2.
- ❑ **Table tags** – They are Struts 2 tags for displaying the table data.

Let's describe a few of these plugins in detail.

Groovy Standalone Plugin

The basic objective of Struts 2 Scripting Support (S2SS) project is to provide support for different scripting languages in Struts 2. These scripting languages include Groovy, Jython, JRuby and others. The Groovy plugin is part of the Struts 2 Scripting Support project and provides support for Actions (and Interceptors) written in Groovy language.

This is called a standalone plugin, because it can be used without Spring, which makes it a good candidate for smaller projects where Spring might be considered overkill, or until Spring enables prototyping of scripted beans. You can either build it from scratch by checking out the source and running ant, or you can simply download the JAR and drop it in your /WEB-INF/lib folder. Edit /WEB-INF/classes/struts.properties (or create it if it doesn't exist yet) and add the following property:

```
struts.objectFactory=groovyObjectFactory
```

The basic requirements for using the Groovy Standalone Plugin are as follows:

- Struts 2.0.x (struts2-api-2.0.x.jar, struts2-core-2.0.x.jar)
- Groovy 1.0 (groovy-all-1.0.jar)
- XWork 2.x (xwork-2.0.x.jar)
- Commons Loggings 1.0 (commons-logging-1.0.4.jar)

Configuration in struts.xml

Before the Groovy ObjectFactory kicks in, you should register your action. If the classname ends with .groovy, the Groovy ObjectFactory will load the filename defined by the class attribute. You should provide the path from the WEB-INF/classes root, so if you have a file called Test.groovy under /WEB-INF/classes/com/acme/groovy/, you should use the following:

```
<action name="groovy" class="com.acme.groovy.Test.groovy">
    <result>/WEB-INF/jsp/groovy.jsp</result>
</action>
```

The same goes for Interceptors you want to use as shown here:

```
<interceptor name="groovyInterceptor" class="com.acme.groovy.Interceptor.groovy"
/>
..
<interceptor-ref name="groovyInterceptor" />
```

Here's an example of a groovy script. The name of the groovy script is `Test.groovy`. Place this script under `/WEB-INF/classes/com/acme/groovy/`. The code for `Test.groovy` is as follows:

```
package com.acme.groovy

class Test {

    def message

    def execute() {
        message = "Hello world"
        return "success"
    }

    def getMessage() {
        return message
    }
}
```

You can now use any other Java classes in your classpath, as well as other Groovy scripts (for example, if you do: `def f = new FunkyObject()`, then Groovy will look for a `FunkyObject.groovy` file in your classpath).

After creating the `Test.groovy`, create a file named `groovy.jsp` for displaying the result and place this file under `/WEB-INF/jsp/`. The code for `groovy.jsp` is as follows:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Groovy</title>
    </head>
    <body>
        <s:property value="message"/>
    </body>
</html>
```

The preceding `groovy.jsp` page code is used to display the message 'Hello world' given in `Test.groovy`.

The Spring MVC Plugin

The Spring MVC plugin allows the Spring MVC controllers and interceptors to be executed in Struts 2. The Spring MVC plugin is useful for the Spring MVC developers who want to migrate to Struts without rewriting their existing Spring MVC controllers and interceptors.

The important features of Spring MVC plugin are as follows:

- This plugin includes a Struts 2 action that can execute Spring MVC controllers.
- The Spring MVC plugin includes a Struts 2 interceptor that can execute Spring MVC interceptors.
- The plugin allows the Spring MVC view to be accessed via the OGNL stack.

- ❑ This plugin can be used by copying the plugin JAR into your application's lib directory of the WEB-INF folder. No other files need to be copied or created.

Using MVC Plugin

The first step in using the Spring MVC plugin is to configure your Spring MVC controller in Spring like this:

```
<bean id="ratingController" class="example.RatingController">
</bean>
```

Now the controller needs to be configured in Struts 2 as follows:

```
<action name="CalcRate"
class="com.googlecode.struts2springmvc.SpringMVCControllerAction">
    <param name="controllerName">
        ratingController
    </param>
    <result name="success">
        /example/displayrate.jsp
    </result>
</action>
```

The controllerName parameter in the preceding code is the name of the controller in the Spring configuration. The ModelAndView viewName is mapped to the Struts 2 result. The model Map is exposed as a model property on the value stack and can be accessed as follows:

```
<s:property value="model['rate']"/>
```

The plugin architecture supported by Struts 2 Framework, provides the flexibility of plugging different functionalities. These plugins provide integration with other technologies and also provide the extension points in the framework.

Let's now move on to the "Immediate Solutions" section to know how to implement some of the most frequently used plugins provided with the Struts 2 distribution.

Immediate Solutions

The application created in this section will use different plugins with full descriptions on how they are created, arranged, and configured to extend the functionalities being provided by the framework.

The `struts-plugin.xml` file associated with each plugin, provide its basic additional configuration required to integrate that particular plugin with the existing framework. The `struts-plugin.xml` file provides definitions for new packages, new classes, interceptors and result types to support different technologies when integrating different plugins in your application.

We already know that different configuration files in the Struts 2 based Web application are loaded in some order, i.e. first the `struts-default.xml` file, then the `struts-plugin.xml` file, and finally the `struts.xml` file. The loading of `struts.xml` file makes sure that all new configurations added by `struts-plugin.xml` file can easily be used in our `struts.xml` file. For example, we can extend the package defined in `struts-plugin.xml` file. Let's create an application with some added plugins and find how they extend the framework provided functionalities.

The Web application developed here, named `Plugs`, follows a standard directory structure which we are familiar with. The directory structure showing all code files used in the application are displayed in Figure 11.1.

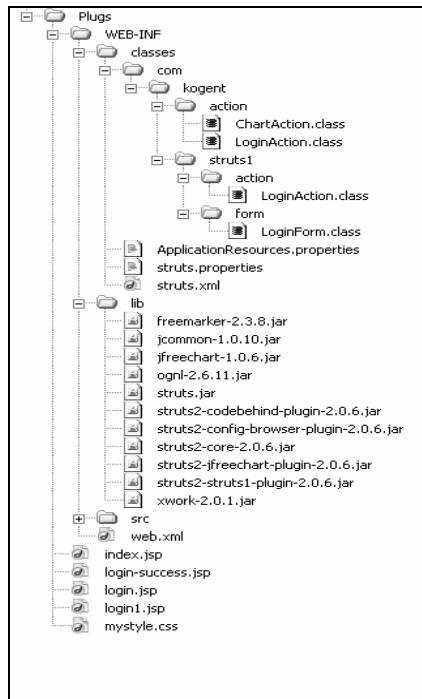


Figure 11.1: Directory structure of the application.

You can copy files, like `web.xml`, `struts.properties`, and `ApplicationResources.properties` with all JAR files to be used in the application and can place them according to the folder hierarchy shown in Figure 11.1. The remaining code files will be discussed and created in the coming sections of this chapter.

The first JSP page (`index.jsp`) of the application provides hyperlinks to navigate and run different examples, which implement different bundled plugins. The four plugins to be discussed through this application are Codebehind plugin, Struts 1 plugin, JFreeChart plugin, and Config Browser plugin. The Plugs application is a combination of four different examples implementing different plugins.

Here's the code, given in Listing 11.4, for `index.jsp` (you can also find `index.jsp` file in `Code\Chapter 11\Plugs` folder in CD)

Listing 11.4: index.jsp file

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" /></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body bgcolor="#3366CC" class="white">
        <h1>Welcome</h1>
        <table cellspacing="5" cellpadding="5">
            <tr><td bgcolor="#FFFFFF" >
                <s:a href="login.jsp">Codebehind Plugin</s:a>
            </td></tr>
            <tr><td bgcolor="#FFFFFF" >
                <s:a href="login1.jsp">Struts 1 Plugin</s:a>
            </td></tr>
            <tr><td bgcolor="#FFFFFF" >
                <s:a href="chartAction.action">JFreeChart Plugin </s:a>
            </td></tr>
            <tr><td bgcolor="#FFFFFF" >
                <s:a href="config-browser/index.action">Config Browser Plugin </s:a>
            </td></tr>
        </table>
    </body>
</html>
```

The output of the `index.jsp` page shows four hyperlinks, as shown in the Figure 11.2.



Figure 11.2: Output of index.jsp showing hyperlinks.

Implementing Codebehind Plugin

The first example in Plugs application implements Codebehind Plugin and this can be executed by clicking over the ‘Codebehind Plugin’ shown in Figure 11.2. This plugin helps in searching the matching page if the action being requested is not found mapped in struts.xml file. Similarly, for a given action mapping, the Codebehind plugin searches for the appropriate matching result (.jsp, .vm, .ftl) according to the result code retuned by the action class. This all need some naming convention, while naming action, and other pages to be displayed for result code ‘success’, ‘input’, and ‘error’.

Let’s create login.jsp and login-success.jsp pages and single action class LoginAction. The login.jsp page is designed with a form having two input fields for username and password. The data from this form is to be submitted to LoginAction action class. The result code ‘success’ from the action will show login-success.jsp page to the user displaying some message.

Here’s the code, given in Listing 11.5, for login.jsp (you can also find login.jsp file in Code\Chapter 11\Plugs folder in CD):

Listing 11.5: login.jsp page

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" />
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <center>
            Enter User Name and Password.<br><br>
            <table>
                <tr><td>
                    <s:actionerror />
                    <s:form action="login">
```

```
<s:textfield name="username" label="User Name"/>
<s:password name="password" label="Password" />
    <s:submit value="Login"/>
</s:form>
</td></tr>
</table>
<br> | <s:a href="index.jsp">Home</s:a> |
</center>
</body>
</html>
```

The output of login.jsp is shown in Figure 11.3, which shows two input fields and a single submit (Login) button.



Figure 11.3: The login.jsp showing input fields.

Here's the code, given in Listing 11.6, for login-success.jsp page (you can also find login-success.jsp file in Code\Chapter 11\Plugs folder in CD):

Listing 11.6: login-success.jsp page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<div align="center">
<h2><s:text name="app.success_login"/></h2>
<a href="index.jsp">L o g o u t</a></div>
</body>
</html>
```

The action form to be used here is `LoginAction` with two input properties `username` and `password`. This type of action classes are used earlier so we are just putting the code of this action class here in Listing 11.7 (you can also find `LoginAction.java` file in `Code\Chapter 11\Plugs\WEB-INF\src\com\kogent\action` folder in CD):

Listing 11.7: com.kogent.action.LoginAction.java

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String execute() throws Exception {
        if(username.equals(password))
            return SUCCESS;
        else{
            this.addActionError(getText("app.invalid"));
            return ERROR;
        }
    }

    public void validate() {
        if ( (username == null) || (username.length() == 0) ) {
            this.addFieldError("username",
                getText("app.username.blank"));
        }
        if ( (password == null) || (password.length() == 0) ) {
            this.addFieldError("password",
                getText("app.password.blank"));
        }
    }
}
```

After compiling and placing `LoginAction.java` class in the appropriate folder and saving `login.jsp` and `login-success.jsp` page in the project folder `Plugs`, let's relate them together by providing an action mapping.

Here's the code, given in Listing 11.8, for creating struts.xml file (you can also find struts.xml file in Code\Chapter 11\Plugs\WEB-INF\classes folder in CD):

Listing 11.8: struts.xml file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>

    <package name="default" extends="struts-default">

        <action name="login" class="com.kogent.action.LoginAction">
            </action>
    </package>

</struts>
```

Here, you can see that the action mapping provided with name="login" has no result configured here. Even no global result is configured in struts.xml file. So, how is next view, according to the result code returned by the LoginAction action, going to be searched.

The answer is our Codebehind plugin. Simply add struts2-codebehind-plugin-2.0.6.jar into WEB-INF/lib folder and the Codebehind plugin is ready to work. After loading the struts-default.xml file, the struts-plugin.xml file is loaded which is contained in almost all JARs for different plugins. This is enough for the Codebehind plugin, which reduces the configuration codes inside the configuration file in our struts.xml file.

Let's see how it works. For an action with name login in namespace / and for result code, say success, the different pages that will be searched are given here in the order they are searched:

- /login-success.jsp
- /login.jsp
- /login-success.vm
- /login.vm
- /login-success.ftl
- /login.ftl

In our example, if our LoginAction returns success, then the page to be searched is login-success.jsp page. The output of this page is shown in Figure 11.4.



Figure 11.4: Output of login-success.jsp.

Similarly, if the result code is `input`, the page to be displayed is `login-input.jsp` or `login.jsp` or `login-input.vm` and so on. This time the `login.jsp` is displayed with all field errors, as shown in Figure 11.5.



Figure 11.5: The login.jsp being shown as a consequence of 'input' as result code.

Hence, we find that we can set different JSP pages as different results providing them a name according the naming convention followed by Codebehind plugin. This reduces the configuration codes inside the configuration files. Similarly, this plugin can search the matching page to be shown for the action that is not even configured.

Implementing Struts 1 Plugin

The Struts 1 plugin enables us to use Struts 1 Actions and Struts 1 ActionForms in our Struts 2 application. This plugin provides some classes to work as wrapper and some interceptors to manage the Struts 1 flow of execution. So, let's create some JSP pages used in this example (login1.jsp and login-success.jsp)—Struts 1 LoginAction class and Struts LoginForm class. The login1.jsp page is similar to login.jsp, except that the action attribute of <s:form> element is set to different action, i.e. struts_login.

Here's the code, given in Listing 11.9, for login1.jsp (you can also find login1.jsp file in Code\Chapter 11\Plugs folder in CD):

Listing 11.9: login1.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
    <head>
        <title><s:text name="app.title" />
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <center>
            Enter User Name and Password.<br><br>
            <table bgcolor="#f0edd9">
                <tr><td>
                    <s:form action="struts1_login">
                        <s:textfield name="username" label="User Name"/>
                        <s:password name="password" label="Password" />
                        <s:submit value="Login"/>
                    </s:form>
                </td></tr>
            </table>
            <br> | <s:a href="index.jsp">Home</s:a> |
        </center>
    </body>
</html>
```

The Struts 1 action class (`com.kogent.struts1.action.LoginAction`) and Struts 1 ActionForm (`com.kogent.struts1.form.LoginForm`) implement the same logic what is implemented by the earlier Struts 2 action class `com.kogent.action.LoginAction` class.

Here's the code, given in Listing 11.10, for Struts 1 action class, `com.kogent.struts1.action.LoginAction` (you can also find this `LoginAction.java` file in Code\Chapter 11\Plugs\WEB-INF\src\com\kogent\struts1\action folder in CD) :

Listing 11.10: com.kogent.struts1.action.LoginAction.java

```
package com.kogent.struts1.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
```

```
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import com.kogent.struts1.form.LoginForm;

public class LoginAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        ActionErrors errors=new ActionErrors();
        LoginForm loginForm = (LoginForm) form;
        if("kogent".equals(loginForm.getUsername()) &&
           "kogent".equals(loginForm.getPassword()))
            return mapping.findForward("success");
        else{
            return mapping.findForward("error");
        }
    }
}
```

Here's the code, given in Listing 11.11, for Struts 1 ActionForm, `com.kogent.struts1.form.LoginForm` (you can also find `LoginForm.java` file in `Code\Chapter 11\Plugs\WEB-INF\src\com\kogent\struts1\form` folder in CD):

Listing 11.11: `com.kogent.struts1.form.LoginForm.java`

```
package com.kogent.struts1.form;
import org.apache.struts.action.*;
public class LoginForm extends ActionForm {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Now, we'll see the settings applied in order to use these Struts 1 Action and ActionForm classes in Struts 2 application. Follow the steps given here:

- Add `struts.jar` (Struts 1 JAR) into `WEB-INF/lib` folder to support the compilation and working of all Struts 1 classes.

- ❑ Add struts2-struts1-plugin-2.0.6.jar file in WEB-INF/lib folder to implement Struts 1 plugin.
- ❑ Create a package extending struts1-default package defined in struts-plugin.xml file in the JAR for Struts 1 plugin.
- ❑ Add an action mapping and interceptor mapping, as shown in Listing 11.12

Listing 11.12: New package definition added into struts.xml file

```
<package name="struts1_default" extends="struts1-default">

    <interceptors>
        <interceptor name="loginForm"
            class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor">
            <param name="className">com.kogent.struts1.form.LoginForm</param>
            <param name="name">loginForm</param>
        </interceptor>
    </interceptors>

    <action name="struts1_login" class="org.apache.struts2.s1.Struts1Action">
        <param name="className">com.kogent.struts1.action.LoginAction</param>
        <param name="validate">true</param>
        <interceptor-ref name="static-params"/>
        <interceptor-ref name="loginForm" />
        <interceptor-ref name="model-driven"/>
        <interceptor-ref name="actionForm-reset"/>
        <interceptor-ref name="basicStack"/>
        <interceptor-ref name="actionForm-validation"/>
        <interceptor-ref name="workflow"/>

        <result>login-success.jsp</result>
        <result name="error">login1.jsp</result>
    </action>
</package>
```

The action mapping, shown here, declares that Struts1Action class is being used as the wrapper around the Struts 1 Action, i.e. com.kogent.struts1.action.LoginAction declared as className parameter. The implementation of ActionForm is handled by configuring a ScopedModelDrivenInterceptor interceptor with the name loginForm, which is defined with parameters, like className and name.

Run the example, given in Listing 11.12, by clicking on ‘Struts 1 Plugin’ hyperlink, as shown in Figure 11.2. This brings you the login1.jsp page. Enter ‘kogent’ as username and password to see the success page, which happens to be login-success.jsp page created and used in the previous example.

Implementing JFreeChart Plugin

The JFreeChart plugin provides a ‘chart’ result type, which helps in generating a chart to be displayed. The plugin uses other APIs, like JFreeChart and JCommon, which provide a class to create a Chart. This example requires the creation of an action class to generate a chart and the configuration to be added in struts.xml file. The following JAR files are to be added into WEB-INF/lib folder of your application:

- ❑ struts2-jfreechart-plugin-2.0.6.jar
- ❑ jfreechart-1.0.6.jar
- ❑ jcommon-1.0.10.jar

NOTE

The struts2-jfreechart-plugin-2.0.6.jar comes with Struts 2 distribution, but another two jar files are to be downloaded separately. You can also copy all jar files from the application folder provided in the CD along with this book.

Now let's create an action class `ChartAction`, which uses various classes from JFreeChart API. Here's the code, given in Listing 11.13, for this action class (you can also find `ChartAction.java` file in `Code\Chapter 11\Plugs\WEB-INF\src\com\kogent\action` folder in CD):

Listing 11.13: `ChartAction.java`

```

package com.kogent.action;

import java.util.Random;

import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.axis.ValueAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.StandardXYItemRenderer;

import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

import com.opensymphony.xwork2.ActionSupport;

public class ChartAction extends ActionSupport {

    private JFreeChart chart;

    public String execute() throws Exception {
        // chart creation logic...
        XYSeries dataSeries = new XYSeries(new Integer(1));
        for (int i = 0; i <= 100; i++) {

            dataSeries.add(i, new Random().nextInt(50));
        }

        XYSeriesCollection xyDataSet = new XYSeriesCollection(dataSeries);

        ValueAxis xAxis = new NumberAxis("Marks");
        ValueAxis yAxis = new NumberAxis("Age");
        chart = new JFreeChart("Chart Title",
                JFreeChart.DEFAULT_TITLE_FONT,
                new XYPlot(xyDataSet, xAxis, yAxis,
                new StandardXYItemRenderer(StandardXYItemRenderer.LINES)),
                false);
        chart.setBackgroundPaint(java.awt.color.white);

        return super.SUCCESS;
    }
}

```

```
}

public JFreeChart getChart() {
    return chart;
}

}
```

Let's add a new package definition with single action mapping. Here's Listing 11.14 showing you the new package definition:

Listing 11.14: New package definition added in struts.xml

```
<package name="jfree" extends="jfreechart-default">
    <action name="chartAction" class="com.kogent.action.ChartAction">
        <result name="success" type="chart">
            <param name="width">500</param>
            <param name="height">300</param>
        </result>
    </action>
</package>
```

The package `jfree` here extends `jfreechart-default` package, defined in the `struts-plugin.xml` file, associated with JFreeChart plugin which makes the result type definition for chart type of result available here to be used. The `ChartAction` can be invoked using the 'JFreeChart Plugin' shown in Figure 11.2. A chart gets displayed, as shown in Figure 11.6.

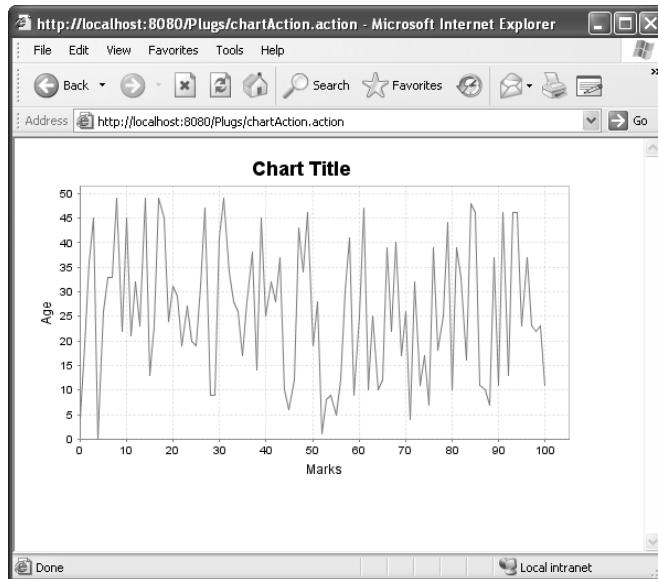


Figure 11.6: A chart displayed using JFreeChart plugin.

Implementing Config Browser Plugin

The last plugin implemented in Plugs Web application is Config Browser plugin. The name suggests that it provides a browsable configuration details about all the packages and action defined in the application. The configuration detail also includes all results and their types, names, and parameter defined for different actions in the application. In addition, the list of all interceptors, exception mapping, properties and Validators is provided in this plugin.

To implement this plugin, simply add struts2-config-browser-plugin-2.0.6.jar file in WEB-INF/lib folder of your application. The struts-plugin.xml coming with this JAR is loaded and makes different actions available, which are responsible for giving all the configuration details. The plugin comes with its own namespace creation with the name config-browser. The configuration provided in struts-plugin.xml file of this JAR defines a package with name= "config-browser" and gives action mapping to different plugin classes in the same namespace.

Invoke the index action of config-browser namespace using the URL <http://localhost:8080/Plugs/config-browser/index.action>. The same can be obtained by clicking over the 'Config Browser Plugin' hyperlink, as shown in Figure 11.2. This bring you a page similar to what is shown in Figure 11.7.

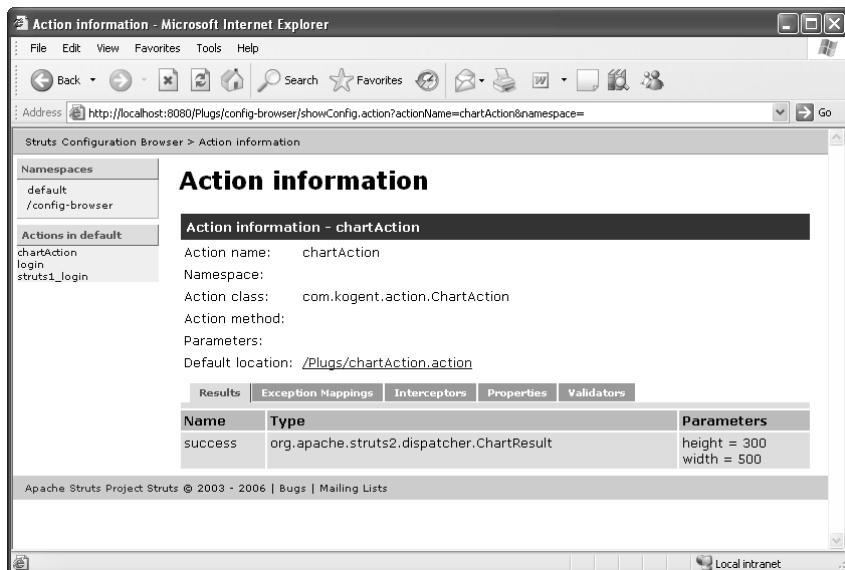


Figure 11.7: Application configuration detail using Config Browser plugin

You can see all the namespaces, actions in the left pane in the page, as shown in Figure 11.7. The pages being shown here are again added by the plugin JAR. You can click on the hyperlink to see other information, like results, exception mappings, interceptors, properties and validators.

In this chapter, we discussed all the bundled and third party plugins, which are supported by Struts 2 Framework. This discussion also included the functionalities provided by these plugins and the extension points offered by these plugin. The "Immediate Solutions" section further described the implementation details of different plugins through examples, showing how easy it is implemented.

In the next chapter, we'll discuss how different security policies are implemented in a Struts 2 based Web application.



12

Securing the Struts 2 Application

If you need an immediate solution to:

See page:

Implementing Transport Level Security	454
Implementing Role-Based Access Control	455
Using Authentication and Authorization	457
Using Container-Managed Security	462
Applying Login Configuration	464
Application-Managed Security	469

In Depth

This chapter focuses on the levels of security that you'll need to implement and secure your Struts applications. There are different security levels. First of all we'll discuss the various security levels that are available for securing Struts applications and then the two major approaches to implement security are discussed—container-managed security and application-managed security.

The main objective of this chapter is to elaborate how security mechanisms can be incorporated and common vulnerabilities, as far as your Struts application is concerned, can be avoided. For a developer, security of an application is one of the main concerns. An application without security issues being well taken care of is worth nothing, since it may be easily changed maliciously or its logic can be easily be hacked, thereby creating problems to the developers as well as for the users. The security mechanisms include the things, like user identification, authentication, and access control of sensitive actions, data integrity, and confidentiality. Though the break-ins caused by vulnerabilities get more publicity in the press, both, incorporating security features as per application requirement and avoiding vulnerabilities, are equally important.

Most of the Web applications always need certain prospects of the system to be secured in some manner. The security requirements are often specified at two levels—system level and the functional level—and these levels are important for a developer. For example, the access to confidential information should be given on the SSL channel. However, in case of functional requirements, it is recommended that the users access certain pages and menu items with administrative privileges only. Now it is up to the developer to decide which of the requirements can be satisfied using standard security mechanisms. In many of the cases, you'll find that the developers are using a combination of standard security mechanisms and customization to achieve their required security policy.

Understanding Levels of Security

Let's now get familiarized with the common security levels available for Web applications. If you go for a brief definition of security, you would find that it encompasses everything from encryption to personalization. We can define the security levels as follows:

- Transport level security
- Authentication
- Authorization
- Role-based access control
- Container-managed security
- Application-managed security

Transport Level Security

The Struts platform facilitates secure communication by adopting transport layer integrity and confidentiality mechanisms, based on SSL/TLS protocols. This ensures a tamperproof encrypted message between the communicating entities. The SSL/TLS transport security properties are configured

at the container level and will be applied to communication, while creating the connection. In addition, Struts provides configuration features that enforce protected communication and reject unprotected requests and responses.

Authentication

Authentication is a process by which a sender can prove himself as an authorized person as well as the receiver can prove himself as an authorized person for receiving the information. We can take banking transaction, like at ATMs, as an example. At an ATM counter, your ATM card is only validated for you. Your bank account cannot be accessed by anyone other than you, until or unless the third person knows your password. In Web application, the authentication is always achieved by entering the username and password. Only if you provide the validated information, will the server allow you to access the features of the website that is meant for an authorized user. A wide variety of authentication schemes is available for Struts applications and supported by J2EE Web containers. These are further discussed later under the section “Immediate solutions.”

Authorization

Authorization allows you to find out the resources available in the Web application for access. The main concern in authorization is ‘permission’. Authorization mechanism limits the interaction with resources to collections of users or systems for enforcing integrity, confidentiality, or availability constraints. Such a mechanism allows only the authenticate user to access the components. This is the process by which the web may choose to restrict certain users from carrying out certain operations. In other words, a user is able to perform only those operations for which he or she is authorized, not more than that. A limited degree of authorization is possible by restricting access to specific URLs or collection of URLs based on the user roles. It is also possible to limit the access to specific HTTP methods, such as GET, POST, PUT, DELETE, and so on. Such authorization rules can be specified in the Deployment Descriptor of the Web application. Some of the authorization rules may require the knowledge of application-specific entities and it may not be possible to specify them in a Deployment Descriptor. A different approach—one involving active participation of the logic of the Web application—is required for specification and enforcement of such authorization rules.

Role-based Access Control

Role-Based Access Control (RBAC) is a technique, which is used for implementing authorization. In this technique, the container-specific means are used for assigning the roles to the users. After the authentication process is over, the user's roles are connected with the HTTP request. Based on the roles defined for the users, access to certain pages is allowed or disallowed. Usually, RBAC allows users to have access to more than one role. However, no hierarchical relationship exists between roles allowed to access.

Container-Managed Security (CMS)

The Web container plays an important role in building and deploying secure Web applications. The Servlet specification outlines what security features should be part of the environment provided by the Web container and how a specific Web application should make use of them. By using the CMS, a developer can specify how the authentication is to be carried out and what authorization needs to be granted. It provides an easy way of adding security to an application. Implementing the security using the CMS technique provides the following advantages:

- ❑ It is declarative. Authentication and authorization both are defined in the web.xml file. All container-specific details, such as security areas, are provided in the server-specific XML configuration files.
- ❑ It supports multiple types' authentication process, which includes password authentication, form-based authentication, authentication using encrypted passwords, and authentication using client-side digital certificates.
- ❑ Using the CMS technique, one can store the data in flat files, relational databases, and Lightweight Directory Access Protocol (LDAP) server.
- ❑ In this technique, redirects are not controlled manually. When visiting a secured URL, the container is first asked for user credentials. After this, if the user is authenticated, the user is redirected to a page in the URL automatically. This technique is most common in email communications.
- ❑ CMS is easy to implement, but it avoids the flexibility of your security policy. Due to differences in the security implementation of the containers, the application becomes less portable. To resolve this problem, developers usually follow the application-managed security.

Application-Managed Security (AMS)

Sometimes, we do not find container-managed security mechanism suitable for our application. The container-managed security does not provide the required amount of flexibility to customize the security implementations in application. This can only be achieved by a different mechanism, known as Application Managed Security (AMS). In case of application-managed security, you have to create your own security structure, which is similar to container-managed security. Sometimes, you'll come across issues common in the AMS and CMS. For this, you should consider the following points, while resolving the common security issues:

- ❑ How and when is the authentication done?
- ❑ How the authorization level is determined?
- ❑ What type of privileges the users have?
- ❑ Is SSL required for transferring the confidential data?

Now we'll move to the "Immediate Solutions" section to understand how these different security mechanisms can be implemented to make a Struts application secure.

Immediate Solutions

Implementing Transport Level Security

The transmission of data can also be performed in a secure fashion using HTTP over Secure Socket Layer (SSL), also known as HTTPS. SSL provides the encryption security while transporting data. This means that all the data transported from one end to the other end is in encrypted form. The Struts application can transfer the data using SSL without any changes. In order to work with HTTPS, you have to take into consideration the magnitude of performance. SSL also improves the performance and due to this, switching between the HTTP and HTTPS takes place. When you send some sensitive information, like login and submission information, to the server, it needs to use the HTTPS protocol. Once the server receives data, the protocol switches back to HTTP.

Integrating Struts with SSL

The Struts platform facilitates secure communication by adopting transport layer integrity and confidentiality mechanisms, based on SSL/TLS protocols. SSL/TLS-based communication security can be specified for Web components and Struts components, including their Web services endpoints. It is the responsibility of the Struts application's Deployment Descriptor to identify the components with method calls whose parameters or return values should be protected for integrity or confidentiality. The component's Deployment Descriptor is used to represent this information.

For secure communication with Web components, such as Servlets and JSP pages, you have to configure the `<transport-guarantee>` sub-element of the `<user-data-constraint>` that is present inside the `<security-constraint>`. In cases, where a component's interactions with an external resource are known to carry sensitive information, these sensitivities should be described in the `<description>` sub-element of the corresponding `<resource-ref>`.

Here's the code, given in Listing 12.1, with a Web Deployment Descriptor snippet (`web.xml`) showing the `<transport-guarantee>` sub-element:

Listing 12.1: Configuring `<transport-guarantee>` sub element in `web.xml`

```
<security-constraint>
    .
    .
    <user-data-constraint>
        <transport-guarantee>
            CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
    .
    .
</security-constraint>
```

Therefore, the HTTPS is very much preferable in cases where sensitive data is passed, whereas in other cases, HTTP is used. This requires redirecting from non-secure pages to secure pages and back again and the redirect needs switching the protocol scheme on a URL from HTTP to HTTPS on each redirection.

One of the drawbacks of using this protocol switching is that the absolute URLs must be hard coded into JSP pages and action classes, which lead to the development and deployment problems that arise when server names are different between development, integration test and production server. To reduce this problem, developers attempted to use the SSLEXT as a rescue point.

NOTE

The SSLEXT is short form of SSL extension, and developed under open-source program. It acts as a plug-in for Struts.

This approach is generally useful while implementing Struts with SSL. It has the following features:

- ❑ It gives information related to securing the action mapping inside the struts configuration file. You can use this feature to change protocols between actions and JSP pages required in your application.
- ❑ SSLEXT contains a plug-in class, which is used for initialization purpose; this class is a customized extension to the Struts RequestProcessor and Struts ActionMapping.
- ❑ SSLEXT can also be provided by some additional JSP tags that let you know whether your entire JSP page is secure or not. All this can be achieved by enabling the SSL channel for your server, which hosts the applications.

Implementing Role-Based Access Control

Role-based access control is achieved by changing the server configuration. We are using Tomcat 5.5.23 Web server to achieve this control. To learn about more about Tomcat, refer the Tomcat documentation present in the installation folder of Tomcat on your system, and list login requirements for two bundled Web applications—Tomcat Administration and Tomcat Manager. Access tp Tomcat Administration is limited to users with *admin* role and that of Tomcat Manager to users with *manager* role. We'll talk about role-based access control for Web applications later in this chapter.

To explore these bundled Web applications, you need to create the roles—*admin* and *manager*—and a user with these roles. Here's Listing 12.2 showing the modification of the default user file *tomcat-users.xml* file, located in the *conf* subdirectory of Tomcat home.

Listing 12.2: tomcat-users.xml

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
<role rolename="tomcat"/>
<role rolename="role1"/>
<role rolename="admin"/>
<role rolename="manager"/>
<user username="tomcat" password="tomcat" roles="tomcat"/>
<user username="role1" password="tomcat" roles="role1"/>
<user username="both" password="tomcat" roles="tomcat,role1"/>
<user username="jstkuser" password="changeit" roles="admin, manager"/>
</tomcat-users>
```

Can you spot the name and password of the newly added user? The username is *jstkuser* and the password is our favorite *changeit*. It is understood that you should select something better as a password (less intuitive) for your production environment. In fact, managing users with such a text file with clear-text passwords is not recommended for a production environment for multiple reasons. Anyone with read access to this file has access to all the usernames and their passwords. In addition, as the number of users and roles grows, the management of the text file becomes difficult. Fortunately,

Tomcat works with a relational database, an LDAP (Lightweight Data Access Protocol) server, or any other kind of server to access the user and role information.

NOTE

Please refer to the online documentation for details.

Why can't Tomcat distribution have a default user with admin and manager roles? That would certainly make it faster to explore administration and manager applications for new users. However, this would achieve ease of use at the cost of security. What would happen if an administrator forgets to change the default? Anyone would be able to run these applications with default username and password and completely compromise the security of the site. In fact, many security breaches are traced to the insecure default settings. The overhead of initial setup is a small price to pay for better overall security.

There is still one problem. As we mentioned earlier, the password is stored in clear-text and can be read by anyone with read access to the users file `conf\tomcat-users.xml`. This is certainly not desirable for high security sites. We'll learn later in this chapter on how to get rid of clear text passwords.

Tomcat reads the users file only during startup – meaning you must stop and restart it after making any modifications to this file. Now that we have modified this file, stop Tomcat and then start it. We are now ready to launch Tomcat Administration and Tomcat Manager Applications. Let's start with Tomcat Administration by clicking the appropriate link from the Tomcat home page. This takes us to a login page. Supply `jsttkuser` as the username, `changeit` as password and click the 'Login' button.

Explore the entities in the left pane. Some of these are higher-level entities consisting of subentities and can be expanded or collapsed. Clicking on an entity in this pane displays the entity properties in the right pane and lists the available actions in a drop-down box. For example, clicking on Roles displays a list of currently active roles and allows the creation of new roles and removal of the existing ones. The same is true with the entity Users. You notice that the roles and users displayed here are the same that we saw in the `conf\tomcat-users.xml` file. This is no coincidence. As per the default configuration, the Administration Tool operates on the user account information stored in this file. As expected, these actions will eventually update the `conf\tomcat-users.xml` file.

NOTE

You can learn about other entities, such as Tomcat Server and Service, and their sub-entities by reading the online documentation.

Most of them have counterparts in the Tomcat configuration file `conf\server.xml`. The information in this file captures the configuration of Tomcat installation and the running instance. You can change this configuration either through the Administration Tool or by directly editing the `server.xml` file.

Among all the elements, the ones to pay particular attention to are Service, Connector, Realm, Host, and Context. As the name suggests, a connector is a component responsible for accepting connections. Recall that Tomcat listens for browser requests on port 8080. Do you see a Connector for this port? What are its properties and which ones can you modify? Logout from Administration Tool by clicking on theLogout button, located at the top right-hand side of the screen. We are finished exploring this tool.

Another tool of interest is Tomcat Manager. Invoke this tool from the default Tomcat homepage. This also requires login, but the login panel decidedly appears different from the login page of the Administration Tool. The login page of the Administration Tool is created by the application itself to carry out what is known as Form Based Authentication. The login panel, shown by the browser for the Tomcat Manager, has resulted from what is known as HTTP Basic Authentication. We'll talk more about different authentication mechanisms later in this chapter.

The Tomcat Manager allows you to view the currently deployed Web applications, go to start page of any of the deployed Web applications, deploy new Web applications, and stop or reload the existing

ones. As we learned earlier, some of these activities (especially deployment) can also be done by simply moving files and modifying appropriate configuration files under the Tomcat home directory.

A noteworthy aspect of the login to the Tomcat Manager is that you do not have a ‘Logout’ button and there is no way to log in as a different user, other than starting a separate instance of the browser. This has to do with the way HTTP Basic Authentication works.

Let’s shift our attention to the server side setup for Tomcat. We have already come across two files in the conf subdirectory:

- ❑ `server.xml` – It contains configuration information about the Tomcat. The Realm element within the Engine element of this file links the user database to file `conf\tomcat-users.xml`.
- ❑ `tomcat-users.xml` – It contains information about the current users, roles and association of users with roles.

Other files of the subdirectory `conf` that are of interest to us are as follows:

- ❑ `web.xml` – It contains Web application descriptor elements common to all the Web applications. The elements in this file are merged with the elements of the Web application-specific Deployment Descriptor to form the complete Deployment Descriptor.
- ❑ `catalina.policy` – It has the Java security policy entries for the Tomcat software.

Besides these files, the directory `webapps` is of special interest. As we have already seen, a WAR file or a subdirectory tree conforming to Java Web application structure moved into this directory gets deployed as a Web application. Another way of deploying a Web application is to add a Context element to a Tomcat configuration file `server.xml` or place an XML file containing a Context element within the `webapps` directory. When you move a WAR file or a directory tree to the `webapps` directory, the base directory of the Web application is within this directory. It is possible to specify a base directory outside the `webapps` directory for Web applications deployed through a Context element by setting the `docBase` attribute of the Context element. This is the case with the Administration Tool.

Here’s the Tomcat Manager, given in Listing 12.3, to set a context path:

Listing 12.3: Setting context path

```
<Context path="/admin" docBase=".//server/webapps/admin"
    debug="0" privileged="true">
    <!--Context sub-elements commented out -->
</Context>
```

Using Authentication and Authorization

The process of an application authenticating a user, based on username and password assumes that the application knows the username and the corresponding password (or is somehow able to confirm its validity) for all its valid users. It is also important to make sure that only the rightful owner knows the password and no one else. There is usually an initial setup phase to accomplish this.

Most applications either maintain their own user account systems or access an existing one. A new user must sign up or register to create a user account and obtain or select a username and password. Most e-commerce sites allow self-registration through the browser itself. They do not really care about your real identity. The only information associated with you, which is taken care, is the credit card number and some times an e-mail id to receive messages.

A user account management system must take reasonable precautions to make sure that the passwords are not easily guessable and are immune to known attacks, such as dictionary attack. It may also implement the policy of mandatory change every few months. In cases where a user forgets his or her password, it should be possible to assign a new password after verifying the identity claim through other means. This process may be automated or may require going through a human operator.

Self-registration over the Internet could make a site vulnerable to denial of service attacks. One or more automated programs could start creating bogus accounts, locking up resources, and denying the service to genuine users. A defense against such attacks is to present a camouflaged visual pattern during registration and ask the user to recognize the pattern. Automated programs are usually not good at identifying such patterns.

In some cases, the user may already have an account. This would be the case if your brick-and-mortar bank started offering Internet banking or your employer started offering employee services through Web browsers.

Because of the overhead and inconvenience involved in creating a user account and remembering the corresponding username and password, most websites allow the same user account to be used for all the Web applications belonging to that website. This is made possible through a single sign-on system that maintain the user accounts and interface with different applications for user authentication.

All this implies that a Web application should be able to interact with different user account systems at the time of deployment. This requirement is not directly addressed by J2EE specification, but Web containers usually provide a mechanism to accomplish this.

User Authentication Schemes

The process of authenticating users, or client programs running on behalf of the user, for Web applications is different from the one specified by JAAS (Java Authentication and Authorization Service). The main difference is that a Web application runs within the context of a Web container and is accessed by a user through a Web browser, over HTTP, whereas JAAS is designed for scenarios where the user-facing component and the backend component are both Java programs running within the same JVM. JAAS does not have to worry about secure exchange of sensitive username and password information between two programs, possibly over an insecure network. It is possible to use JAAS for distributed programs where the authenticated subject is transmitted from one program to another, provided both are Java programs and the security of the exchange is guaranteed through other means.

An insecure network could allow data to be read and/or altered by a malicious third party. There are many kinds of issues of authentication mechanism, which the Web applications must address.

HTTP/1.1 specification defines two authentication schemes for authenticating a client to a server:

- Basic authentication scheme
- Digest authentication scheme

Besides these schemes, a Web application can authenticate a user by prompting the user for a username and password by sending an HTML FORM to the browser, getting this information within a HTTP POST request and passing it to a user account system for validation, very much the same way as is done by JAAS. This scheme is also known as *FORM-based authentication scheme*, due to its reliance on the HTML FORM element. A fourth authentication scheme is possible when the underlying transport for HTTP is SSL. Recall that SSL can optionally require the client to present its own X.509 certificate and prove possession of the corresponding private key. This scheme is known as *Certificate-based authentication scheme*.

A Web container is required to support all, but the HTTP Digest authentication schemes. We'll cover the specifics of this support in the next section. The focus of this section is to learn about these mechanisms independent of what setup is needed in a Web container to make them work.

Basic Authentication Scheme

This scheme uses a technique known as a challenge-response mechanism that has the following steps in the authentication process:

- ❑ The client requests the server for a resource, with the request URI included in the HTTP request identifying the resource.
- ❑ The server responds that access to this resource is limited to authenticated users and challenges the client to provide client credentials (i.e. username and password information).
- ❑ The client repeats the request, including the username and password information as part of the request.
- ❑ The request succeeds if the server is able to validate the supplied user credentials. Otherwise, the server resends the challenge response.

Here's the code, given in Listing 12.4, that shows the relevant portions of a successful exchange between a client and server using Basic authentication scheme:

Listing 12.4: Exchange between client and server using Basic Authentication.

```
[1]C --> S
GET /rmb2/index.jsp HTTP/1.1
... skipped ...

[2]C <-- S
HTTP/1.1 401 Unauthorized
... HTTP headers skipped ...
WWW-Authenticate: Basic realm="RMB2 Basic Authentication Area"
... skipped ...

[3]C --> S
GET /rmb2/index.jsp HTTP/1.1
... HTTP headers skipped ...
Authorization: Basic cGFua2FqOnBhbmtthag==
... skipped ...
```

Essentially, the server challenge is an HTTP header `WWW-Authenticate` whose value consists of the authentication scheme identification (Basic) and a realm string (RMB2 Basic Authentication Area). The realm value is an opaque string assigned by the server to partition its URI space into multiple protection spaces, each having its own unique realm value.

The client response is also an HTTP header. This header is named `Authorization` and its value consists of the authentication scheme identification (Basic) and base64 encoded value of `username:password` string (`cGFua2FqOnBhbmtthag==`). Subsequent requests for resources in the same protection space use the same `Authorization` header.

Although the user credentials portion of `Authorization` field appears to be inscrutable at first sight, it can easily be converted back to the original string having the username and password in clear-text by base64 decoding. This makes Basic authentication a rather weak mechanism as anyone who can snoop on the network can collect the username and password. The problem is made worse by the fact that

people usually keep the same username and passwords for multiple accounts and a weak authentication mechanism of one account could lead to compromise of other accounts as well.

Recall the authentication process of the Tomcat Manager and the browser-generated login panel. That process was done using HTTP Basic authentication. The browser caches the username and password information and uses this value for all requests URIs with the same realm string.

Digest Authentication Scheme

The Digest authentication scheme is somewhat similar to the Basic authentication mechanism. The only feature that makes it different from Basic authentication is that in the Digest authentication scheme the credentials, like password, is never transmitted over the wire in a form that can be used by the snooper. That is why this authentication mechanism is more secure than the basic authentication. Though, it is not widely deployed, still it is good to know how this mechanism does the transmission of password in a secure manner.

Here's Listing 12.5 showing how Digest authentication is able to avoid transmission of password over the wire:

Listing 12.5: Exchange between client and server using Digest authentication

```
[1] C --> S
GET /rmb2/index.jsp HTTP/1.1
... skipped ...

[2] C <-- S
HTTP/1.1 401 Unauthorized
... HTTP headers skipped ...
WWW-Authenticate: Digest realm="RMB2 Digest Authentication Area", \
qop="auth", nonce="f9b9c89377323747f5b3825093f31a0b", \
opaque="ac052870edb30301762b7e860ef75deb"
... skipped ...

[3] C --> S
GET /rmb2/index.jsp HTTP/1.1
... HTTP headers skipped ...
Authorization: Digest username="pankaj", \
realm="RMB2 Digest Authentication Area", \
qop="auth", algorithm="MD5", \
uri="/rmb2/index.jsp", \
nonce="f9b9c89377323747f5b3825093f31a0b", \
nc=00000001, cnonce="b0f1d2743b114410b19beea2dd8778e0", \
opaque="ac052870edb30301762b7e860ef75deb", \
response="7956e10c4fa50167f9a7a562dbbbfbcd"
... skipped ...
```

The `realm` attribute here has the same meaning as it has in the Basic authentication and is used to partition the URI space into multiple authentication zones. In fact, the browser treats Digest authentication in a manner, similar to Basic authentication in many other ways also. These include prompting the user with the browser generated login panel and caching the username and password for the life of the browser execution.

The use of `request uri` and `nonce` value (shown in Listing 12.5) in computing the hash provides protection against replay attacks. A server would be able to detect if a third party captures the request and tries to issue the captured request at a later point in time or for another request URI.

FORM-Based Authentication Scheme

Till now, we know that in the authentication mechanism one provides the username and password that are transmitted in clear text as a part of HTTP POST request body. This is insecure like that of Basic authentication mechanism. However, if such insecurity could be avoided by performing some process, like encryption, which is implemented in case of HTTP traffic flows over SSL, then this mechanism can be a good one.

In fact, the FORM-based authentication over SSL is quite popular for a number of reasons, some of which are as follows:

- The application can control the look and feel of the login window.
- The error message, on unsuccessful login, can be customized to display a friendly message.
- The application has more control over 'logged in' status of the user, allowing a user to logout once the security sensitive operation is over.

Certificate-Based Authentication Scheme

Certificate-based authentication scheme is one of the strongest and reliable authentication schemes among all the authentication schemes described so far. This requires configuration of the server to accept HTTP requests over SSL and demand some client certification. When the client is a browser, it searches for a conforming certificate and a private key in its certificate store. Here the browser may prompt the user to confirm a particular certificate or may prompt to select one from the multiple conforming certificates. Always a certificate is issued to an entity that contains a corresponding private key, which is used for client authentication. As the certificates with their private keys are typically password protected, the user will have to provide their corresponding certificate's password as well.

Though this scheme is one of the most powerful schemes for authentication, still it is not widely used in Web application. This is due to the complexity, which involves while implementing this authentication mechanism.

Authorization

Once a client is authenticated, a Web container needs to perform authorization checks. Authorization policies configured in an enterprise describe which users or groups are allowed to access protected Web resources. The Application Assembler, in the Deployment Descriptor of the Web module packaging a Web application, specifies which security roles are required in order to access a URI managed by a Web container. Because an Application Assembler is probably not familiar with the deployment environment of the enterprise in which the application is deployed, the Application Assembler specifies authorization policies in terms of J2EE security roles, with a security role being a named collection of J2EE authorizations, implemented as `java.security.Permission` objects. When a security role is mapped to a user or a group in an enterprise, then that user or group is authorized to access the resources protected by those permissions associated with the security role. For example, if the travel organization has established that access to online reservations should be allowed only to customers, the Deployment Descriptor can specify that only the principals corresponding to the Customer security role are authorized to access the travel reservation URLs.

The application servers like WebSphere authorize access to all the J2EE resources, unless they are explicitly protected. Therefore, System Administrators can deploy Web resources without declaring them explicitly in a Web module's Deployment Descriptor.

The URL to which they are mapped accesses Web resources. An Application Assembler, therefore, declares authorization policies in the Web module's Deployment Descriptor in terms of URLs. In the Deployment Descriptor, security constraints are defined to restrict access to URLs. Such constraints are

associated with a set of URLs relative to the Web application's context path using the security-constraint XML element. For example, if the context path of the AAA travel application is /travel, the relative URL to access the ReservationServlet will be specified in terms of the relative URL /reserve.

The relative URLs to which the restriction applies can be defined via URL patterns. In the Web module's Deployment Descriptor, such a collection of URLs, known as a Web resource collection, is specified through a web-resource-collection XML element. Associated with a Web resource collection is its authorization constraint policy, specified by an auth-constraint element. An authorization constraint policy is specified in terms of J2EE security roles. Security administrators are responsible for managing authorization policies for the users and groups authorized to access the protected URIs.

Using Container-Managed Security

A Web container supports two kinds of security mechanisms—Declarative and Pragmatic. Declarative security allows implementing the security using a declarative syntax applied during the application's deployment, whereas Pragmatic security allows expressing and enforcing security decisions at the application's invoked methods and its associated parameters. This is upto the user to decide, in which case he has to use the declarative or pragmatic. In case of declarative security, a deployer specifies security characteristics, such as authentication mechanism, authorization to access certain URIs, user roles, etc. declaratively in the Web application Deployment Descriptor. In case of programmatic security, the access control rule and associated logic is directly implemented into the application. We'll discuss this security in Application-based security because pragmatic security is best-suited in case of application-managed security.

When a client interacts with a Struts application, depending upon the application component architecture it accesses a set of underlying components and resources, such as JSPs, Servlets, and other back-end applications. Because processing a client request involves a chain of invocations with subsequent components and resources, the Struts platform allows introducing a client authentication at the initial call request. After initial authentication, the client identity and its credentials can be propagated to the subsequent chain of calls.

Struts platform allows establishing user authentication in all application tiers and components. The Struts environment provides support for the following three types of authentication services:

- Agent-based authentication
- Container-based authentication
- Application-based authentication

Let's discuss them here.

Agent-Based Authentication

With this mechanism, Struts applications use a third-party security provider for authentication and this third-party security provider offers agents, typically to provide a single sign-on service to the portals, J2EE managed business applications. The agent usually resides as a proxy that intercepts the user requests to the J2EE server. Generally, to make the best utilization agent-based authentication mechanism, the J2EE server infrastructure uses JAAS-based authentication module to integrate custom authentication technologies.

Container-Based Authentication

To protect and control access to Web applications, the Web container facilitates authentication mechanisms that can be configured for a Web application prior to its deployment. When an

unauthenticated user attempts to access a protected Web application, the Web container will prompt the user to authenticate with the Web container using the configured authentication mechanisms. The user's request will not be accepted by the Web container until the user is authenticated to the Web container with its required credentials and identified to be one of the users with granted permission to access the application and its resources.

This is the standard authentication service provided by the J2EE server infrastructure. This allows the J2EE environment to authenticate users for access to its deployed applications. The J2EE specification mandates Web container support for four authentication types, which include the following:

- ❑ **HTTP basic authentication**—The Web container component authenticates a principal using a username and password entered in the dialog box presented at the time of accessing the secured Web application.
- ❑ **Form-based authentication**—This is similar to basic authentication, but the user needs to enter the username and password on the customized web form for passing to the Web container.
- ❑ **Client-certificate/mutual Authentication**—Both the client and server use X.509 certificates to establish their identities, and this authentication usually occurs over a secure communication channel using SSL/TLS protocols.
- ❑ **Digest authentication**—In this Authentication, the client is authenticated with a message digest containing the client password and the same is send with the HTTP request message.

Application-Based Authentication

In Struts application, while it approaches to use the Application-based authentication, the application relies on a pragmatic security approach. Using this approach, it collects the user's credential and starts verifying the identities against the security. Details about Application-based authentication are given later in this chapter.

To elaborate container-managed security, let's take a tour through an example to understand it. Suppose we want to implement the container-managed security to our application Customer Management System (CMS). Assume that this application resides on xyz Inc.'s corporate intranet. This application provides access to employee search section and everybody accesses it. However, it is only the administrators who can make the modification based on the employee's details from the database. If someone wants to access the administrator's functions (section meant for the modification of employee's information), then that user is prompted to input a user name and password. If both the information is correct to that of the value stored in the database, then the user acts as an administrator and the application redirects to the requested page.

All this functionality can also be achieved without making any modification to Java code or JSP codes of your applications, but it is a good practice to group URLs under role-specific path prefixes. Therefore, it needs to keep all the Java and JSP pages of modification section in the location meant for the admin only. It means, if you have add.jsp and add.action for adding employee's information into your application, then you have to shift this file administrator/add.jsp and administrator/add.action. For this purpose, you need to change your web.xml. Though it is a little difficult, still developers implement this, to make security easier.

Here's Listing 12.6 showing how web.xml is changed:

Listing 12.6: Customized web.xml to support Application-based authentication.

```
<web-app>
    [... snipped ...]
    <security-constraint>
        <web-resource-collection>
```

```
<web-resource-name>PagesUnderAdmin</web-resource-name>
<description>Pages for Administrator</description>
<url-pattern>/administrator/*</url-pattern>
</web-resource-collection>
<auth-constraint>
    <role-name>administrator</role-name>
</auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>CMS</realm-name>
</login-config>
<security-role>
    <description>CM Administrator</description>
    <role-name>administrator</role-name>
</security-role>
</web-app>
```

In the XML file in Listing 12.6, we have three XML elements, such as `security-constraint`, `login-config`, and `security-role`. These three files are added to define the security requirement. The `security-constraint` element contains a collection of page with a role. You can identify the pages using the patterns based on URLs. If the user attempts to access a page that matches one of the patterns that are described inside the configuration then the user is allowed to access the respective pages. The user is prompted to login according to the settings of `login-config` element, if not authenticated. If the user is authenticated and has administrative privileges then the user is redirected to the requested page.

In this configuration, observe the `url-pattern` of `security-constraint` element. This URL mapping can be achieved in the following four ways:

- ❑ **Explicit mapping**—In this mapping, no wildcards is used (e.g. /add.jsp or /administrator/remove.do).
- ❑ **Path prefix mapping**—This mapping contains a / or /* symbol, which is used as path prefix for specifying the branch of the Web application (e.g. /administrator/* or /search/company/*).
- ❑ **Extension mapping**—This mapping contains *, which is followed by a prefix. You can specify all the files of a particular extension using this mapping, e.g. *.jsp. Usually, Struts extensions are mapped using extension mapping.
- ❑ **Default mapping**—It is used for matching all the URLs. This mapping allows you to match all the URLs of a Web application. URL's are matched on the basis of the context path.

Before you use this Mechanism inside the Struts module, you must create some partitioning for your application so that each module will be in its own path off context root. This modularization makes implementation of security policies easier.

Applying Login Configuration

The `login-config` element present inside the `web.xml` indicates the type of authentication to be performed. It gives all the information related to the user. Inside a Web application, you'll find only one `login-config`. The `auth-method`, nested element indicates the type of authentication and accepts one of the following four types of values as listed in Table 12.1.

Table 12.1 Different types of authentications	
Authentication	Description
BASIC	The browser pops up a dialog box that allows the user to enter his username and password. The provided username and password are then encoded using the Base-64 algorithm (a Web encoding scheme that is often used to encode e-mail attachments) and sent to the server
FORM	It allows a custom form to be specified. The form is based on UserName_Var field for entering the username and a PassWord_Var field for entering the password. You need to submit the form to Security check for authentication and authorization
DIGEST	It is similar to Basic authentication scheme with a difference that the username and password are transferred in encrypted format into a message digest value
CLIENT-CERT	Authentication is done using digital certificate. It is secured, however costly. Certificates must be valid

Basic Login

The simplest way to get started with basic container-managed security is to use Basic authentication that involves the use of the security realm and this security behaves as a reference to container specific security storage. If you use Tomcat then you can achieve container-managed security through the Tomcat UserDatabase realm. Inside `tomcat-users.xml`, you are required to insert username, passwords, roles, and role assignments, which can be retrieved by the realm.

Here's the text to be added, given in Listing 12.7, to the `tomcat-users.xml` file, present in the `<Tomcat_Home>/conf` directory:

Listing 12.7: Configuring username, password and roles in `tomcat-users.xml` file

```

<tomcat-users>
    <role name="administrator"/>
    <user name="santosh" password="jena" roles="administrator" />
    <user name="praksh" password="kumar" roles="administrator" />
    <user name="kogent" password="india" roles="employee" />
</tomcat-users>

```

In the configuration file shown in Listing 12.7, we have defined two users—administrators and employee. Let's suppose that we are creating an application where we are doing two things, one is to add an employee into the databases and other is to search an employee from the database. The insertion of a new employee's information can only be performed by the administrator and the searching of an employee can be done by any user. Therefore, to perform the adding operation, the user must have an administrator and he will be prompted for his username and password.

After the successful entry of username and password, he will be redirected to the next page, i.e. for the adding operation of employee's information. Once the user is authenticated through this process, your Web application can gather the useful user data from the HTTP request.

Form-Based Login

Form-based login is another option for container-managed login configuration. To implement this Form-based login, we have to supply a web page that may be a JSP page or HTML page, which contains the login form and a page when it encounters errors. The main goal of using Form-based login approach is to increase the look and feel of your application. To observe the features of a Form-based login in the application described here in this chapter, you have to supply a web page that has a form for login, which will follow some guidelines. The form will submit two parameters, named `UserName_Var` and `Password_Var`, holding the username and password, respectively, to the `Securitycheck` URL optionally. These values, `Securitycheck`, `UserName_Var`, and `Password_Var` are specified by the Servlet API and are processed by the Web container. This is how the Web container is able to grab the username and password values and do the necessary validation. To display login failures and error conditions, it is also necessary to define an error page.

Here's the code, given in Listing 12.8, for `login_form.html` that specifies a web page that will be displayed, when an error is encountered:

Listing 12.8: `login_form.html`

```
<html>
<head>
<title>XYZ, Inc. Customer Management Portal</title>
</head>
<body>
<font size="+1">Portal Login</font><br>
<hr width="100%" noshade="true">
<form action="Securitycheck">
    Username: <input type="text" name="UserName_Var"/><br/>
    Password: <input type="password" name="Password_Var"/><br/>
    <input type="submit" value="Login"/>
</form>
</body>
</html>
```

In addition to this, you have to make changes to `login-config` element of `web.xml` to use the Form-based authentication. Here's Listing 12.9, showing the changes made to `login-config` element:

Listing 12.9: Configuring Form-based login

```
<login-config>
    <auth-method>FORM</auth-method>
        <realm-name>MiniCMRealm</realm-name>
        <form-login-config>
            <form-login-page>/login_form.html</form-login-page>
            <form-error-page>/login_error.html</form-error-page>
        </form-login-config>
    </login-config>
```

These changes result in a browser showing a new page, instead of a standard dialog box. Moreover, once it is authenticated the user is taken back to the index page, where the hidden link will be displayed. All these modifications are done to place the login form on the main index page; as a result, it reduces the number of mouse clicks and the page display. You may have seen that most of the Web applications use a login form as their main page, which provides a lot of information to everybody. However, if you'll move the login form to the index page and provide the username and password, you'll find that it does not work and provides an error 'HTTP Status 400 - Invalid direct reference to form login page'. This is a Tomcat generated error. It tells that the login page can only be managed by the container. When a user will access a protected page, it should trigger the display of a login page. This is a feature of the container-managed authentication workflow that, once the user is authenticated, the user is redirected to the requested protected page. If you'll browse the login page directly and submit the form, the container would not know where to redirect the request and will give an error as discussed before.

This is irritating, while working with the Form-based authentication. To overcome this problem we have to consider the container-managed secure transport. If you are able to implement this with the constraints of container-managed security, then it will provide a rich security to your application with minimal coding.

Container-Managed Secure Transport

This service operates by specifying whether the pages should be accessed using HTTPS. This information, you have to specify using the user-data-constraint sub element of the security-constraint element.

Here's the code, given in Listing 12.10, to find out how the security-constraint element is configured inside the web.xml file:

Listing 12.10: web.xml file

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>PagesunderAdmin</web-resource-name>
        <description>Pages for Administrator</description>
        <url-pattern>/administrator/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>administrator</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>
            CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

The main part in this code here is the `transport-guarantee`. This element can specify any one of the three options:

- NONE
- INTEGRAL
- CONFIDENTIAL.

NONE means there is no restriction and normal HTTP is used. The other two options, INTEGRAL and CONFIDENTIAL, use HTTPS.

The main thing to remember is that the use of user-data-constraint sub element does not require specification of an authentication constraint. It means you can define Web resources that are secure but those that do not require authentication.

Digest Authentication

The digest authentication mechanism provides the same features and user experiences as are provided by the Basic authentication mechanism. The only difference is, here username and the password are not sent directly to the server. All the information regarding the login information are sent followed by an encryption method. Encryption is one of the processes used for encoding. It will encode your original value to some other value, that may be an irregular text. Again, at the server level, this will be followed by a process decryption, which will decode your encoded values to the original values and then work according to the user's requests. Apart from this, all the processes behind this mechanism are quite similar to the Basic authentication mechanism. To secure a Web application using Digest authentication-based login access, you have to configure the login-config element in the Web component Deployment Descriptor web.xml. The login-config element contains an auth-method, which contains the element value DIGEST, which tells the client to transmit the username and password encrypted using a message digest.

Here's Listing 12.11 showing the configuration:

Listing 12.11: Setting <auth-method> to DIGEST

```
<web-app>
  . . .
    <security-constraint> . . . </security-constraint>
  <login-config>
    <auth-method>DIGEST</auth-method>
  </login-config>
  . . .
</web-app>
```

CLIENT-CERT Authentication

To implement this type of authentication technique, you require the help of digital certificate. This is a secured configuration. It allows configuring the Web container or Servlet engine to accept HTTP requests using bidirectional SSL and authenticate client users' public-key certificates issued by a trusted certificate authority (CA). To secure a Web application using Client-Certificate or Mutual Authentication-based login access, you have to configure the login-config element in the web.xml, the Web component Deployment Descriptor. The login-config elements have an auth-method, which contains the element value CLIENT-CERT.

Here's Listing 12.12 showing the configuration:

Listing 12.12: Setting <auth-method> to CLIENT-CERT

```
<web-app>
  . . .
    <security-constraint> . . . </security-constraint>
  <login-config>
    <auth-method>CLIENT-CERT</auth-method>
```

```
</login-config>
</web-app>
```

Application-Managed Security

Sometimes the developer needs to implement application-managed security mechanism to achieve security. This is because some applications need to supply their credentials used to connect to the resource themselves rather than relying on the values configured into the container. This can be achieved using the application-managed-security in the connector configuration.

We have generated the application in the container-managed security section. The following section gives the idea behind the application-managed security mechanism.

Creating a Security Service

Generally, the implementation of application-managed security requires developers to be clearer about the logic implementations with good programming practices which is not required in case of container-managed security. You need to thoroughly implement the traditional *best practices* development.

Let's consider an application where we add employees into the databases. The information about a new employee can only be added by the administrator. Therefore, to perform the adding operation the user must have to be an administrator and he will be prompted for his username and password. After successfully entering the username and password, the administrator will be redirected to the next page that is for the adding operation of employee's information.

Here's the code, given in Listing 12.13, of the `SecurityService` interface which can further be implemented by some class to provide basic authentication of the user through `authenticate()` method (you can find `SecurityService.java` file in `Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\security` folder in CD):

Listing 12.13: `SecurityService.java`

```
package com.kogent.struts.security;
import javax.security.sasl.AuthenticationException;

public interface SecurityService {
    public User authenticate(String username, String password)
        throws AuthenticationException;
}
```

Observe the Listing 12.14 for `SecurityServiceImpl` class, which implements the `SecurityService` interface and uses Hash Map to store user data in memory (you can find `SecurityServiceImpl.java` file in `Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\security` folder in CD):

Listing 12.14: `SecurityServiceImpl.java`

```
package com.kogent.struts.security;
import java.util.HashMap;
import java.util.Map;
```

```
import javax.security.sasl.AuthenticationException;

public class SecurityServiceImpl implements SecurityService {
    private Map users;
    private static final String ADMIN_ROLE = "administrator";
    private static final String CLERK_ROLE = "clrek";

    public SecurityServiceImpl(){
        users = new HashMap();
        users.put("santosh", new User( "santosh", "kogent", new String[]
            {ADMIN_ROLE, CLERK_ROLE}));
        users.put("prakash", new User( "prakash", "kogent", new String[]
            {CLERK_ROLE}));
    }

    public User authenticate(String username, String password) throws
    AuthenticationException {
        User user = (User) users.get(username);
        if (user == null) throw new AuthenticationException("Unknown user");
        boolean passwordIsValid = user.passwordMatch(password);
        if (!passwordIsValid) throw new AuthenticationException("Invalid password");
        return user;
    }
}
```

The `SecurityService` interface and its implementation class `SecurityServiceImpl` offers the basic security features to the application being created here. You'll find one method `authenticate()` inside this program, which verifies the user's username and password, and returns an object that represents the user.

Here's the code, given in Listing 12.15, for the `User` class which gives the information related to the user (you can find `User.java` file in `Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\security` folder in CD):

Listing 12.15: User.java

```
package com.kogent.struts.security;
import java.io.Serializable;
public class User implements Serializable {
    private String username;
    private String password;
    private String[] roles;
    public User() {
    }
    public User(String name, String pwd, String[] assignedRoles) {
        username = name;
        password = pwd;
        roles=assignedRoles;
    }
    public String getUsername() {
        return username;
    }
    boolean passwordMatch(String pwd) {
        return password.equals(pwd);
```

```

        }

    public boolean hasRole(String role) {
        if (roles.length > 0) {
            for (int i=0; i<roles.length; i++) {
                if (role.equals(roles[i])) return true;
            }
        }
        return false;
    }
    public boolean isAdministrator() {
        return hasRole("administrator");
    }
}

```

The User class has two String type fields—username and password. Along with this, you'll find that one of the methods added here is isAdministrator(), which proves whether the user is an administrator or not. Another method hasRole() is also present, which tells whether or not a user is assigned a given role and this method will be useful with customized Struts role processing. To tie this custom security criterion into the application, you need to create a class that will process the user login.

Here's the code, given in Listing 12.16, for LoginAction class (you can find LoginAction.java file in Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\action folder in CD):

Listing 12.16: LoginAction.java

```

package com.kogent.struts.action;

import javax.servlet.http.*;
import com.kogent.struts.security.SecurityService;
import com.kogent.struts.security.SecurityServiceImpl;
import com.kogent.struts.security.User;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletRequestAware;

public class LoginAction extends ActionSupport implements ServletRequestAware{

    private String username;
    private String password;
    private HttpServletRequest request;

    public void setServletRequest(HttpServletRequest request) {
        this.request=request;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {

```

```
        this.password = password;
    }
    public String execute() throws Exception {
        HttpSession session = request.getSession();
        SecurityService service = new SecurityServiceImpl();
        try {
            User user = service.authenticate(username, password);
            session.setAttribute("User", user);
            this.addActionMessage("You have Successfully Logged in.");
            return SUCCESS;
        } catch (Exception e) {
            this.addActionError(e.getMessage());
            return LOGIN;
        }
    }
    public void validate() {
        if (username == null) || (username.length() == 0)) {
            this.addFieldError("username", getText("app.username.blank"));
        }
        if (password == null) || (password.length() == 0)) {
            this.addFieldError("password", getText("app.password.blank"));
        }
    }
}
```

Here, in this program, the action gets the value of the username and password from a login form which can be designed as `login.jsp`, which is yet to be created. Then, it calls the `authenticate()` method of the security service. The `authenticate()` method returns an type `User`, given that the user with the entered username and password exists. In case there is no such user, the `authenticate()` method throws an `AuthenticationException`, which has been handled in `LoginAction` class (see Listing 12.16).

Now create `index.jsp` page to provide links for login and add a new employee which is accessed by only those users who have administrator role. Other users are to be restricted to access the `adddemployee.jsp` page.

Here' the code, given in Listing 12.17, for `index.jsp` (you can find `index.jsp` file in `Code\Chapter 12\Struts2Security` folder in CD):

Listing 12.17: `index.jsp`

```
<%@ page language="java" session="true" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Struts Security</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <br>
    <s:actionerror/>
    <s:actionmessage/>
```

```

<h2>A Secure Application</h2>
Welcome, <s:property value="#session['User'].username" default="Guest"/>!
<br><br>
<s:url id="addemp" action="addEmployeeAction"/>
<s:a href="#">Add Employee</s:a><br><br>
<s:set name="user" value="#session['User']"/>
<s:if test="#{user==null}">
<s:a href="login.jsp">Login</s:a>
</s:if>
<s:else>
<s:url id="logout" action="logoutAction"/>
<s:a href="#">
</s:else>
</s:else>
</body>
</html>

```

The other two JSP pages, which are used in this application, are `login.jsp` and `adddemployee.jsp`. Here's the code, given in Listing 12.18, for `login.jsp` (you can find `login.jsp` file in `Code\Chapter 12\Struts2Security` folder in CD):

Listing 12.18: `login.jsp`

```

<%@ page language="java" session="true" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Struts Security</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <s:actionerror/><br>

    <h3>Login!</h3>
    <s:form action="loginAction">
        <s:textfield name="username" label="User Name"/>
        <s:password name="password" label="Password"/>
        <s:submit/>
    </s:form>
</body>
</html>

```

Here's the code, given in Listing 12.19, for `adddemployee.jsp` (you can find `adddemployee.jsp` file in `Code\Chapter 12\Struts2Security` folder in CD):

Listing 12.19: `adddemployee.jsp`

```

<%@ page language="java" session="true" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Struts Security-Adding Employee</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />

```

```
</head>
<body>
    <h2>Add Employee</h2>
    <s:actionerror/>
    <s:form action="addEmployee">
        <s:textfield name="employeename" label="Employee Name"/>
        <s:textfield name="department" label="Department"/>
        <s:textfield name="company" label="Company"/>
        <s:submit value="Add Employee"/>
    </s:form>
    <s:url id="logout" action="logoutAction"/>
    <s:a href="#">Logout</s:a>
    <s:a href="#">index.jsp</s:a>| Home |</s:a>
</body>
</html>
```

The action, which is used to process the request to add a new employee is AddEmployeeAction class. This action checks whether the user is logged in with proper role before returning success as result code, which consequently shows addemployee.jsp page to the user. In this way, only the user having administrator role can access addemployee.jsp page.

Here's the code, given in Listing 12.20, for AddEmployeeAction class (You can find AddEmployeeAction.java file in Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\action folder in CD):

Listing 12.20: AddEmployeeAction.java

```
package com.kogent.struts.action;
import javax.servlet.http.*;

import com.kogent.struts.security.User;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletRequestAware;

public class AddEmployeeAction extends ActionSupport implements
ServletRequestAware{

    private HttpServletRequest request;

    public void setServletRequest(HttpServletRequest request) {
        this.request=request;
    }

    public String execute() throws Exception {
        HttpSession session = request.getSession();
        User user=(User)session.getAttribute("User");
        if(user!=null){
            if(user.isAdministrator())
                return SUCCESS;
            else{
                this.addActionError("You are not authorised for this action.");
                return LOGIN;
            }
        }
        else{
    }
```

```

        this.addActionError("You must be Logged in.");
        return LOGIN;
    }
}
public String addEmployee(){
    //Implement code for adding new Employee Here.
    this.addActionMessage("Employee Added Successfully.");
    return SUCCESS;
}
}

```

Similarly, we have created another simple LogoutAction action class, which is used for just invalidating the session. Here's the code, given in Listing 12.21, for LogoutAction action class (you can find LogoutAction.java file in Code\Chapter 12\Struts2Security\WEB-INF\src\com\kogent\struts\action folder in CD):

Listing 12.21: LogoutAction.java

```

package com.kogent.struts.action;
import javax.servlet.http.*;

import com.kogent.struts.security.SecurityService;
import com.kogent.struts.security.SecurityServiceImpl;
import com.kogent.struts.security.User;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletRequestAware;

public class LogoutAction extends ActionSupport implements ServletRequestAware{

    private HttpServletRequest request;
    public void setServletRequest(HttpServletRequest request) {
        this.request=request;
    }
    public String execute() throws Exception {
        HttpSession session = request.getSession();
        session.invalidate();
        return SUCCESS;
    }
}

```

Here's the code, given in Listing 12.22, showing how we can create our web.xml file without any container-managed security implementation (you can find this web.xml file in Code\Chapter 12\Struts2Security\WEB-INF folder in CD):

Listing 12.22: web.xml without any CMS implementation

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>Struts 2 Security-AMS</display-name>
    <filter>

```

```
<filter-name>struts2</filter-name>
<filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
</filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Here's the code, given in Listing 12.23, for the struts.xml file with all action mappings used in this application (you can find struts.xml file in Code\Chapter 12\Struts2Security\WEB-INF\classes folder in CD):

Listing 12.23: struts.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="default" extends="struts-default">

        <action name="loginAction" class="com.kogent.struts.action.LoginAction">
            <result name="success">/index.jsp</result>
            <result name="login">/login.jsp</result>
            <result name="input">/login.jsp</result>
        </action>
        <action name="addEmployeeAction"
            class="com.kogent.struts.action.AddEmployeeAction">
            <result name="success">/addemployee.jsp</result>
            <result name="login">/index.jsp</result>
        </action>
        <action name="addEmployee"
            class="com.kogent.struts.action.AddEmployeeAction" method="addEmployee">
            <result name="success">/index.jsp</result>
        </action>
        <action name="logoutAction" class="com.kogent.struts.action.LogoutAction">
            <result name="success">/index.jsp</result>
        </action>
    </package>
</struts>
```

Observe different result pages configured for different action executions. Let's run this application implementing application-managed security in our Struts 2 based Web application. The application shows index.jsp page as the first page when it runs and its output is shown in Figure 12.1.



Figure 12.1: The index.jsp with hyperlinks to add employee and login.

Here, if you try to add a new employee without logging in, you see an error message intimating that you have not been logged in, as shown in Figure 12.2.



Figure 12.2: Message intimating that the user is not logged in.

The two users implemented through the `SecurityServiceImpl` class, shown in Listing 12.14, for their username, password, and assigned roles. You can login by clicking over 'Login' hyperlink shown in Figure 12.1, which brings you an output screen shown in Figure 12.3.



Figure 12.3: The login.jsp page showing a login form.

You can enter by using a valid username and password as coded in the Listing 12.14. The valid combinations to be used here are santosh/kogent and prakash/kogent. The user 'prakash' has a single role, i.e. CLERK_ROLE, while the user 'santosh' has two roles, i.e. ADMIN_ROLE and CLERK_ROLE. You can see the error messages on login.jsp, if an invalid username and password is entered. Let's login using prakash/kogent as username/password combination to see the index.jsp now with a different output, as shown in Figure 12.4.



Figure 12.4: The index.jsp page intimating successful login.

The user prakash does not have admin role and, hence, any request to access the console to add a new employee must be restricted here. This has been implemented in our AddActionEmployee action class.

The AddActionEmployee action class looks for the User object set as a session attribute and, if found, checks whether the user is having ADMIN_ROLE by invoking the `isAdministrator()` method. Click on the ‘Add Employee’ hyperlink to get another error message showing that the user is not authenticated for this action, as shown in Figure 12.5.



Figure 12.5: Message intimating user for its non authorization

Click on ‘Logout’ option and re-login as another user having ADMIN_ROLE and add a new employee. Now we have implemented application-managed security in our Struts 2 based application using a security service and its implementation.

Use of Servlet Filters for Security

Filters can be applied to static HTML pages, JSP pages, Struts actions – essentially any resource that can be specified with a URL. Generally what does a filter do? A filter alters a request before it arrives at its destination and, similarly, modifies the response after it leaves a destination. You can use a filter to implement role-based access controls. The filter performs the same work like that of the RequestProcessor in Struts request processing. It determines whether a user is allowed access to a given Web resource. It checks whether the user is authenticated and has one of the required roles. If either of these checks fail, the filter stores an appropriate error message in the request and forwards the request to a URL. Some initialization parameters are required to specify authorization as well as the page to forward to if an error occurs. These parameters enable the creation of filter classes that can be easily reused. The authorization filter that is used inside our application is given in Listing 12.24.

Listing 12.24: AuthorizationFilter.java

```
package com.kogent.struts.security;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.Globals;
import org.apache.struts.action.*;
```

```
public class AuthorizationFilter implements Filter {
    private String[] roleNames;
    private String onErrorUrl;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        String roles = filterConfig.getInitParameter("roles");
        if (roles == null || "".equals(roles)) {
            roleNames = new String[0];
        }
        else {
            roles.trim();
            roleNames = roles.split("\\s*,\\s*");
        }
        onErrorUrl = filterConfig.getInitParameter("onError");
        if (onErrorUrl == null || "".equals(onErrorUrl)) {
            onErrorUrl = "/index.jsp";
        }
    }

    public void doFilter(ServletRequest request,
        ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;
        HttpSession session = req.getSession();
        User user = (User) session.getAttribute("user");
        ActionErrors errors = new ActionErrors();
        if (user == null) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("error.authentication.required"));
        } else {
            boolean hasRole = false;
            for (int i=0; i<roleNames.length; i++) {
                if (user.hasRole(roleNames[i])) {
                    hasRole = true;
                    break;
                }
            }
            if (!hasRole) {
                errors.add(ActionErrors.GLOBAL_ERROR,
                    new ActionError("error.authorization.required"));
            }
        }
        if (errors.isEmpty()) {
            chain.doFilter(request, response);
        }
        else {
            req.setAttribute(Globals.ERROR_KEY, errors);
            req.getRequestDispatcher(onErrorUrl).forward(req, res);
        }
    }

    public void destroy() { }
}
```

In Listing 12.24, the AuthorizationFilter class implements Filter. Thus, it must implement the `init()`, `doFilter()`, and `destroy()` methods. The comma-separated list of roles and the URL of the error page to forward to are retrieved from the initialization parameters in the `init()` method. The work of the `doFilter()` method is to check if there is a User in the session. If not, an appropriate `ActionError` is created and no further checks are performed. Otherwise, it iterates through the list of roles to determine if the user has any of them. If not, an `ActionError` is created. If any errors were created, then a `RequestDispatcher` is created to forward to the given URL; otherwise, the `doFilter()` method calls the `chain.doFilter()` to continue normal processing. When you are approaching this type of a security mechanism, you need to configure the filters, like that of a Servlet. Inside the `web.xml` file, specify the filter name and class, and the initialization parameters.

Here's Listing 12.25 showing the filter configuration:

Listing 12.25: `web.xml`

```

<filter>
  <filter-name>adminAccessFilter</filter-name>
  <filter-class>
    com.kogent.struts.security.AuthorizationFilter
  </filter-class>
  <init-param>
    <param-name>roles</param-name>
    <param-value>administrator</param-value>
  </init-param>
  <init-param>
    <param-name>onError</param-name>
    <param-value>/index.jsp</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>adminAccessFilter</filter-name>
  <url-pattern>/administrator/*</url-pattern>
</filter-mapping>

```

In this way, we can implement Servlet filters to implement security in our Web application. The `AuthorizationFilter` is used here to implement security. This filter class is configured in `web.xml` to filter out all request url matching `url-pattern /administrator/*` here. The two init parameters have been set here, which are used by our `AuthorizationFilter` filter class.

In this chapter, we discussed the various types of Security levels and their implementation in Struts application along with the two different approaches in the form of container-managed security and application-managed security. In addition to this, the authorization and authentication techniques, different configuration details were also discussed in the chapter.

The next chapter will focus on testing Struts 2 application using JUnit.



13

Testing the Struts 2 Application

If you need an immediate solution to:

Testing a Struts 2 Action Class

See page:

497

In Depth

After developing application, it should always be tested and checked to ensure that it functions correctly. Testing is the one of the important phases of System Development Life Cycle. Testing phase reveals the bugs that may have been committed during the development and may further cause problems when the software is run. These days most of the organizations use testing tools to test the accuracy of programmed code. The organizations can use different testing tools, while developing applications. With the help of testing, you can find drawbacks in your project before you begin implementing them. The following two approaches used to perform testing are as follows:

- Top-down approach
- Bottom-up approach

In the bottom-up approach, smaller components are tested first and then the bigger components. On the other hand, in the top-down approach, major components are tested first followed by the smaller components. The progression and definitions of completed tests vary between organizations. Struts testing is same as Web application testing. First separate modules are tested and then they are brought together to implement testing on the complete system. However, before we delve into testing Struts 2 application, we are going to discuss unit testing, and different testing tools available for testing like JUnit.

Understanding Unit Testing

Unit testing can be defined as testing the smallest possible units of an application separately. In other words, unit testing is based on the independent testing of individual software components, which is performed by the developer. Usually, in a Java based application, a unit can be a simple Java class, but the extent of unit to be tested depends on the extent of a set of logical function of the class. The reason in support of unit testing before integration is that it is far easier and cheaper to identify and correct problems in isolation. The testing of small separate units is always easy in comparison to testing a large application. Unit testing can be of three types:

- Code logic testing**—The traditional view of unit testing
- Integration unit testing**—All units are tested for their proper communication with other units
- Functional unit testing**—All units are verified for its output for known inputs

For different business components and numerous services in a Struts application there may be bulk of unit tests which are required. You cannot run a test code without a runtime environment. Also, it is desirable to automate the running of unit tests. For doing so the unit tests need to meet a certain criteria, e.g. a successful test should not need manual checking and a failed test should deliver adequate documentation for diagnosis. We can use any of the available unit testing frameworks, which provides different APIs to be used for unit testing.

Unit Testing Frameworks

Each testing framework provides functionalities, like testing the code, providing output, and reporting the errors to the users who are testing a particular code. These functions are common in frameworks used for testing. All that the test developer needs to write is just the test code. The two common unit testing frameworks are as follows:

- JUnit
- Cactus

JUnit Testing Framework

JUnit is a framework, which provides the functionality of creating and performing unit testing on different Java classes available in the code. JUnit comes with a base test class, which is extended by programmers to perform a particular test. The JUnit uses both the Text and the GUI mode to represent the test result of User Interface (UI).

NOTE

JUnit can be downloaded from <http://www.junit.org>.

To create and run the test cases, you need to add `junit.jar` and `src.jar` in your classpath of your application.

Here's Listing 13.1 showing the basic JUnit `run()` method:

Listing 13.1: Simple JUnit `run()` method

```
public void run()
{
    setUp();
    runTest();
    tearDown();
}
```

There are a number of ways in which one can test whether a proper result is produced. The `TestCase` class extends the `junit.framework.Assert` class and implements the `junit.framework.Test` interface. JUnit provides many features to simplify the test of application in the `junit.framework.TestCase` class. These features, which have been implemented as methods of `junit.framework.Assert` class, are as follows:

- `assertEquals`
- `assertFalse`
- `assertNotNull`
- `assertNotSame`
- `assertNull`
- `assertSame`
- `assertTrue`
- `fail`

Using JUnit

Let's test some code with JUnit. First you'll need a Java class to test. You can look at the following example describing the sample test. Let's assume that we have a class named Calculation, which is to be tested here.

Here's the Calculation class, given in Listing 13.2, with methods like `int add(int, int)` and `int subtract(int, int)`:

Listing 13.2: Calculation.java

```
package example;
public class Calculation
{
    static public int add(int a, int b) {
        return a + b;
    }

    static public int subtract(int a, int b) {
        return a - b;
    }
}
```

The two methods of Calculation class take two integer types of arguments to perform addition and subtraction operations. To test a given class, we need to create a test case which is again implemented as a simple Java class extending `junit.framework.TestCase` as base class.

Here's the test class, given in Listing 13.3, for the Calculation class created as `CalculationTest.java`:

Listing 13.3: CalculationTest.java

```
package example;
import junit.framework.TestCase;
public class CalculationTest extends TestCase {

    public void testSubtract() {
        int number1 = 10;
        int number2 = 2;
        int total = 8;
        int minus = 0;
        minus = Calculation.subtract(number1, number2);
        assertEquals(minus, total);
    }
}
```

When designing a test class we can create different `testMethodName()` methods, which signify that the `methodName()` method is supposed to be tested for its functionality. For example, Listing 13.3 shows a test class with the `testSubtract()` method that is created here to test the functionality of `Calculation.subtract()` method.

You can follow the given steps to create this basic application with Eclipse and JUnit:

Chapter 13: Testing the Struts 2 Application

Create a simple core Java project named CoreTest by clicking File | New | Project menu options.

Similarly create an example.Calculation class with the code provided in Listing 13.2 (Figure 13.1).

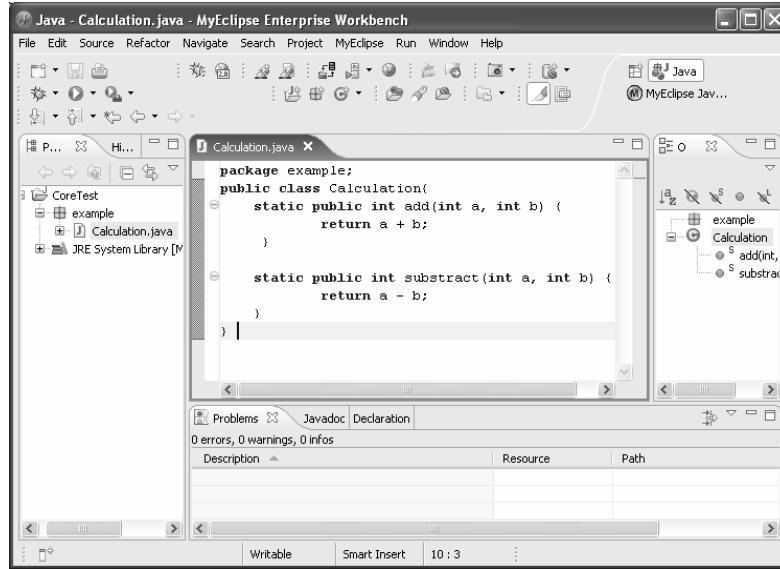


Figure 13.1: A sample CoreTest application.

Add the JUnit library. You can configure the Build Path by right-clicking on JRE System Library icon in the package explorer (the left most pane), as shown in Figure 13.2.

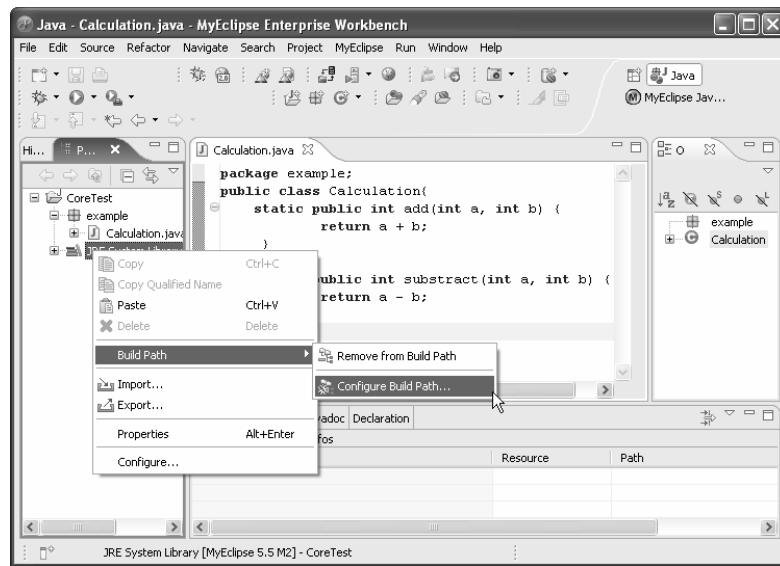


Figure 13.2: Configuring Build Path.

Click over the ‘Library’ tab and then click on ‘Add Library’ button to display the Add Library dialog box, as shown in Figure 13.3.

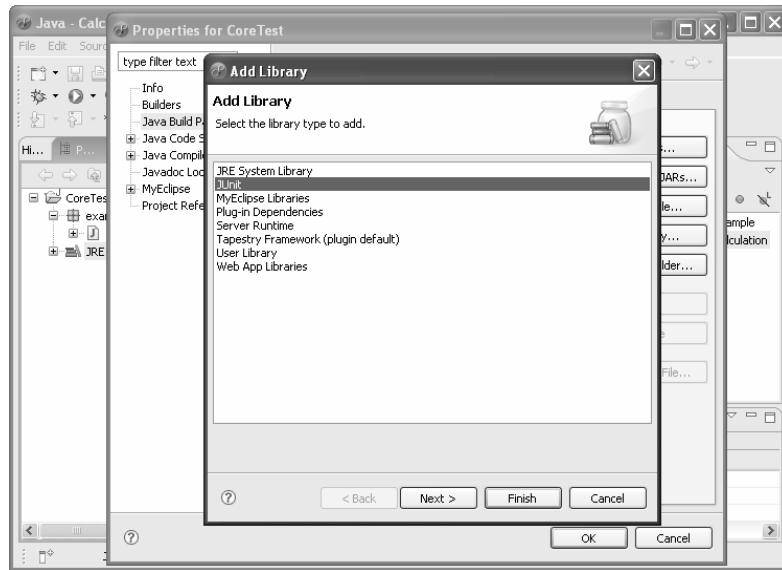


Figure 13.3: Adding JUnit Library.

Select ‘JUnit’ library from the dialog box and click on ‘Next’ button.

Select the 3.8.1 version and click on Finish button followed by clicking on OK button to complete the addition of JUnit library.

Click File | New | Other...option which opens a New dialog box shown in Figure 13.4.

Create a test class by selecting ‘JUnit Test Case’ option from the ‘Select a wizard’ screen, as shown in Figure 13.4, and click on ‘Next’ button.

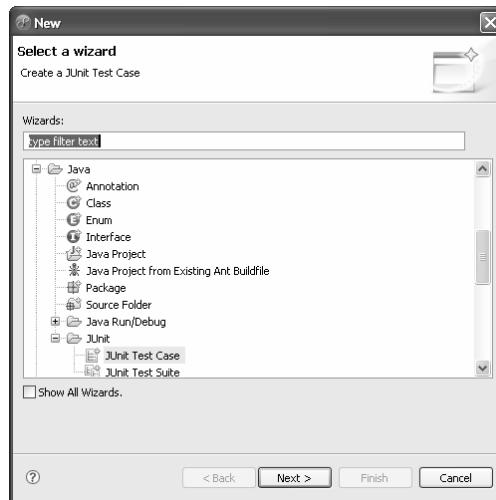


Figure 13.4: Creating JUnit Test Case

Chapter 13: Testing the Struts 2 Application

In the ‘JUnit Test Case’ screen, add a package name in the ‘Package’ text box, say example, class name as CalculationTest of the test class being created, and fill the ‘Class under test’ field with the name of the class being tested (Figure 13.5)

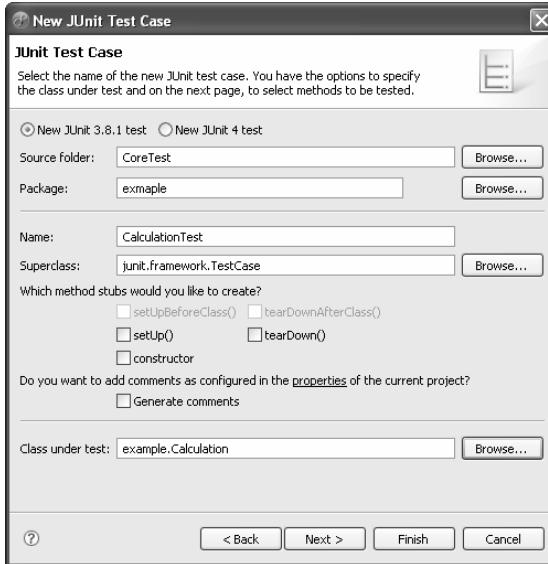


Figure 13.5: Creating CalculationTest class

Finally click on ‘Next’ button.

In the ‘Test Methods’ screen, select the method of the Calculation class to be tested for its functionality, as shown in Figure 13.6. The method selected here is `substract()`.

After selecting the method, click over ‘Finish’ button.

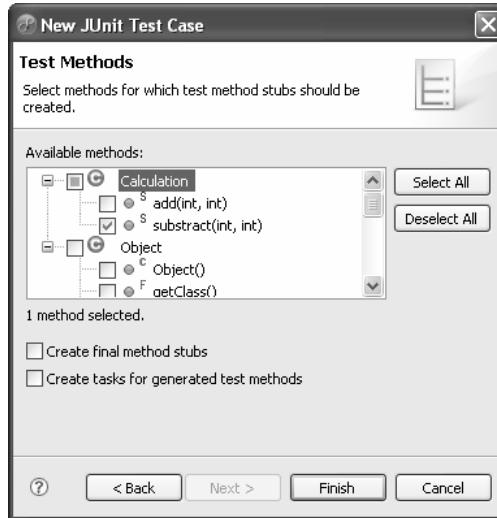


Figure 13.6: Selecting Class method to test

Add code lines in CalculationTest.java file, according to Listing 13.3 and save the file. See Figure 13.7 for the Calculation and CalculationTest class files created.

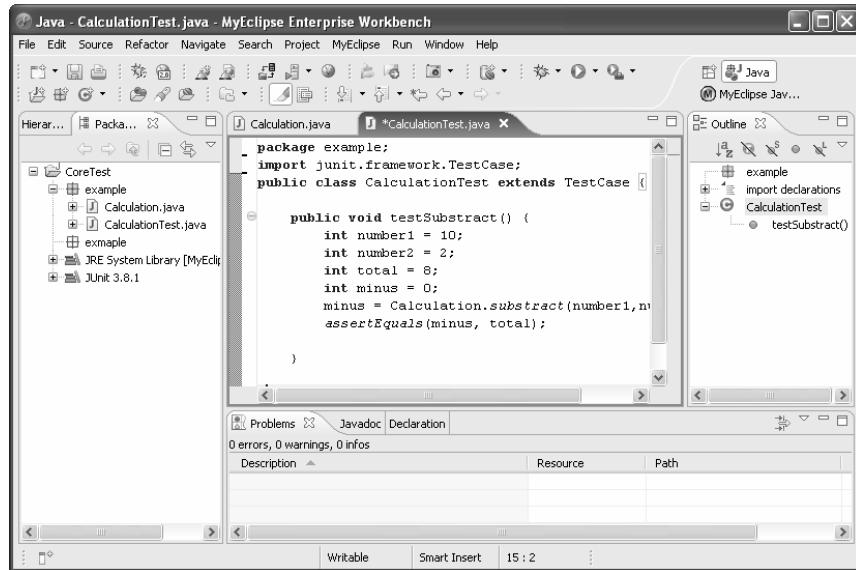


Figure 13.7: The CoreTest application with two class files.

Now you are ready with your test class to test the `subtract()` method of `Calculation` class. You can run the `CalculationTest` class by selecting `Run | Run As | JUnit Test` menu option to display the output shown in Figure 13.8.

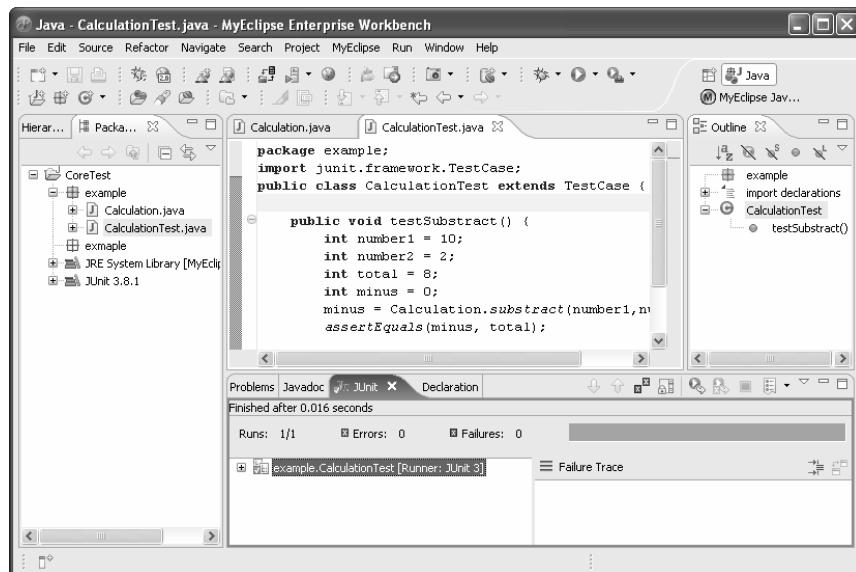


Figure 13.8: Running CalculationTest class

Observe the JUnit tab shown in Figure 13.8. The color of the bar shows whether the test case was successful or failed. The red colored bar indicates failure whereas the green colored bar shows success. In addition, you can see other details, like number of ‘Runs’, ‘Errors’, and ‘Failures’.

Cactus Testing Framework

The Cactus Framework is used to solve the problem created by *mock* object. Cactus uses an in-container strategy to simplify this problem. It extends the JUnit Framework to the web container. Cactus Framework has the following two parts:

- ❑ Traditional JUnit Framework
- ❑ The in-container part

Traditional framework has additional `WebRequest`, which is used to specify the request parameters and add HTTP headers. The in-container part helps you to connect the `WebRequest` with the `HttpServletRequest`. This `WebRequest` is user-defined. Join point is also provided for interacting with the `TestCase`. The join point is used to add request attributes and session attributes.

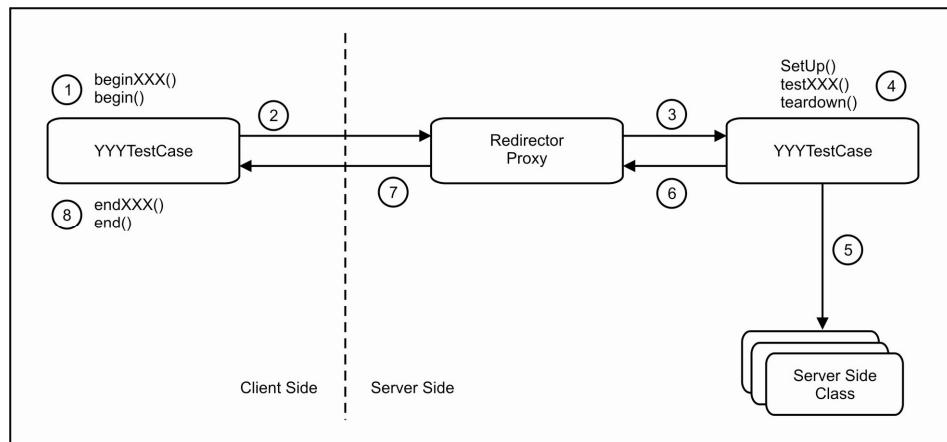


Figure 13.9: Traditional Cactus TestCase

Cactus Framework comes with several `TestCase` classes that are used for extending the `TestCase` provided by JUnit. This framework also comes with several kinds of redirectors, like `ServletRedirector`, `JSP Redirector`, etc. for redirecting the Servlets and JSP pages. Figure 13.9 is the general one. Here, `YYYTestCase` can be a `ServletTestCase`, `FilterTestCase`, and `JspTestCase`. `xxx` is the name of the test case. Each `YYYTestCase` class contains many test cases.

Let's now understand the working of `CactusTestCase`. The following steps give you information on how testing is performed:

1. The JUnit Test Runner invokes a `YYYTestCase.runTest()` method. This method first searches for a `begin (WebRequest)` method and executes it, if found. Note that this method is called before each test. In Figure 13.9, HTTP parameters are set by the `WebRequest` parameter, which was passed to `beginXXX ()` method. These parameters will be used by the `Redirector proxy`.

After setting the HTTP parameters, the `YYYTestCase.runTest()` method releases an HTTP connection to the `Redirector proxy`. HTTP request will start receiving the parameters of the `beginXXX()` method.

TestCase class uses the Redirector proxy, which act as proxy on the server side. Test case used is instantiated two times, first on the client side using the JUnit, and secondly on the on server side using the Redirector proxy. The client side instance is used to execute the begin (), beginXXX(), endXXX(), and end() methods. The server side instance is used to execute testXXX() methods. The Redirector proxy performs the following tasks:

- ❑ It creates an instance of your test class using reflection
- ❑ It creates instances of Cactus wrappers from some server objectHttpServletRequest, servletconfig.ServletContext
- ❑ It creates an HTTP session, if the user has expressed the wish to use the WebRequest.setAutomaticSession(Boolean) code in the beginXXX() method

The setup (), testXXX(), and teardown() methods of your test class are executed in the order specified in the diagram. They are called by the Redirector proxy using reflection. The setUp() and tearDown() methods are optional.

Your testXXX () method calls your server side code to test. It executes the test and uses the JUnit assert API to assert the result using assert (), assertEquals (), or fail () method.

Exceptions are thrown by the testXXX () methods and the same are caught by Redirector proxy, if the test fails. After catching the exceptions the Redirector proxy sends the exception information back to the clients.

If an exception occurs, Redirector proxy returns the information about the exception back to the client side.

If no exception occurs, then YYYTestCase.runTest () method searches for an endXXX (org.apache.cactus.WebResponse) method and executes it, if found. This end () method is called after each test.

Redirector is actually a Servlet that is used for unit testing Servlet methods or any associated Java class that works with Servlet object. Now let's go through an example to know how Cactus works. Take a simple Servlet class with a test case.

This is our simple Servlet class where we'll print two buttons on the page depending on the request attribute. If the request will be 'Hi' then a button with name button1 is created on the page, else a button with name button2.

Here's the code, given in Listing 13.4, for ButtonServlet.java:

Listing 13.4: ButtonServlet.java

```
package unittest.cactus;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ButtonServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        String attribute = req.getParameter("type");
        PrintWriter pwt = res.getWriter();
        res.setContentType("text/html");
        if(attribute.equals("Hi"))
            pwt.println("<input type='button' value='button1' />");
        else
            pwt.println("<input type='button' value='button2' />");
    }
}
```

```
pwt.print("<html><head/><body>");  
pwt.print("<form name='index'>");  
if (attribute.equals("Hi")) {  
    pwt.print("<input type=button name='button1' value='Click' />");  
} else {  
    pwt.print("<input type=button name='button2' value='Click' />");  
}  
pwt.print("</form>");  
pwt.print("</body></html>");  
}  
}
```

Here's Listing 13.5 showing how the test class to test `ButtonServlet` class can be created by extending the `org.apache.cactus.ServletTestCase` class:

Listing 13.5: MyServletTest.java

```
package unittest.cactus;  
import org.apache.cactus.ServletTestCase;  
import org.apache.cactus.WebRequest;  
import com.meterware.httpunit.Button;  
import com.meterware.httpunit.HTMLElement;  
import com.meterware.httpunit.WebResponse;  
public class ButtonServletTest extends ServletTestCase {  
    public ButtonServletTest(String testname) {  
        super(testname);  
    }  
    protected void setup() throws Exception {  
        super.setUp();  
    }  
    protected void tearDown() throws Exception {  
        super.tearDown();  
    }  
    // prepare http request parameters  
    // set type parameter to Hi  
    public void beginDoGet1(WebRequest req) {  
        req.addParameter("type", "Hi");  
    }  
  
    // test case 1 for doGet() method  
    // in ButtonServlet  
    public void testDoGet1(){  
        ButtonServlet servlet = new ButtonServlet ();  
        try {  
            servlet.doGet(request, response);  
        } catch (Exception e) {  
            fail("exception: " + e);  
        }  
    }  
  
    // compare the result  
    public void endDoGet1(WebResponse res) {  
        HTMLElement[] element = null;  
        try {  
            element = res.getElementsWithName("button1");  
        }
```

```

        assertEquals(1, element.length);
        assertFalse(((Button)element[0]).isEnabled());
    } catch (Exception e) {
        fail("exception: " + e);
    }
}

// prepare http request parameters
// set type parameter to Bye
public void beginDoGet2(webRequest req) {
    req.addParameter("type", "Bye");
}

// test case 2 for doGet() method
// in ButtonServlet
public void testDoGet2(){
    ButtonServlet servlet=new ButtonServlet ();
    try{
        servlet.doGet(request, response);
    } catch (Exception e) {
        fail("exception: " + e);
    }
}

// compare the result
public void endDoGet2(webResponse res){
    HTMLElement[] element = null;
    try {
        element = res.getElementsWithName("button1");
        assertEquals(1, element.length);
        assertTrue(((Button)element[0]).isEnabled());
    } catch (Exception e) {
        fail("exception: " + e);
    }
}
}

```

Using Cactus for Integration Testing

Unit tests are isolated from outside dependencies, so it is much easier to identify the cause of errors, when a test fails. However, you can perform integrated unit testing. Integrated testing is useful for some types of objects, which can only be tested in Web container. Container-provided services, such as transactions and persistence are not easily mocked up. Therefore here, we use integrated testing. Integrated testing is also useful with regression testing. An integrated unit test is required for checking the deployment of different application servers and operating systems. Cactus was developed to provide such a type of integrated unit testing. Configuration and deployment of Cactus tests are complex rather than writing them. To perform the cactus based testing, you need to download a Cactus 1.5 from the <http://jakarta.apache.org/cactus/downloads.html>.

Here's Listing 13.6 that defines the task definitions for the ANT tasks that Cactus provides:

Listing 13.6: Task definition for ANT tasks

```
<path id="cactus.classpath">  
  <fileset dir="cactus/lib">  
    <include name="*.jar"/>
```

```
</fileset>
</path>
<taskdef resource="cactus.tasks">
```

This defines a path containing the Cactus .jar files and creates the Cactus task definitions. Cactus is an extension of JUnit and provides base classes that extend `junit.framework.TestCase`. Cactus runs on both the client and the server. Cactus tests run in two JVMs—one copy on the client JVM, and another on the server JVM.

In this section, we covered the types of unit testing framework and how simple Java and web-based application using JUnit and Cactus are tested. Let's move forward to the “Immediate Solutions” section, to discuss Struts 2 Testing with JUnit, i.e. how to test a Struts 2-based application using JUnit.

Immediate Solutions

Struts 2 provides certain plug-ins for Junit testing and makes testing the Struts 2 application using JUnit very easy. With Struts 2 and Junit there is no need to add mock objects for testing as was required, previously, with Struts 1 application.

Testing a Struts 2 Action Class

Here we'll discuss a simple application built using Struts 2 Framework. The application consists of one JSP page, and an action class, configuration files, and tag libraries. The basic view components designed here is `hello.jsp`. The basic logic implemented in this application is to print a message on the user interface (`hello.jsp`). This message is generated by the execution of action class (`HelloAction`). The message is printed on the JSP page according the result code returned from the `execute()` method of action class.

But to make all these work together in the framework, all components are developed and configured according to what the framework supports. We'll go through the code files for all these components before getting a look at how these are configured in `web.xml` and `struts.xml`.

Creating an Struts 2 Application

In this application the view component, as mentioned earlier, for this application is `hello.jsp`. The `hello.jsp` simply prints a message from the value stack by invoking the `getMessage()` method. In addition, it provides the hyperlink to invoke the `HelloAction` action class.

Here's the code, given in Listing 13.7, for `hello.jsp` (you can find `hello.jsp` file in `Code\Chapter 13\HelloTest` folder in CD):

Listing 13.7: `hello.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>struts2 Testing</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <h2><s:property value="message" /></h2>
        <s:a href="helloAction.action">Click to Run Action</s:a>
    </body>
</html>
```

The Action class in Struts 2 application is `HelloAction`, which contains an input properties message with getter and setter methods for this property. In addition, the `execute()` method of `HelloAction` class just sets the message and returns 'SUCCESS' as result code.

Here's the code, given in Listing 13.8, for `HelloAction` class (you can find `HelloAction.java` file in `Code\Chapter 13\HelloTest\WEB-INF\src\example` folder in CD):

Listing 13.8: `HelloAction.java`

```
package example;
import com.opensymphony.xwork2.ActionSupport;
public class HelloAction extends ActionSupport {

    public static final String MESSAGE = "Successfully Tested ...";

    public String execute() throws Exception {
        setMessage(MESSAGE);
        return SUCCESS;
    }

    private String message;

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

    public String getMessage() {
        return message;
    }
}
```

To configure this application, the two configuration files used are `web.xml` and `struts.xml` files. The `web.xml` file is set similar to the Deployment Descriptors of other Struts 2 applications.

Here's the code, given in Listing 13.9, describing the structure of `web.xml` (you can find `web.xml` file in `Code\Chapter 13\HelloTest\WEB-INF` folder in CD):

Listing 13.9: `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>Testing Struts 2</display-name>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

```
<welcome-file-list>
    <welcome-file>hello.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

The Struts configuration file, struts.xml, contains a single action mapping provided here with the name helloAction that executes example.HelloAction class. The 'SUCCESS' returned as result code brings the hello.jsp page again.

Here's the code, given in Listing 13.10, for the action mapping provided in struts.xml file (you can find struts.xml file in Code\Chapter 13\HelloTest\WEB-INF\classes folder in CD):

Listing 13.10: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <package name="example" extends="struts-default">
        <action name="helloAction" class="example.HelloAction">
            <result>/hello.jsp</result>
        </action></package>
    </struts>
```

To create this application in Eclipse IDE, you can use the following steps:

1. Create a simple Web Project named HelloTest by clicking File | New | Project menu options.
2. Similarly create an example.HelloAction class with the code provided in Listing 13.8.
3. Create struts.xml, web.xml file, and hello.jsp according to the code described in their listings.

Figure 13.10 shows our HelloTest application being developed using Eclipse IDE.

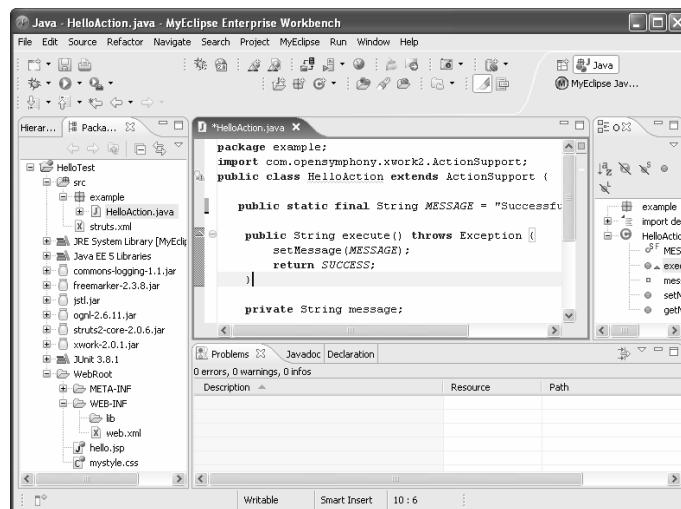


Figure 13.10: HelloTest - A Struts 2 Application

NOTE

You can see Figure 13.10 for the different components that are created in HelloTest application. All components have been described with their associative listings in this chapter. Remember to add Struts 2 JAR files in the lib folder of your Web application created here. Make it sure that all components are at the same location as shown in Figure 13.10.

Creating Test Case

There are two things we can test in this application. First is to test whether the string being set in message property of our action class is same as that of the value set for final property MESSAGE. And, second one, is to check the action configuration of the struts.xml. Here the test class name is HelloActionTest and it is shown in Listing 13.11 (you can find HelloActionTest.java file in Code\Chapter 13\HelloTest\WEB-INF\src\example folder in CD):

Listing 13.11: HelloActionTest.java

```
package example;

import com.opensymphony.xwork2.ActionSupport;
import junit.framework.TestCase;

public class HelloActionTest extends TestCase {

    public void testExecute(){
        HelloAction hellotest = new HelloAction();
        String result = null;
        try {
            result = hellotest.execute();
        } catch (Exception e) {
            e.printStackTrace();
        }

        assertTrue("Expected a success result!",
                   ActionSupport.SUCCESS.equals(result));
        assertTrue("Expected the default message!",
                   HelloAction.MESSAGE.equals(hellotest.getMessage()));
    }
}
```

NOTE

You can follow the steps here which are discussed for CoreTest application in the “In Depth” section for adding the JUnit library, creating a test case for a given class, and specifying a specific method of the class to be tested. .

Now, you are ready to run HelloActionTest class to test your HelloAction action class. Similar to our CoreTest application discussed in the “In Depth” section, you can execute HelloActionTest to see the output, as shown in Figure 13.11. Figure 13.11 displays Error and Failures occurred during the execution of HelloActionTest class in its JUnit tab.

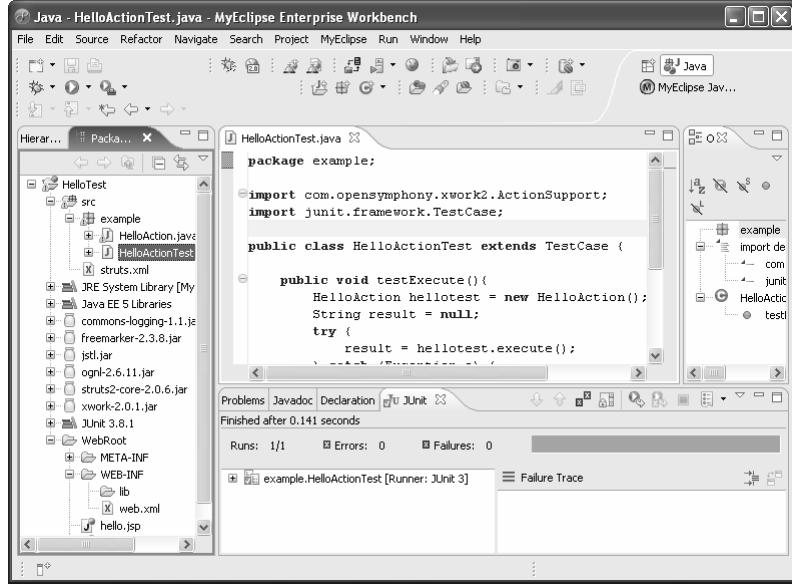


Figure 13.11: Testing HelloAction with JUnit.

This chapter described unit testing and its types and introduced various unit testing frameworks, like JUnit, Cactus with simple examples of test cases in each case. Finally, it provided a simple struts application and test cases to test it.

With this, we come to the end of our discussion on different concepts, tools and APIs of Struts 2 Framework. We have provided different appendices to cover various complimentary topics. We have provided separate appendices to discuss implementation of Tiles plugin and type conversion supported in Struts 2. Other appendices will introduce you to AJAX, Freemarker, and Velocity, and their integration with Struts 2 Framework that has just revolutionized the highly interactive dynamic Web application development.



A

AJAX and Struts 2

With the increased use of Internet, enhanced browser capabilities and the broadband connection available, the power of browser is changing to real-time dynamic user interactivity from simple form-based processing. Introduction of Asynchronous JavaScript And XML (AJAX) has brought an evolution in the field of Web Application Development by shifting the web page based Web Application Development to the Data Centric Application Development. Before the advent of AJAX, retrieving any data from the server required the whole page to be refreshed in the user's computer, at the client side. As a result, systems were often designed with less interaction. For example, when the user submits a form after filling the form with information, it takes a quite long time to validate different fields and return the response to the user. In contrast, AJAX systems can validate one or two items at a time behind the scenes without making the session an awkward condition. It is the scenario in which the requested data can be fetched from the server behind the scene asynchronously and can be inserted to the web page dynamically and thus eliminate the need to reload the entire web page. Hence, its implementation increases the interactivity and speed of the Web application, which is now comparable to the desktop applications. It basically works by transferring a part of application's processing onto the client's machine so that the to and fro traversal to the server is not required for reloading the entire page for all the user interactions (based on when only a small segment of the web page is to be updated, then why to reload the whole page again?). AJAX development architecture is different from the earlier client-server programming. As AJAX depends on JavaScript and XML in the browser, it gives cross-browser compatibility.

An HTML page can make asynchronous calls to the server by using JavaScript and can retrieve XML documents. JavaScript can use this XML document to modify Document Object Model (DOM) of the HTML page. This interaction model is implemented Asynchronous JavaScript and XML (AJAX). So the two important features of AJAX implementation are as follows:

- To make the request to the server without reloading the whole page
- Parse and work with XML document

Defining AJAX

AJAX is not a technology; rather it is a collection of technologies. AJAX can be defined as a methodology for developing a highly interactive Web application using existing technologies, like JavaScript, XML, XSLT, XHTML, CSS, and DOM, etc. AJAX can also be defined using these technologies and their credential to this new interactive model as follows:

Appendix: A

- ❑ XHTML and CSS for standards-based presentation
- ❑ Document Object Model for dynamic display and interaction
- ❑ XML and XSLT for data interchange and manipulation
- ❑ JavaScript for binding everything together

Traditionally, in a Web application, the user simply sends an HTTP request to the server. The Server returns an HTTP response to the user after some processing and interacting with the database (if required), as shown in Figure A.1.

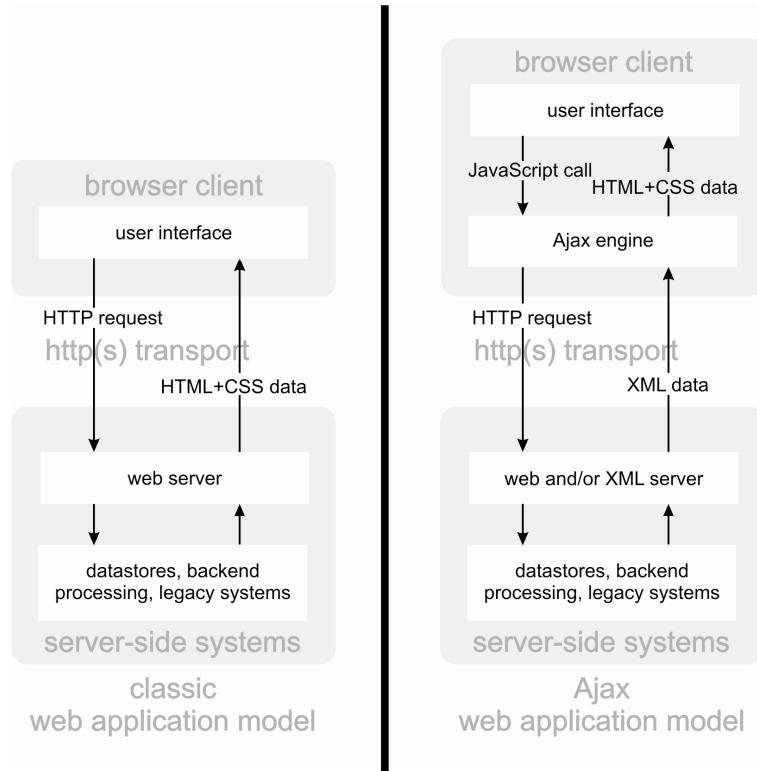
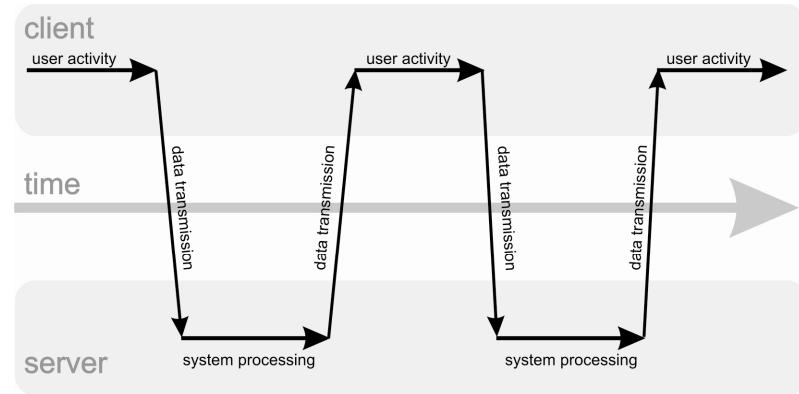


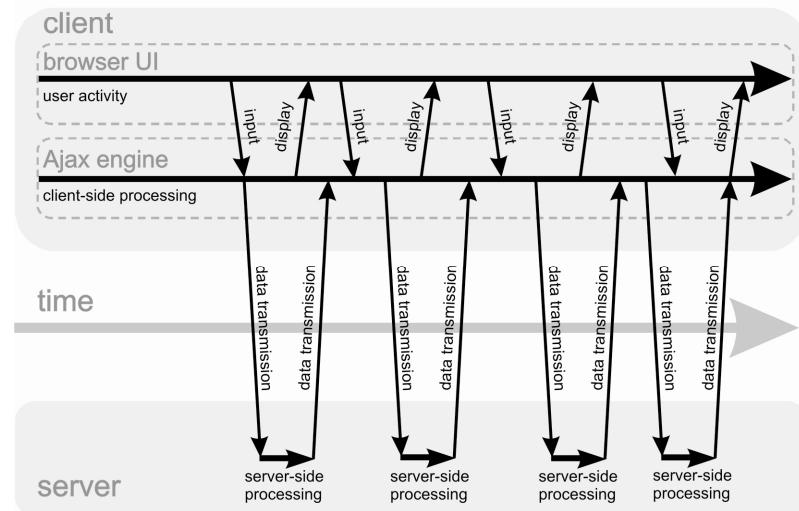
Figure A.1: Classic model and AJAX model for Web applications.

This model is good enough until we talk about the response time, which is of great importance in case of software applications. The problem: When the server is doing its processing, what is the user doing? Waiting! Of course. For every new interaction, the user has to wait for a longer time. The user interaction is interrupted each time the application needs something from the server as the whole page is to be reloaded.

In an AJAX application, an intermediate AJAX Engine is introduced between the Client (User) and the Server, as shown in Figure A.1. This layer increases the responsiveness of the application. The browser loads the AJAX engine, which is written in JavaScript, instead of loading a Web page with new session. This engine is responsible for two things—first for having communication with the server on behalf of the user and second for rendering the user interface. Now the user can interact with the application asynchronously. This user-application interaction is independent of the communication with the server. So the blank browser windows and hourglass icon may become the past.



Ajax web application (asynchronous)

**Figure A.2: Synchronous interaction pattern (traditional web application) and Asynchronous interaction pattern (AJAX Web application).**

Every HTTP request by the client is converted into a JavaScript call to the AJAX engine, as shown in Figure A.2. All user actions which don't need to communicate with the server is handled by the engine, like validation of data, editing data, some changes in the presentation of a little segment of the page, etc. Sometimes the engine needs to communicate with the server for the actions, like submitting data, fetching new interface code, or new data. This communication between the engine and server is asynchronous, uses XML and does not halt the user's interaction with the application.

Tools and Technologies in AJAX Ecosystem

As it has been defined earlier that AJAX is a combination of different technologies working together. Hence, we can say that AJAX has generated a new ecosystem of various technologies, including development tools, programming languages, etc. These technologies and tools are as follows:

Appendix: A

- JavaScript
- XML
- XMLHttpRequest
- CSS
- Server Side Scripting

JavaScript

JavaScript is the most important aspect in any AJAX application. It binds together various parts of the application. JavaScript is used for all client-side processing of information, which is fetched from the server. All the logic is in the form of JavaScript code that is on the client machine and responsible for the communication with the server.

XML

Development of an AJAX application totally depends upon the use of XML. All data, which is transferred from client to server or vice-versa, is in XML format. The XMLHttpRequest object is used to send data over HTTP. Some other XML related technologies, like Extensible Style sheet Language Transformation (XSLT), is used to produce HTML or XML from XML data.

XMLHttpRequest Object

Microsoft Internet Explorer 5.0 introduced the XMLHttpRequest object as an ActiveX object. XMLHttpRequest object is implemented in most of the browsers. This object helps in transferring data, while the user is interacting with the Web page. In an AJAX application, XMLHttpRequest is used in most of the communications. It helps in loading the new content without being changed. It also allows synchronous calls to be made from the JavaScript. Some important methods associated with this object are as follows:

- XMLHttpRequest: open()
- XMLHttpRequest: send()
- XMLHttpRequest: setRequestHeader()
- XMLHttpRequest: getResponseHeader()

XMLHttpRequest: open() Method

This method is used to set the request type using the following three arguments,

- The method name (Get or Post),
- The URL of the requested page
- The true or false for call being asynchronous or not.

Here's their simple usage:

```
var request = new XMLHttpRequest();
request.open('Get', 'index.jsp', true);
```

XMLHttpRequest: send() Method

This method creates the connection to the URL, which is specified in open. If the call is asynchronous, the return will be immediate, else it will wait until the page is downloaded. Here's their simple usage:

```
request.send(null);
OR
request.send('hello=world&XMLHttpRequest=test');
```

XMLHttpRequest: setRequestHeader() Methods

Sometimes, setting header on the request is useful, e.g. setting content type. It takes two arguments, i.e. header and its value. Here's their simple usage:

```
request.setRequestHeader('Content-type',
'application/x-www-form-urlencoded;charset=UTF-8');
```

XMLHttpRequest: getResponseHeader() Method

This method allows us to get a single response header from the response. Here's their simple usage:

```
var request = new XMLHttpRequest();
request.open('Get', 'index.jsp', false);
request.send(null);
if (request.status==200){
alert(request.getResponseHeader('Content-type'));
alert(request.getAllResponseHeader());
}
```

CSS

Cascading Style Sheets are used in an AJAX application to change the style and design of the application. It gives presentation and rendering information, which is used by the browser to display the page in a standard manner.

Server Side Scripting

Any standard processing technology can be used for the sever side scripting. This can be JSP, Servlets, PHP, and ASP. The data is transmitted as an XML message.

AJAX implementation has reduced the communication bandwidth required, as only the related and required data moves between the client and server, instead of reloading the whole web page. The load on the server is also reduced in the sense that the user interface logic is implemented on the client machine. It needs great planning when developing an AJAX application as different types of technologies need to interact.

Using AJAX with Struts 2

In this section we will discuss the use of the AJAX inside the Struts 2 applications. While developing applications that are related to Struts 2, you might have developed some parts of the application, which may contain two or more dependable HTML select boxes. This means, suppose you have made two dependable select boxes, then the value of the second select box will always depend on the value selected in the first select box. To achieve this type of operation you can follow the following three ways:

- ❑ You can perform all these operations simply with the help of JavaScript arrays in which each set of options links to one of the options that are present in the first select-box.
- ❑ Another option for this operation can be obtained with the help of an event `OnChange` on the first select box so that it can automatically submit the form and go back to the server. At the server-side newly suggested options can be collected and the page can be generated again.
- ❑ The third way to get this operation implemented is using AJAX, which has the power to retrieve the new options for the second select box asynchronously. This way is always preferred the over other two ways said above.

However, it can also be possible using the `<s:doubleselect>` tag, which is present in Struts 2 Tag library under UI tags. In case of the first option, you need to implement a variety of logics inside the HTML page, which is a little complicated. In addition, it would have been visible to the user and using JavaScript is somewhat difficult to debug also. These are some of the reasons for which the first option is not treated as a suitable one.

Now let's arrive at the second option. Though this option is better than the first option, but it is not one of the reliable processes to achieve the desired operation. One of the disadvantages in the implementation part of the second option is that you have to schedule your journey a complete a round trip on the server to gather the new options. In addition, the entire page needs to be refreshed. So this type of operation is also quite time consuming, though it is able to perform the operations.

Now we are left with only the third option, i.e. using AJAX. Developers prefer to implement AJAX to solve the operations related to the *dependency of select boxes* because of the asynchronous nature of AJAX. Though Struts 2 comes with some AJAX tags, it does not have much AJAX support yet; you have to add it yourself whenever you require. This is because AJAX is not considered as a framework, but just a technique to make web pages more dynamic with the help of the technologies, like JavaScript, DHTML, etc.; it is quite easy to use it in combination with Struts 2, which leads the developers to prefer this technique while developing applications containing this type of operation. Using AJAX you can update the desired part of a web page and don't need to change the whole page. This is because, when AJAX is implemented inside any program or application, it interacts with a program through *asynchronous calls* only.

AJAX Tags in Struts 2

The Struts 2 Framework provides a set of AJAX tags in order to achieve the benefits of AJAX inside Struts. This is not enough; to *ajaxify* your application first of all you have to set the `theme` attribute to Ajax. So let's discuss about Ajax themes first.

The ajax Theme

The ajax theme extends the xhtml theme with AJAX features. The xhtml theme is one of the default themes used in the WebWork. It requires the use of the DOJO AJAX/JavaScript toolkit. The theme uses the following AJAX features:

- ❑ AJAX client-side validation, which is obtained with the combination of both JavaScript and DOM manipulation.
- ❑ Remote form submission support.
- ❑ It provides a template known as `div`, which provides a dynamic reloading of partial HTML.
- ❑ It provides a template known as a template, which loads and evaluates JavaScript remotely.
- ❑ It provides the `tabbedpanel` widgets using which, the page can be refreshed each time the user selects the tab, i.e. an AJAX-only `tabbedpanel` implementation.
- ❑ It provides a rich event model.
- ❑ Interactive auto complete tag.

Importance of Head Tag

Head tag is generally used for the configuration of page for the AJAX themes. If your page is having AJAX component and the default theme has not been set to ajax, we have to use head tag to set the theme to ajax. Consequently, a typical AJAX header setup is included in the page. Here's the code that shows how a theme can be set using head tag:

```
<Head>
<title>my page</title>
<s:head theme="ajax" calendarcss="calendar-green"/>
</head>
```

Common AJAX Tags Attribute

All Ajax tags have some attributes which can be set for different values to customize their behavior. All the attributes that are used commonly in all ajaxian tags are given in the Table A.1.

Table A.1: Common AJAX tags attributes		
Attribute	Type	Description
<code>href</code>	String	It is an option to mention the 'url', which is used to make the request
<code>listenTopics</code>	String	This attribute lets the tag to reload its content (like, Div, Autocompleter), or perform an action (like, Anchor, Submit). A comma provided in between <code>listenTopics</code> , separates the list of topic name
<code>notifyTopics</code>	String	It is set with a comma separated list of topic names published by the tag
<code>showErrorTransportText</code>	Boolean	It tells whether an error message should be displayed; default value is <code>true</code>
<code>indicator</code>	String	It is the id of an element that will be displayed while a request is in progress

Here's some more description about the following attributes:

Appendix: A

- ❑ href—The href must contain a URL that is built with the url tag like the one shown in the following example:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />
<s:div theme="ajax" href="#">Initial Content
</s:div>
```

Here, it is clear that the AJAX tags will not work, unless their URL is not built with url tags.

- ❑ Topics—Topics provide an easy way to listen and publish events. To listen a topic, you should use the following syntax:

```
dojo.event.topic.subscribe("/refresh", function(param1, param2) {
    //---
    //---
});
```

This function will be called every time when the /refresh operation is published. And when you want to publish a topic write the following:

```
dojo.event.topic.publish("/refresh", "foo", "bar");
```

- ❑ notifyTopics—The notifyTopics has three parameters—data, type, and request. The last one has a cancel property, which is used to prevent a request. The following example shows the use of the request parameter inside notifyTopics:

```
<script type="text/javascript" language="javascript">
dojo.event.topic.subscribe("/request", function(data, type, request) {
//cancel request
request.cancel= true;
});
</script>

<s:url
id="ajaxTest"
value="/AjaxTest.action" />

<s:submit
type="submit"
theme="ajax"
value="submit"
notifyTopics="/request"
href="#">"/{ajaxTest}"/>
```

- ❑ Indicator—Indicator lets the user know that a request is in progress. An example of indicator is an image. The indicator should be hidden at the time of loading.

The Table A.2 describes the basic Ajax tags. They will all be described in brief later.

Table A.2: Ajax tags	
Basic Ajax tag	Description
<s:div>	Its duty is to create an area, which can load the contents through Ajax, optionally refreshing.
<s:submit>	Its work is to update element(s) as well as submits a form via Ajax.
<s:a>	Its responsibility is to update element(s) via Ajax.
<s:tabbedPanel>	It generates a tabbed panel, which contains static or dynamic <s:div.../> tab contents.
<s:autocompleter>	Its responsibility is to provide proper suggestions or update elements based on their current value.

The **div** Tag

The div tag is a content area that can load its content asynchronously. It can force to reload its content using topics. To define the topics that will trigger the refresh of the panel, you have to implement the listenTopics attribute. The following example shows how the div tag is used:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />
<s:div theme="ajax" href="#">%{ajaxTest}" listenTopics="/refresh0,/refresh1"/>
```

In this example the div will refresh every time the topics /refresh0 or /refresh1 are published.

The div tag can also be configured to update its contents in a time-to-time manner by using a timer. This property can be achieved by using the updateFreq attribute. This can set the interval for the timer and its value is always expressed in milliseconds. If autoStart, in this case, is set to true, the delay attribute can be used to cause the timer to wait for some moments that is mentioned in the delay period before starting. In this case the required coding will be like the following snippet:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />
<s:div theme="ajax" href="#">%{ajaxTest}" updateFreq="2000" delay="3000"/>
```

The autoStart attribute handles the functionality of timer, that means it indicates whether the timer will be started when the page is loaded. By default, this is set to be true. Generally, the functionality timer, like starting and stopping, can be achieved using the topics startTimerListenTopics and stopTimerListenTopics, respectively. An example using these two attributes is as follows:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />
<s:div
    theme="ajax"
    href="#">%{ajaxTest}"
```

Appendix: A

```
startTimerListenTopics="/startTimer"
stopTimerListenTopics="/stopTimer"
updateFreq="3000"
autoStart="false"/>
```

If the loaded content contains JavaScript code sections, then this will be executed if and only if the `executeScripts` attribute is set to true. When the parameters need to be passed to the URL, the role of the `formId` comes into picture. This `formId` attribute is generally used to identify a form whose fields are serialized and passed in the request as parameters. The element `formFilter` can be set to the name of a JavaScript function. Its responsibility is to filter the fields that are present in `formId`. The `formFilter` function will be called for each field for the purpose of filtration. The field's DOM node is passed as a parameter and the filter method returns true/false according to the corresponding operation. That is, if the field is to be included it returns true, otherwise it returns false. An example of implementing this type of tag is as follows:

```
<script type="text/javascript">
function filter(field) {
    return field.name == "firstName";
}
</script>

<form id="userData">
<label for="firstName">First Name</label>
<input type="textbox" id="firstName" name="firstName">
<label for="lastName">Last Name</label>
<input type="textbox" id="lastName" name="lastName">
</form>

<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:div href="#">"/{ajaxTest}"
```

`theme="ajax" formId="userData" formFilter="filter"/>`

In the preceding example, `<s:div>` tag can be used to submit the value of the field 'firstName' only and the rest of the fields of form `userData` are ignored. When the attribute handler needs to be implemented, the JavaScript function, specified by its value, will be called instead of making the request. This handler function will be implemented when the user wants to handle the request itself. The following example gives the idea regarding the use of the handler function:

```
<script type="text/javascript">
function handler(widget, node) {
    alert('I will handle this myself!');
    node.innerHTML = "Done";
}
</script>

<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:div theme="ajax" href="#">"/{ajaxTest}"
```

`handler="handler"/>`

All the attributes discussed are given in Table A.3.

Table A.3: The div tag attributes		
Attribute	Type	Description
handler	String	It is set with the JavaScript function's name that will make the request
formId	String	It is set with the form id of the form whose fields will be serialized and passed as parameters
formFilter	String	It is set with the function's name used to filter the fields of the form
loadingText	String	It is set with the text to display in the div while content is loaded
errorText	String	It is set with the text to display in the div when there is an error
refreshListenTopic	String	It is set with the Topic's name that will cause the div content to be reloaded
startTimerListenTopics	String	It is set with the comma separated Topic names that will start the timer
stopTimerListenTopics	String	It is set with the Topic names that will stop the timer
executeScripts	Boolean	It can be set to true to enable execution of all JavaScript code in the loaded content
updateFreq	Integer	It is set with the time between requests (in milliseconds)
delay	Integer	It is set with the time to wait before making the first request (in milliseconds)
autoStart	Boolean	It can be set to true to start timer automatically when the page loads

The **submit** Tag

The submit tag is generally used to update the contents of its targets attribute with text returned from the asynchronous request. The targets attribute is optional. Here's an example that uses a regular submit button that will update the contents of div1:

```
<div id="div1">Div 1</div>
<s:url id="ajaxTest" value="/AjaxTest.action" />
<s:submit
type="submit"
```

Appendix: A

```
theme="ajax"
value="submit"
targets="div1"
href="#">"%{ajaxTest}"/>
```

We can generate the submit button using an image by defining its `src` property in the previous syntax. Here's the syntax:

```
div id="div1">Div 1</div>

<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:submit
type="image"
theme="ajax"
label="Alt Text"
targets="div1"
src="#">"\${pageContext.request.contextPath}/images/struts-rocks.gif"
href="#">"%{ajaxTest}"/>
```

All the attributes present inside the `submit` tag, along with their working criteria, are given in Table A.4. These are very much similar to those of `div` tag.

Table A.4: The submit tag attributes

Attribute	Type	Description
targets	String	It is comma-delimited list of ids of the elements whose contents will be updated
handler	String	It is JavaScript function's name that will make the request
formId	String	It is a Form id of the form whose fields will be serialized and passed as parameters
formFilter	String	It is function's name used to filter the fields of the form
loadingText	String	It is set with the text to be displayed in the targets while content is loaded
errorText	String	It is set with the text to display in the targets when there is an error
refreshListenTopic	String	It is the topic name that will cause the targets content to be reloaded
executeScripts	Boolean	It can be set to true to enable execution of all JavaScript code in the loaded content
src	String	It is set with the image source for image type submit button.

The **anchor** Tag

The working principle of the anchor tag is similar to that of the submit tag. The following example shows the use of the anchor tag:

```
<div id="div1">Div 1</div>
<div id="div2">Div 2</div>

<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:a theme="ajax" href="#">" targets="dev1,dev2">Update divs</s:a>
```

In this example, the anchor will update the contents of div1 and div2 with text returned from the /AjaxTest.action. The anchor tag can also be used to submit a form by writing the following snippet :

```
<s:form id="form1">
<input type="textbox" name="data">
</s:form>

<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:a theme="ajax" href="#">" formId="form1">Submit form</s:a>
```

The **tabbedPanel** Tag

The tabbedPanel tag can implement both static and dynamic tabs. Here, in this case, every div tag present inside the tabbedPanel is treated as a tab. The label attribute is required for each tab and this attribute is treated as its caption. The following example describes the use of the tabbedPanel tag:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:tabbedPanel>
<s:div label="static">
    This is an static content tab.
</s:div>
<s:div href="#">" theme="ajax" label="dynamic">
    This is a dynamic content tab.
    The content of this div will be replaced with the text returned
    from "/AjaxTest.action"
</s:div>
</s:tabbedPanel>
```

In this example; the tabbedPanel has one static tab as well as one dynamic tab. In case of a dynamic tab, the content of the div will be replaced by text that is returned from /AjaxTest.action.

Inside the tabbedPanel tag, you can use the property labelposition to specify the place where to place the tab labels. The possible values used inside it are top, bottom, right, and left. The Tab also has a 'Close' button, which has the capacity to remove the tab from its tabbedPanel. By default, the 'Close' button does not appear. To get it visible the closeButton property of the panel can be introduced.

Appendix: A

The following snippet shows the operations related to the ‘Close’ button. This means, it gives a detail about how the tabbedPanel will place the tab labels on the left with a ‘Close’ button on each tab; by default, its value is selected as dynamic:

```
<s:url id="ajaxTest" value="/AjaxTest.action" />

<s:tabbedPanel labelposition="left" closeButton="tab" selectedTab="dynamic">
    <s:div label="static" id="static">
        his is an static content tab.
    </s:div>
    <s:div href="#">%{ajaxTest}" theme="ajax" label="dynamic" id="dynamic">
        his is a dynamic content tab.
        he content of this div will be replaced with the text returned
        rom "/AjaxTest.action"
    </s:div>
</s:tabbedPanel>
```

All the available attributes inside the tabbedPanel along with their working functionality is given in Table A.5.

Table A.5: The tabbedPanel Tag attributes		
Attribute	Type	Description
closeButton	String	It gives details about the place where the ‘Close’ button will be placed. The possible values are tab and panel
selectedTab	String	It gives the information related to the id of the tab that will be selected by default.
doLayout	Boolean	It gives the idea about the tabbedPanel’s height. If doLayout is set to be false (default), the tabbedPanel’s height will be as much as the currently selected tab
labelposition	String	It gives the details regarding where to place the tabs; possible values of Labelposition are top (default), right, bottom, and left

*The **autocomplete** Tag*

The autocomplete tag is one of the best tags used in the programming. The main functionality of the autocomplete tag is to load the options that are provided before, asynchronously when the page loads and suggest options, based on the text entered in the textbox. The autoComplete attribute of autocomplete tag can be set to true or false to enable auto complete. The autocomplete tag always displays a dropdown list with some matching options, which have at least a partial match with the text entered in the textbox so that when the user gives some message by typing some text in the text box, the autocomplete makes a suggestion in the textbox. When the user clicks on the dropdown list, all the options related to the text entered will be shown in the drop-down box. To make this property available to the user, you have to set the attribute forceValidOption to true.

Here's an example snippet for autocomplete tag without its autoComplete attribute set to false:

```
<s:url id="json" value="/JSONList.action" />
<s:autocomplete theme="ajax" href="#">%{json}
```

This tag can also be used in the *simple* theme, without Ajax functionality. In this case, you can use the value as well as the list attribute to specify the available options. The following code gives better clarification:

```
<s:autocomplete theme="simple" list="{'apple','banana','grape','pear'}"
" value="grape"/>
```

Here, in this example, you can find the value `grape`, by default, and the values that are inside the list in the drop-down list.

Generally, the default size of the drop-down list is 120 px. When the box contains few items and is smaller than 120 px, the height will be adjusted to the size of the value. To make changes in the default value of the drop-down's height, you can take the help of the `dropdownHeight` attribute. The following example defines an autocomplete with a drop-down box having height of 180 px:

```
<s:url id="json" value="/JSONList.action" />
<s:autocomplete theme="ajax" href="#">%{json} dropdownHeight="180"/>
```

When the topic specified in the `refreshListenTopic` attribute is published, the option autocomplete will be reloaded. If the `onValueChangedPublishTopic` attribute is provided with a value, then, whenever the selected value changes, a topic with only that name will be published. These two attributes can be used to link two autocompleters. The following example shows the implementation of two autocompleters:

```
<s:url id="json" value="/JSONList.action" />
<form id="selectForm">
  Autocompleter 1
  <s:autocomplete theme="simple" name="select" list="{'fruits','colors'}"
  "notifyTopics="/Refresh" />
</form>
  Autocompleter 2
  <s:autocomplete theme="ajax" href="#">%{json} formId="selectForm"
  listenTopics="/Refresh"/>
```

Here, when the selected value of the `autocomplete1` changes in `autocomplete1`, the `/refresh` is published, and forces `autocomplete2` to reload its options by submitting the form `selectForm`.

If the user wants to show the options in the drop-down box after typing some letter, you can take the help of `loadMinimumCount`. An example of using this attribute is as follows:

```
<s:url id="json" value="/JSONList.action" />
<s:autocomplete theme="ajax" href="#">%{jsonList}" loadOnTextChange="true"
  loadMinimumCount="4" showDownArrow="false"/>
```

Appendix: A

Here, in this example, the autocomplete reloads its contents everytime the user types inside the text box. But the autocomplete will activate its properties, if and only if the typed-text's length is four, not before that. All the attributes that are used in the autocomplete along with their working principles are given in Table A.6.

Table A.6: The autocomplete tag attributes		
Attribute	Type	Description
autoComplete	Boolean	This attribute specifies whether to make suggestions for auto complete in the text box or not. Suggestions in the drop-down will still be made, irrespective of the value set for this attribute.
forceValidOption	Boolean	It specifies whether the text entered has to match an option or not. Otherwise, the value will be cleared when autocomplete loses focus
delay	Integer	It specifies the time to wait before making the search (in milliseconds)
searchType	String	It specifies how to match entered text against the available options, startstring (default), startword and substring
dropdownHeight	Integer	It sets the height of the dropdown (in pixels); default 120
dropdownWidth	Integer	It sets the width of the drop-down (in pixels); default same as autocomplete's width
formId	String	It is used to set the form id of the form whose fields will be serialized and passed as parameters
formFilter	String	It is set with function's name used to filter the fields of the form
value	String	It is used to set the default value when the theme is simple
list	String	It is used to specify Iteratable source to populate options
loadOnTextChange	Boolean	It specifies whether to reload options as user enters some key
loadMinimumCount	Integer	It is used to set the text length that will trigger a reload of the options if loadOnTextChange is set to true; default value is 3.
showDownArrow	Boolean	It specifies whether to show or hide the dropdown arrow; true by default

Libraries

There are a number of AJAX libraries, which are involved with Struts. There might be some warning for the implementation, but they somehow perform useful functions inside the Struts 2 application. Read on to understand some of the useful libraries.

AjaxParts Taglib

The AjaxPart Taglib, abbreviated as APT, is completely declarative like Struts. Unlike most other libraries, the APT is also Java-rich, because it uses custom taglib to do all its work. You can simply drop a tag onto a page to attach an AJAX event to an element on the page and can also define how that event works via XML config file. There is no requirement for writing JavaScript. APT is a powerful library with rich content of most common AJAX functions.

Java Web Part

JWP also offers some good functionalities. One of these functionalities is the JavaScript implementation of commons Digester. It provides some useful utility functions, such as `getPostBody()`, which returns the contents of the request's POST body as a string. Generally, it is a good option, while you are sending XML or JSON from a client for instance.

Prototype

Prototype is another useful JavaScript library, which helps in the development of dynamic Web applications.

Dojo

Dojo is also one of the useful JavaScript libraries, like prototype. Inside Dojo, you can find a number of GUI widgets utility classes to do client-side persistence storage, DOM manipulation functions, JavaScript collection implementations, as well as some solid Ajax functionalities.

Scriptaculous

This library is generally very useful when the user wants to add various effects to his pages. It also offers some handy JavaScript unit testing to your application.

Uses of AJAX Implementation

There are different scenarios where the AJAX implementation can be very useful, which may include auto completion of some input field, real time validation, refreshing a segment of the page, etc. Following are some of the uses of AJAX implementation:

- ❑ It can be used in the implementation of auto completion
- ❑ Some data, like name, city, and email, can be filled automatically as the user types.
- ❑ It supports Real-Time form validation
- ❑ Some Critical fields, like User Id, Postal Code that need server side validation can be validated before the form is submitted.
- ❑ It can be used with all typical User Interface Controls.
- ❑ The user interface controls, like menus and progress bars, can be designed that do not need page refreshing.
- ❑ It is useful in implementing Data Refreshing on the Page.
- ❑ The web page can fetch up-to-date data from the server, e.g. scores, weather report.

- ❑ It allows Server Side Notifications.
- ❑ It is for the notifications that notify the client with a message, refresh page data, or redirect to some different location.
- ❑ Detailed information can be fetched about the data from the server, which is based on the client event. For example, the click on the Menu can show submenu without the page refreshing.

Drawbacks

AJAX has increased the user interaction with the Web application. The AJAX supported applications can do a number of new things which they were not able to do earlier. But AJAX has its own drawbacks too which are as follows:

- ❑ Complexity: It needs great planning before designing an AJAX application. The developer needs to go through the presentation logic for HTML pages and the logic to generate XML content on the server side, which is needed by the client-side HTML page. Using a number of technologies together makes it somewhat complex. But, in future, it will become easier as there are strong possibilities that some exiting frameworks will come out for the rescue and support this interaction model.
- ❑ Debugging: As the logic is embedded on the both sides, i.e. on the client as well as on the server, the debugging is difficult.
- ❑ XMLHttpRequest: The XMLHttpRequest object is not a standard and not a part of JavaScript technology. So its behavior may change.
- ❑ JavaScript: The JavaScript must be activated on client browser to enable AJAX to work. The can be asked to enable JavaScript execution using browser options. This message must be provided to user using <noscript> tag, which is executed when <script> tag is not executed.
- ❑ Source Available: The code implemented using JavaScript on the client-side is visible to the client. This can be viewed using view source option of the browser. Hence, the application code and its logics can be hacked easily.

This introduction to AJAX will help you to develop dynamic Web applications, which can be compared with desktop applications in terms of the interactivity involved. Now we can develop a highly interactive Web application using rich AJAX tags and libraries available. Struts 2 Framework also fully supports AJAX which has further encouraged the use of Struts 2 for interactive Web application development.



B

FreeMarker and Velocity

FreeMarker is a Java-based template engine used to generate text outputs. It is also an alternative to JSP technology. A FreeMarker is not an application for the end-users in itself but it is something that the programmers can use in their projects. FreeMarker also supports the MVC architecture, and helps to separate the design from the programming.

Sometimes the information on a web page is not available for display to the user. This information is given to the user by FreeMarker. FreeMarker replaces the codes present in the HTML document with appropriate data at the time of displaying the page to the user.

Here's a simple example of FreeMarker template file:

```
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        Hello ${name}
    </body>
</html>
```

The instructions given in the code snippet are enclosed within the \${....}. These instructions are given to the FreeMarker to tell where it should replace the text at the time of sending the output to the client. The data to be inserted in the instructions given in the template file is available in the data model. The data model is just like a container where we can store data to be retrieved in future, if required. The data model is available in the memory of the computer and is accessed, whenever necessary. The data to the data model comes from the database or from the programs written by the programmers. The templates along with the data results into the actual output for the end user. The data is fetched here from the Data Model. Let's discuss the data model in detail.

Data Model

The data model is the temporary storage of data that we are going to access and use in the template file in FreeMarker. Data Model acts as a tree with some subnodes and some subvariables, and consists of sequences, hashes, and scalars. Figure B.1 demonstrates the hierarchy of the data model.

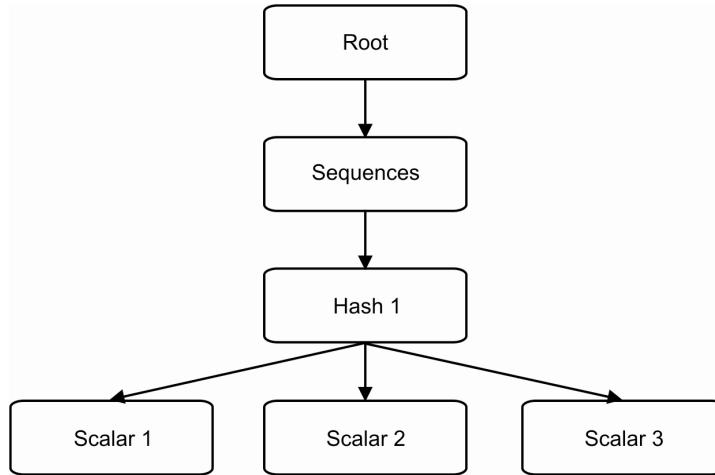


Figure B.1: Hierarchy of data model.

Let's understand sequences, hashes, and scalars.

Sequences

A sequence attaches a number with each variable it contains. The variables can be retrieved by the sequence number. The index of the first item always starts with '0' and so on. The variables act as a holder for the objects in sequence. It is similar to 'hash', but they do not store any name for the variable they contain. The only difference between hash and sequences is that the latter is used for ordering purposes, whereas the former do not have any ordering criteria.

Hash

Hashes are the containers for other variables and objects that are not displayed to the user. It attaches a lookup name to each variable it contains. The variables can be accessed by using the name attached to them in hashes (#).

Scalar

The variables that store a single value are called scalars. The scalar variables are of the following types – String, number, date and Boolean values.

- ❑ The string variables are the simple text variables. These values can be directly given in the template. These variables must be enclosed within double quotation (" ") marks.
- ❑ The number variables are the numerical values given to the variables. These can also be directly used in templates.
- ❑ The date type specifies the date/time representation of the variables. It can be represented as only the date precision, day or as the date-time precision. The time is always represented in terms of milliseconds.
- ❑ The Boolean value always represents the true/false value of an object or a variable. There are only two possible values – true and false. The Boolean values always work on the conditional statements.

For designing a template file, instead of these data model variables, some user-defined variables are also used. These include the use of user-defined methods, macros, and transforms. Methods are used to calculate a value, based upon a set of parameters. The parameters and output of the method is of the same type as that of the parameters passed to the method.

Macros are the template fragments associated with the variables. Macros itself do not print anything. It is used as a directive in the template file. We can create macro variables in a macro directive within a template file. Generally, it is used to perform repetitive tasks.

Template

Templates are programs written in a language called FreeMarker Template Language. These are the simple HTML files. These HTML files are sent to the client as the output. The template is made up of mainly following elements:

- ❑ **Text**—Text is used to print to the output as it is.
- ❑ **Interpolations**—Interpolations are represented by \${...}. These are simple texts that are included in the template file. These can be replaced with any calculated value.
- ❑ **FTL tags**—The FTL (FreeMarker Tag Language) tags are similar to HTML tags, but they are instructions to FreeMarker and will not be displayed to the output.
- ❑ **Comments**—Comments are similar to the HTML comments and are delimited by <!-- and -->. Anything written within these delimiters is ignored by FreeMarker.

Directives

The FTL tags are used to access directives. The directives are represented within start and end tags. Here we are using two types of directives—predefined directives and the user-defined directives. Both, the predefined and the user-defined directives, start and end with the start and end tags. The following syntax shows the representation of the directives:

```
Start-tag:<#directivename parameters>
End-tag:</#directivename>
Empty directive tag :< #directivename parameters/>
```

The predefined directives are defined by FreeMarker. These are always available for the FreeMarker components. Some examples of predefined directives are **if**, **else**, **include**, **import**, **global**, and **escape**. Some predefined directives are as follows:

If Directive

The **if**, **else**, **else if** directives are used to conditionally check a statement in a template file. The condition always evaluates to true or false. The **else** and **else if** directives must occur inside the start and end tags of the **if** directive. The following syntax shows the representation of **if** directives:

```
<if condition>
  ...
<elseif condition2>
  ...
<elseif condition3>
  ...
  ...
```

```
<#else>
...
</#if>
```

Include Directive

The `Include` directive is used to insert the contents of another file into the template. The output from the included file is inserted at the point where the `include` tag occurs. The `include` directive only processes the content of the included file. The following syntax shows the representation of `include` directive:

```
<#include filename>
or
<#include filename options>
```

Here, the `filename` is the name of the file which is to be inserted in the template. The option specified in the syntax can be `encoding` or `parse`. These are used for the expression evaluated to string and Boolean values, respectively.

Import Directive

The `import` attribute simply imports a library into the template. It creates an empty namespace and then executes the template with the specified path parameter and the template populates with the namespace variable, like macros, transforms, etc. Then the newly created namespace is made available to the caller by using the hash variables. The hash variable is a plain variable and is used by the `import` directive. The following syntax shows the representation of the `import` directive:

```
<#import path as hash>
```

Here, the `path` used in the syntax is the path of the template and is evaluated to a string. `Hash` is the hash variable and is used to access the namespace.

Global Directive

The variables created in this directive are visible to all the namespaces and they are not inside any particular namespace. We can create these variables within a data model. If a variable with the same name exists in the data model then the variable is replaced by the variable created with this directive. If a variable with the same name exists in the current namespace, then that will hide the variable created by the `global` directive. These variables can be accessed globally. The following syntax shows the representation of `global` directive:

```
<#global name=value>
Or
<#global name1=value1 name2=value2 ... nameN=valueN>
Or
<#global name>
    Capture this
</#global>
```

Here, the name is the name of the variable. The name variable can be written as a string literal and the value is the value assigned to the variables.

Escape Directive

When a template is marked with an escape directive, the interpolations that occur in the block are combined with escaping the expressions automatically. This avoids repetition of similar expressions. It does not affect the interpolations in string literals. The following syntax shows the representation of escape directive:

```
<#escape identifier as expression>
...
<#noescape>...</#noescape>
...
</#escape>
```

FreeMarker does not define the user-defined directives, rather they are defined by the user. These are the application domain directives. Macros and transforms are used as the user-defined directives in FreeMarker.

Expressions

An expression in FreeMarker illustrates the output after evaluation of the expression. The expressions can be of any type depending upon the type of value assigned in the variables. Some of the expressions are described here:

- ❑ **String operations**—String operations include the concatenation and getting of a character. The string concatenation is otherwise known as the interpolation of the characters. The \${....} or # {....} is used as the concatenation operators. Let's consider the following example:

```
 ${"Hello ${user}!"}
```

Suppose the user is a guest. So after the interpolation the expression will result ‘Hello Guest’ as the output. Similarly, we can access a single character from a string by using the index of the character. The index must be a number starting from 0 onwards. It can be accessed by using the \${....} delimiter with an index. Here \${user [0]} will return ‘G’.

- ❑ **Sequence operations**—The sequence operation includes two main features, such as concatenation and sequence slice. Concatenation operation is used to specify a list of items or names in a group. Let's look at the following example:

```
<#list ["winter", "spring"] + ["summer", "autumn"] as x>
${x}
</#list>
```

The preceding example prints ‘winter’, ‘spring’, ‘summer’, and ‘autumn’ as a list of values. Sequence slice is done by using startindex and the lastindex of any sequence. By this, we can access a slice of a sequence from a sequence. The statement seq[1..4] will access the sequence starting from index 1 to index 4 and is returned as the output.

- ❑ **Hash operations**—To use a hash in a template, list the key/value pairs separated by commas. The key and value is separated by a colon. Finally, put the list into curly brackets:

```
<#assign ages = {"John":23, "Smith":25}>
- John is ${ages.John}
- Smith is ${ages.Smith}
The output generated by this example is as follows:
- John is 23
- Smith is 25
```

- **Arithmetic operations** – The arithmetic operations are the simple arithmetic operations performed on a numeric data. The arithmetic operators, such as the +, -, *, /, are used to perform arithmetic operations.
- **Logical operations** – The logical operations use the logical operators to generate the output. The logical operators are the logical OR, logical AND, and logical NOT. These operations work on the Boolean type of data and return the TRUE/FALSE value to the user.
- **Built-in operations** – A built-in provides some information about the variable or formatted value of variable. The syntax for accessing a built-in is the same as accessing a subvariable in a hash, except a question mark instead of a dot. Some built-ins that we use in a string are as follows:
 - html – The string with all special HTML characters replaced with entity references
 - cap_first – This converts the initial character of a string into its upper case equivalent
 - lower_case – This is used to convert a string into its lower-case
 - upper_case – This is used to convert a string into its upper-case equivalent
 - trim – This built-in is used to return a string, which does not have any leading and trailing white spaces
 - size – The number of elements in a sequence. This is the only built-in used with the sequences
 - int – The integer part of a number. This is the only built-in used by the number in an operation.

Method call

A method can be called by using some syntax. Syntax is a comma-separated list of expressions in parentheses. These values separated by commas are called parameters. For example, assume that there is method display. It accepts a string as the first parameter, a number as the second parameter, and returns a string that displays the first parameter the number of times specified by the second parameter:

```
Example ${display("welcome", 3)}
```

The output of the preceding code is as follows:

```
welcome welcome welcome
```

Interpolations

You use interpolation to insert values into the output. There are two types of interpolations:

- **Universal interpolation** – \${exp}

□ **Numerical interpolation**—`(#{exp} or #{exp; format})`

The universal interpolation includes string, numerical, date, or Boolean values into the output, whereas the numeric interpolation only includes the numeric values. The advantage of Numerical interpolations over Universal interpolations is that we can specify a number formatting in the interpolation. Here's the syntax of the numerical insertion:

```
#{{exp; format}}
```

We can specify different format here. For example to specify that the length of decimal fraction is always exactly 1, we use `m1`. Similarly, to specify that the minimal length of decimal fraction is 1, but it can extend up to 3 digits, we can use `m1M2` as shown in following syntax:

```
#{{x; m1}} <!-- 2.6 -->
#{{x; m1M2}} <!-- 2.58 -->
```

A Simple FreeMarker Application

In this section, we'll take a closer look at a simple application using FreeMarker and Struts 2 Framework. This simple GuestBook application is used to store messages from different users. By using this application, a user can enter information about him and can write a message into the Guest Book. This application is based on Struts 2 Framework and uses FreeMarker template to create different output screens for the user. So let's start with View or Presentation layer first. The directory structure of GuestBook application is shown in Figure B.2.

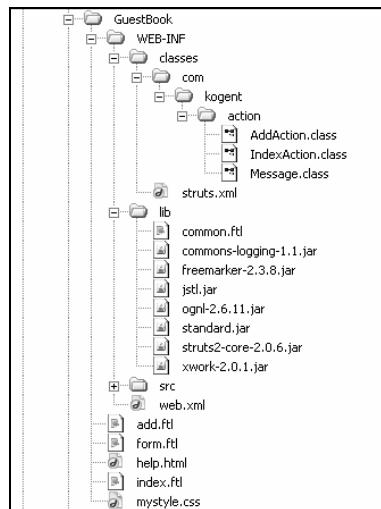


Figure B.2: Directory structure of GuestBook application

View Components

Here the view components contain four FreeMarker templates to represent the data—`common.ftl`, `index.ftl`, `form.ftl`, and `add.ftl`. Here's the code, given in Listing B.1, for the `common.ftl` file

Appendix: B

that is used as sample layout for other pages to be designed (you can find common.ftl file in Code\Appendix B\GuestBook\WEB-INF\lib folder in CD):

Listing B.1: common.ftl

```
<#macro page title>
<html>
<head>
    <title>Guest Book Example - ${title?html}</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h1>${title?html}</h1>
    <hr>
    <#nested>
    <hr>
</body>
</html>
</#macro>
```

Save common.ftl file in your application, say in WEB-INF/lib folder of your project folder GuestBook. The other pages have to import common.ftl file giving the path to its location. The first page with appears in the GuestBook application is a simple HTML page name help.html.

Here's the code, given Listing B.2, for help.html that provides a simple hyperlink to invoke a proper action (you can find help.html file in Code\Appendix B\GuestBook folder in CD):

Listing B.2: help.html

```
<html>
<head>
<title>Guest Book Example - Help</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
To try this example you should visit<br><br>
<a href="indexAction.action">http://yourserver:portnumber/projectname
/indexAction.action</a>
<hr>
</body>
</html>
```

The three more .ftl files to be created here are index.ftl, form.ftl, and add.ftl. The index.ftl works as the main page of the application and shows all the messages entered in the application.

Here's the code, given in Listing B.3, for index.ftl file (you can find index.ftl file in Code\Appendix B\GuestBook folder in CD):

Listing B.3: index.ftl

```
<#import "WEB-INF/lib/common.ftl" as com>
<@com.page title="Index">
<a href="showForm.action">Add new message</a>
<if messages?size = 0>
```

```

<p>No messages.
<#else>
<p>The messages are:
<table border=0 cellspacing=2 cellpadding=2 width="600">
<tr align=center valign=top>
<th bgcolor="#C0C0C0" width="300">Name
<th bgcolor="#C0C0C0" width="300">Message
<#list messages as e>
<tr align=left valign=top>
<td bgcolor="#E0E0E0">${e.name} <if e.email?length != 0> (a
href="mailto:${e.email}">${e.email}</a>)</if>
<td bgcolor="#E0E0E0">${e.message}
</#list>
</table>
</#if>
</@com.page>
```

An ArrayList stored in session scope is searched here and iterated over to display all messages. In addition, an hyperlink appears which takes to form.ftl page. The form.ftl page gives you a form with input fields to enter ‘Name’, ‘Email’ and ‘Message’ along with a ‘Submit’ button.

Here’s the code, given in Listing B.4, for form.ftl page (you can find form.ftl file in Code\Appendix B\GuestBook folder in CD):

Listing B.4: form.ftl

```

<#import "WEB-INF/lib/common.ftl" as com>
<#escape x as x?html>
<@com.page title="Add Entry">
<form action="addAction.action" method="post">
    Your name:<br>
    <input type=text name="name" size="30"><br><br>
    Your e-mail (optional):<br>
    <input type=text name="email" size="30"><br><br>
    Message:<br>
    <textarea name="message" rows="3" cols="30"></textarea><br>
    <input type="submit" value="Submit">
</form>
<a href="indexAction.action">Back to the index page</a>
</@com.page>
</#escape>
```

The last page to be designed here is add.ftl which simply displays the current message entered by the user. Here’s the code, given in Listing B.5, for add.ftl file (you can find add.ftl file in Code\Appendix B\GuestBook folder in CD):

Listing B.5: add.ftl

```

<#import "WEB-INF/lib/common.ftl" as com>
<#escape x as x?html>
<@com.page title="Entry added">
<p>You have added the following entry to the guestbook:
<p><b>Name:</b> ${name}
<p><b>Email:</b> ${email}
<p><b>Message:</b> ${message}
```

Appendix: B

```
<p><a href="indexAction.action">Back to the index page</a>
</@com.page>
</#escape>
```

Model Components

The Model component of the GuestBook application is a JavaBean class with the name Message. The Message class has three fields, i.e. ‘name’, ‘email’, and ‘message’ with public getter/setter methods for these fields.

Here’s the code, given in Listing B.6, for Message class (you can find Message.java file in Code\Appendix B\GuestBook\WEB-INF\src\com\kogent\action folder in CD):

Listing B.6: Message.java

```
package com.kogent.action;

public class Message {
    private String name;
    private String email;
    private String message;

    public String getEmail() {
        return email;
    }

    public String getMessage() {
        return message;
    }

    public String getName() {
        return name;
    }

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

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

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

Controller Components

The two action classes designed in this application are IndexAction and AddAction. Both of them are simple Struts 2 action class, which extends the ActionSupport class. They both implement the SessionAware interface to interact with the session scoped attributes.

Here's the code, given in Listing B.7, for `IndexAction` class (you can find `IndexAction.java` file in `Code\Appendix B\GuestBook\WEB-INF\src\com\kogent\action` folder in CD):

Listing B.7: `IndexAction.java`

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Map;

import org.apache.struts2.interceptor.SessionAware;

import com.opensymphony.xwork2.ActionSupport;

public class IndexAction extends ActionSupport implements SessionAware{

    Map session;
    public void setSession(Map session) {
        this.session=session;
    }
    public String execute() throws Exception {
        ArrayList messages=(ArrayList)session.get("messages");
        if(messages==null){
            messages=new ArrayList();
        }
        session.put("messages", messages );
        return SUCCESS;
    }
}
```

Another action class, i.e. `AddAction`, is used to add a new entered message details in the session scoped `ArrayList` object. The session scoped `ArrayList` object, which is set as session attribute with name `messages`, stores objects of `Message` class.

Here's the code, given in Listing B.8, for `AddAction` action class, which has three input fields – 'name', 'email', 'message' – with getter/setter methods in addition to the `execute()` method (you can find `AddAction.java` file in `Code\Appendix B\GuestBook\WEB-INF\src\com\kogent\action` folder in CD):

Listing B.8: `AddAction.java`

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.*;

public class AddAction extends ActionSupport implements SessionAware{

    private String name;
    private String email;
    private String message;
```

```
private Map session;
public void setSession(Map session) {
    this.session = session;
}

public Map getSession(){
    return session;
}
public String getEmail() {
    return email;
}
public String getMessage() {
    return message;
}
public String getName() {
    return name;
}
public void setEmail(String email){
    this.email = email;
}
public void setMessage(String message) {
    this.message = message;
}
public void setName(String name) {
    this.name = name;
}
public String execute()throws Exception{
    ArrayList messages=null;
    Message mess = new Message();
    mess.setName(name);
    mess.setEmail(email);
    mess.setMessage(message);
    messages=(ArrayList)session.get("messages");
    if(messages==null)
        messages=new ArrayList();
    messages.add(mess);
    session.put("messages", messages);
    return SUCCESS;
}
}
```

Configuring Application

The configuration done in `web.xml` and `struts.xml` file is shown in Listing B.9 and Listing B.10 respectively. The `web.xml` of this application is similar to other Struts 2 application created, except the `help.html` file is listed in welcome-file-list.

Here's Listing B.9 showing the configuration done in `web.xml` (you can find `web.xml` file in `Code\Appendix B\GuestBook\WEB-INF` folder in CD):

Listing B.9: `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
```

```

< xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<display-name>Guest Book</display-name>
<filter>
<filter-name>struts2</filter-name>
<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<welcome-file-list>
<welcome-file>help.html</welcome-file>
</welcome-file-list>
</web-app>

```

The configuration file, struts..xml file, binds various components of this application. Here's the code, given in Listing B.10, showing the action mapping provided in struts.xml file (you can find struts.xml file in Code\Appendix B\GuestBook\WEB-INF\classes folder in CD):

Listing B.10: struts.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
<package name="free-default" extends="struts-default">
<action name="addAction" class="com.kogent.action.AddAction">
<result name="success" type="freemarker">/add.ftl</result>
</action>
<action name="indexAction" class="com.kogent.action.IndexAction">
<result name="success"
type="freemarker">/index.ftl</result>
</action>
<action name="showForm">
<result type="freemarker">/form.ftl</result>
</action>
</package>
</struts>

```

There are three action mappings provided in struts.xml file. The first action mapped with name `addAction` invokes the `AddAction` action class and result code `success` from this action sends the user to `add.ftl` page. Similarly, the next action mapped with the name `indexAction` invokes the `IndexAction` action class, and finally sends the user to `index.ftl`. The last action `showForm` is just to show `form.ftl`. To access any `.ftl` file the invocation of freemarker result followed by proper interceptor execution is required and, hence, we need to use some action here.

With Struts 2 Framework, you can simply use FTL files instead of JSP files. You *must* copy the `.jar` files that are commonly required for Struts 2 applications into the `WEB-INF/lib` directory manually, before deploying the Web application including `freemarker-2.3.8.jar` file.

Appendix: B

Now to run the application start your server and access the application by entering following the URL, `http://localhost:8080/GuestBook/`, in the address bar of your browser to see the output, as shown in Figure B.3.

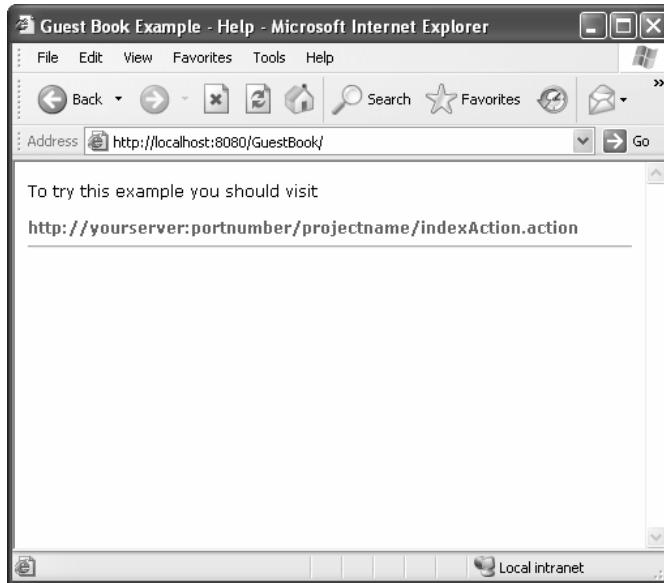


Figure B.3: The help.html showing a hyperlink.

You can click on the hyperlink shown in Figure B.3 to invoke the `IndexAction` class, which consequently sends you to `index.ftl` page. The `index.ftl` page is shown in Figure B.4.



Figure B.4: The index.ftl showing hyperlink to add new message.

Click on ‘Add new message’ hyperlink to see the output of `form.ftl` page, which gives input fields to fill up, as shown in Figure B.5.

The screenshot shows a Microsoft Internet Explorer window titled "Guest Book Example - Add Entry - Microsoft Internet Explorer". The address bar displays the URL `http://localhost:8080/GuestBook/showForm.action`. The main content area has a heading "Add Entry". Below it are three input fields: "Your name:" with a text input box, "Your e-mail (optional):" with a text input box, and "Message:" with a multi-line text input box. A "Submit" button is located below the message input field. At the bottom of the form, there is a link "Back to the index page". The status bar at the bottom of the browser window shows "Done" and "Local intranet".

Figure B.5: The `form.ftl` showing a form with three input fields.

Enter name, email and some message text, and click over the ‘Submit’ button to see the output of `add.ftl` page, which shows the current message detail added (Figure B.6).

The screenshot shows a Microsoft Internet Explorer window titled "Guest Book Example - Entry added - Microsoft Internet E...". The address bar displays the URL `http://localhost:8080/GuestBook/addAction.action`. The main content area has a heading "Entry added". Below it, a message states "You have added the following entry to the guestbook:". Underneath, the details of the added entry are listed: "Name: Kogent", "Email: info@kogentindia.com", and "Message: Hello from Kogent". At the bottom of the page, there is a link "Back to the index page". The status bar at the bottom of the browser window shows "Done" and "Local intranet".

Figure B.6: The `add.ftl` showing current message detail.

Similarly, you can add more messages by accessing `form.ftl` page. After adding a few messages, the output of `index.ftl` page will be similar to what is shown in Figure B.7.

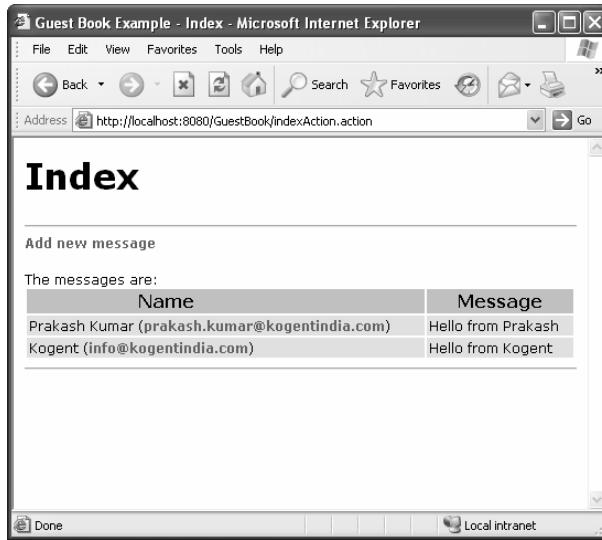


Figure B.7: The index.ftl showing list of messages.

After discussing FreeMarker for its all syntax and components, we can now move to another view technology here, i.e. Velocity. Similar to FreeMarker, Velocity is also supported by Struts 2 to design different views to be used in the application.

Introduction to Velocity

Velocity is an open source templating tool of Apache Software Foundation. It is compatible with the MVC-2 architecture. In the Struts Framework, the Velocity is introduced as a replacement for the Java Server Pages (JSP). It is a simple template language and is used as the form for the document. It is used as a template engine for the Web applications. The Web application includes the use of servlets, JSPs, and other useful technologies required for developing an application. A web template engine is the software that produces web document by processing the web template source files along with the data from a relational database. There is no Java class in a Velocity template. It is just an HTML document with some Velocity placeholders. The Velocity engine easily integrates into any Java application environment, especially Servlets. Velocity isolates Java code from the web pages, making web sites more maintainable. Therefore, It provides good alternative to JSP. Velocity is well-suited to J2EE Web development because the platform easily accommodates output technologies other than JSP. All the changes made to the applications will result in the configuration file. The configuration files contain detailed information about a particular application. The Velocity Framework offers the following features so that it is easy to implement Velocity as compared to using the JSP pages:

It is light, fast and versatile

The Velocity templates used in the application consists of less code as compared to Java Server Pages. It can also be implemented easily within an application. The output of the templates can be viewed, while executing the application. These can be used in several applications and in multiple environments, like SQL, XML, and Postscripts.

It is simple and powerful

It is observed that the Velocity template language is an API that the developers use to fit it for an application. The Velocity codes appear in a simple HTML editor. So it is easier to work with the

templates in an editor, instead of working with the Java Server Pages. The dynamic contents generated by using the Velocity tags can easily be edited by using editors, like Macromedia Dreamweaver, Microsoft FrontPage, etc. so that it is referred as a simple, yet the powerful implementation of ASF.

Velocity Template Language

Template languages are the formats for writing the codes for a document. In Struts Framework, the Velocity Template Language (VTL) is used to define variables for a document in a Web application. The VTL uses \$ sign to represent the texts that needs to be changed in the document. The Velocity Template Language (VTL) consists of two parts—references and directives. Let us understand them.

References

References available in VTL are used to access data in a particular document. These can be mixed with the non-VTL content available in the document. A reference always starts with a \$ (Dollar) sign. The Velocity template always accesses the object's public method. It refers to the object in the context. If no corresponding data object exists within the context then the template is simply treated as a text and the text will be displayed to the viewer.

Directives

Like the Velocity references, the Velocity directives can also be mixed with the non-VTL context in a document. The directives available in the Velocity tag language are given in Table B.1.

Table B.1 Velocity Directives

#set ()	#if ()	#else
#elseif ()	#end	#foreach ()
#include ()	#parse ()	#macro ()

Here's a description of the directives of the Velocity tag language:

- #set ()—The #set () directive is used to set a reference value within the template. The use of set can replace an existing object and can insert a new object in place of the old one. It is useful for the programmer to represent presentation logic in a document. We can use the common mathematical operators (+, -, *, /, %) to evaluate the contents of the method. This may also return the Boolean values:

```
#set ($startcount= 0)
#set ($Newcount = $current + 1 )
```

- #if(), #else(), #elseif()—It is a set of directives that are used to provide conditional functionalities in a document. These are the same as in the common programming languages. These directives use the common logical operators (&&, | |, ==, >, >=, <, <=) as well as the Boolean values:
-

Syntax:
 #if (condition)
 Expression 1

```
#else  
Expression 2  
#end
```

- ❑ `#end()` – This tag is used to finalize any statement in a document.
- ❑ `foreach()` – This directive is used to represent a list of objects. This tag takes two arguments, the reference to which the value is assigned and the value collection to the loop.
- ❑ `#include()` and `#parse()` – Both these directives take a template or a resource name as an argument, include the template in a place, and then implement it to the output stream. The difference between both the directives is that `#include()` includes the content to the current content without any processing, whereas the `#parse()` treats the specified resource as a template and processes the content against the current content.
- ❑ `#macro()` – This directive is used to create a parameterized bit of Velocity tag language called `velocimacro`. It can also be invoked like a directive. It is used to create reusable components of VTL that contains explicit parameters for easier usage, readability, and maintenance.

Working with Velocity

We have discussed earlier in this chapter that the Velocity templates are the replacements to the Java Server Pages used in a Web application. For page designers, working with Velocity makes implementation of codes in an application easy. In this context, we are going to use the Velocity tag templates, which are to be used by the application in generating the output. To apply templates in our application we need to maintain the complete directory structure for representing the application. The template file is to be placed along with the classes and lib folders in the WEB-INF directory. You can download Velocity JARs, like `velocity-1.5.jar` and `velocity-dep-1.5.jar`. You can also copy these JARs from the CD. Also make it sure that these jar files are included in the class path of your application. Figure B.8 shows the directory structure of the `velocity` application created here to implement Velocity.

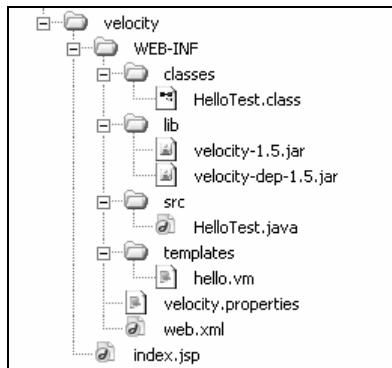


Figure B.8: Directory structure for representing Velocity templates.

The Velocity tags can be configured by placing the configuration items in the `velocity.properties` file. The `velocity.properties` file is placed with the `web.xml` file in the `WEB-INF` directory. Here's code, given in Listing B.11 for `velocity.properties` file (you can find `velocity.properties` file in `Code\Appendix B\velocity\WEB-INF` folder in CD):

Listing B.11: velocity.properties

```
resource.loader = file
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path =
C:/Program Files/Apache Software Foundation/Tomcat 5.5/webapps/velocity/WEB-
INF/templates
file.resource.loader.cache = true
file.resource.loader.modificationCheckInterval = 2
```

Here's Listing B.12 showing the configuration provided in web.xml file for this application (you can find the web.xml file in Code\Appendix B\velocity\WEB-INF folder in CD):

Listing B.12: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>HelloTest</servlet-name>
    <servlet-class>HelloTest</servlet-class>
    <init-param>
      <param-name>properties</param-name>
      <param-value>/WEB-INF/velocity.properties</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloTest</servlet-name>
    <url-pattern>/HelloTest</url-pattern>
  </servlet-mapping>
</web-app>
```

The web.xml file deploys the HelloTest servlet, which is used in the application. It also deploys the Velocity.properties file. The property file describes some features of the Velocity tags included in the application. The features include the standard configuration of the Velocity files. The Velocity.properties file for the application has already been described in Listing B.11.

The resource.loader tells Velocity that the templates are received from files. The file.resource.loader.class property tells Velocity location to search for templates. The value of file.resource.loader.cache property decides whether templates will be cached.

Here's the code, given in Listing B.13, for the template file, hello.vm, used in the application (you can find hello.vm file in Code\Appendix B\velocity\WEB-INF\templates folder in CD):

Listing B.13: hello.vm

```
<html>
<body style="font-family:verdana">
Hello $name
<br><br>
```

Appendix: B

```
Your Company: $company
</body>
<html>
```

A Velocity template is an HTML document with some special placeholders that Velocity will identify and replace with data supplied in Java program. The template file can contain any HTML code according to the need of the programmer. The \$name used in the template file is the variable. This variable access the data from the Servlet file described later in the chapter. This file is to be placed in the WEB-INF /templates directory. After running the following Java program, \$name is replaced with data supplied in the program.

Here's the code, given in Listing B.14, that describes the Servlet through which the template accesses the data for the application context (you can find HelloTest.java file in Code\Appendix B\velocity\WEB-INF\src folder in CD):

Listing B.14: HelloTest.java

```
import org.apache.velocity.Template;
import org.apache.velocity.servlet.VelocityServlet;
import org.apache.velocity.app.Velocity;
import org.apache.velocity.context.Context;
import javax.servlet.http.*;

public class HelloTest extends VelocityServlet {

    public Template handleRequest( HttpServletRequest request,
                                  HttpServletResponse response,
                                  Context context ) {

        Template template = null;
        try {
            context.put("name", "John");
            context.put("company", "Kogent Solutions Inc.");
            template = Velocity.getTemplate("hello.vm");
        }
        catch( Exception e ) {
            System.err.println("Exception caught: " + e.getMessage());
        }
        return template;
    }
}
```

This HelloTest Servlet extends the VelocityServlet, which is the base class for the Servlets in Velocity. In the handleRequest() method, an extra argument called context is passed along with the request and response objects. Here we place data that we want to use in place of the placeholder. The context () method takes the value that is to be accessed by the template file. We have provided index.jsp page with a simple hyperlink, which can be clicked to invoke HelloTest Servlet.

Here's the code, given in Listing B.15, for index.jsp (you can find index.jsp file in Code\Appendix B\velocity folder in CD):

Listing B.15: index.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head></head>
<body>
<a href="HelloTest">click Here</a>
</body>
</html>
```

Clicking over ‘Click Here’ hyperlink will invoke the HelloTest Servlet and the output will be similar to what is shown in Figure B.9.

**Figure B.9: Output of HelloTest Servlet showing use of velocity template file.**

The HelloTest accesses the template file `hello.vm` in order to display the output. If the servlet does not find any template file then it throws an exception and returns null, otherwise the contents of the template file along with the Java file is returned as the output to the user.

In addition to Servlet-based Web applications, Velocity is used in Java and SQL code generation, XML processing and transformation, and text processing.

FreeMarker vs. Velocity

The advantage of the Velocity over FreeMarker is that the Velocity is very simple than FreeMarker and has a larger third party support and user community, whereas the FreeMarker can be used in Model 2 Frameworks, such as Struts. The list of features that are extra in FreeMarker when compared to Velocity are as follows:

- Number and date support**—FreeMarker is used to perform arithmetic calculations and comparison on any number type. It is used to compare and format date/time values.
- Internationalization**—It is used to format dates and numbers locale sensitively. Identifiers may consist Arabic, Chinese letters, etc.

Appendix: B

- ❑ **Loop Control**—You can exit from loops. You can access control variables of outer loops from inside the inner loops.
- ❑ **Arrays**—You can ask the size of the array. You can access both primitive and non-primitive array elements by index.
- ❑ **Macros**—Macro calls may pass parameters either by position or by name. Macro parameters can have default values, which are effective when the parameter is omitted on call.

We can change the case of string to upper, lower or title case; convert string to HTML, XML or RTF. Also we can access elements of lists by index, concatenate lists and query size of lists. FreeMarker can be integrated with other technologies, like JSP custom tag libraries in templates, and python objects. FreeMarker has powerful XML transformation capabilities. We can use it in place of XSLT.

This chapter introduced Velocity and FreeMarker template languages. We created a Struts 2 based application with full implementation of FreeMarker. Another application simply introduced you to the uses of Velocity template files.



C

Implementing Tiles Plugin

When designing a Web application with a number of web pages, we may sometimes require some regions over the pages to be same in order to provide a consistent look and feel to all web pages. These regions can be classified as header and footer of the page, the menu bar on the page, and navigation, etc. Writing the code for these regions on each page creates results in duplication of the same code on all pages. Any change in the layout used in designing pages with consistent look or change in the structure of different regions defined earlier needs change in all the pages using the duplicated code. This is very cumbersome and needs some solution as the number of pages in an application may be in hundreds.

A web page can be divided into different fragments, like header, footer, menu and body, which can be grouped together to form a page. These fragments can be created separately in different single files. These separately developed fragments can be reused in all web pages designed in an application. This provides a consistent look to all pages and the change in these fragments automatically reflects in all pages which use them. These fragments can be defined as tiles and the framework, which enables us in defining these reusable tiles and integrating them together with multiple web pages with consistent look and feel is known as Tiles Framework.

Tiles is a framework which helps in designing the web pages using grouped fragments of the codes. The fragments, or tiles, are further assembled to get a complete page. This reduces the code duplication of the page elements which are common in the web pages. The reusable templates created using Tiles Framework can be used to provide a common look and feel to all the pages in the application.

The Tiles Framework is popular among developers as one of the components of Struts Framework, but Tiles can be used without Struts also. The Struts Tiles is extracted from Struts 1 Framework and further developed for its different versions. Tiles Framework has also been integrated with Struts 2 Framework with the help of a plugin provided with Struts 2 distribution. This plugin has been bundled as `struts2-tiles-plugin-2.0.6.jar` in Struts 2 API distribution. But, before describing the Struts 2 integration with Tiles 2 Framework, we need an in depth discussion on the Tiles 2 Framework first.

Tiles 2 Framework

Tiles, which is a templating system, is used to create web pages having common layout and were based on common template. Earlier Tiles were used with Struts 1 framework only, but now the new version of Tiles framework, which can be used without Struts 1 or Struts 2 framework, has also been released. This is known as Tiles 2 Framework. The latest version of Tiles 2 is version 2.0.4, but in this chapter we are going to focus on version 2.0.1, which is currently supported by Struts 2 Tiles plugin. Hereafter, we'll refer Tiles 2 simply as Tiles.

Tiles allows the creation of different reusable view components (also known as fragments or tiles) and assembling them into a single page according to a specific layout. The common layouts are used as templates for all the pages sharing similar design. The pages can be loaded with different tiles according to the definition provided for it. This page definition can be provided in a central configuration file (XML file) or can directly be inserted into the JSP page. The page definitions define the layout to be used and the tiles to be placed in the layout (template). We can create different layouts or reuse single layout. When a page is reloaded every time, the tiles are assembled dynamically. Figure C.1 explains this better.

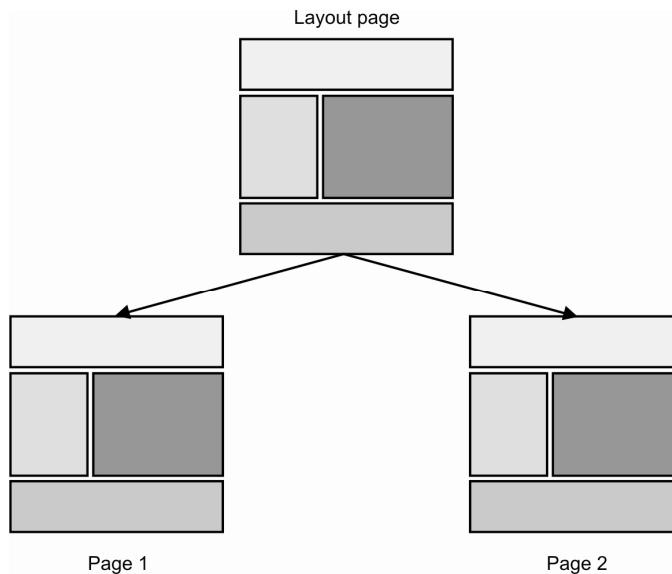


Figure C.1: Layout defining four different Tiles.

Figure C.1 shows that a layout page can be seen as a group of tiles/fragments. These tiles are other JSP pages, which can be defined to be inserted at the tiles location. The content to be inserted at various tiles locations is defined using a page definition for the page. The page definitions for different required output screens use a layout as common template with different tiles (JSP pages). The different shades in Page 1 and Page 2, shown in Figure C.1, indicate that these pages use the same layout and some common tiles (matching shades) at few places. But the two pages are different as the content of one tile is different (see tiles with different shades in Page 1 and Page 2). This use is formalized by Composite View pattern which allows the creation of web pages having similar structure, but having different content in their different sections. Tiles, which is an implementation of composite view pattern, makes it concrete by adding its own concepts, which includes the concept of Template, Attribute, and Definition.

Template

Normally, most of the pages in a Web application follow the same structure that can be achieved by using a single template. A template defines the common layout for all the web pages, which need consistency in its look and feel. A template provides the basic structure of the page in terms of tiles to be inserted at runtime, but the actual content for different regions is described using definitions.

Figure C.2 shows such a template.

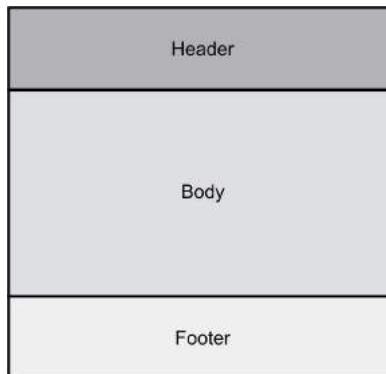


Figure C.2: A template with three Tiles.

Here's Listing C.1 that describes the code for a JSP page representing the layout shown in Figure C.2:

Listing C.1: A simple Layout JSP page (layout.jsp)

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<html>
<head>
<title><tiles:getAsString name="title"/></title>
</head>
<body>
<table cellspacing="0" align="center" >
<tr>
    <td height="100">
        <tiles:insertAttribute name="header"/>
    </td>
</tr>
<tr height="300">
    <td>
        <tiles:insertAttribute name="body"/>
    </td>
</tr>
<tr>
    <td height="100">
        <tiles:insertAttribute name="footer"/>
    </td>
</tr>
</table>
</body>
</html>
```

Attribute

The template creates a blank structure/layout for the page, but the content to be filled in by different segments/tiles is defined using the `insertAttribute` tag (see Listing C.1). These tags define the different attributes to be used by the template. Different pages may use the same template with different set of attributes. Four different attributes defined in Listing C.1 are title, header, body and footer. The type of different attributes can be as follows:

- ❑ **String**—An attribute of type `String` is directly rendered to the user as it is.
- ❑ **Template**—Template with all attributes filled, if any.
- ❑ **Definition**—A page definition with all attributes filled for the template used.

Definition

A definition is used to render a page to the user. The definition for a web page to be shown uses a template and values for attributes for this template. The definition may provide some or all attributes to be used in the template.

Here's a definition for a page using the template created in Listing C.1 that will look like the one shown in Listing C.2:

Listing C.2: tiles.xml with a definition

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
"http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<definition name="welcome.page" template="/layout.jsp">
    <put-attribute name="title" value="welcome" type="string"/>
    <put-attribute name="header" value="/mainheader.jsp"/>
    <put-attribute name="body" value="/mainbody.jsp"/>
    <put-attribute name="footer" value="/footer.jsp"/>
</definition>

</tiles-definitions>
```

If all the attributes to be used by the template are filled here, then the definition is known as the abstract definition that can be used as the base definition for other definitions extending it. The Tiles configuration file, which by default is `tiles.xml`, is used to define page definitions. A sample definition provided in `tiles.xml` file is shown in Listing C.2.

NOTE

Though Struts 2 framework provides lots of internationalization capabilities, Tiles 2 also provides the functionality of selecting tiles definitions according to the user locale. For this we just need to create different Tiles definition files for different locales. For example, if we have `tiles.xml` file as base tiles definition file, we can `tiles_en.xml`, `tiles_fr.xml` and `tiles_de.xml` files to provide tiles definitions to be used in case of different locales. The suffix used here, i.e. en, fr and de, follows the same conventions as used by resource bundles for Struts 2. Every thing about internalization has been discussed in Chapter 10.

Nesting Definitions

The page definitions have different sections which are filled by different contents using `<put-attribute>` element. We can insert another page definition into a single section of another page definition. This nesting of definitions is supported by Tiles.

Here's the code given in Listing C.3, showing how a definition can be put as an attribute value of a template:

Listing C.3: Two definitions showing nesting of definitions

```
<definition name="welcome.note" template="/someLayout.jsp">
    <put-attribute name="one" value="/ajsp"/>
    <put-attribute name="two" value="/b.jsp"/>
    <put-attribute name="three" value="/c.jsp"/>
</definition>
<definition name="welcome.page" template="/layout.jsp">
    <put-attribute name="title" value="Welcome" type="string"/>
    <put-attribute name="header" value="/mainheader.jsp"/>
    <put-attribute name="body" value="welcome.note"/>
    <put-attribute name="footer" value="/footer.jsp"/>
</definition>
```

Extending Definitions

A definition can extend an existing page definition also. The definition extending page can fill the partial filled definitions and can also override the attributes given in the base definition. Extending the definitions reduces the new page definitions to few lines, as in this case we do not need to provide all attributes. Here's an example, as shown in Listing C.4:

Listing C.4: Two definitions showing extending of definition

```
<definition name="welcome.page" template="/layout.jsp">
    <put-attribute name="title" value="Welcome" type="string"/>
    <put-attribute name="header" value="/mainheader.jsp"/>
    <put-attribute name="body" value="mainbody.jsp"/>
    <put-attribute name="footer" value="/footer.jsp"/>
</definition>
<definition name="manager.welcome.page" extends="welcome.page">
    <put-attribute name="title" value="Welcome - Manager" type="string"/>
    <put-attribute name="body" value="/manager_main.jsp"/>
</definition>
```

The definition `manager.welcome.page` extends definition `welcome.page` and inherits header, and `footer` attributes from it. The two attributes—`title` and `body`—are overridden here. Similarly, we can also override the template used in the base class as shown here:

```
<definition name="manager.welcome.page"
    extends="welcome.page" template="/otherLayout.jsp" >
```

This page definition extends the definition `welcome.page` and overrides the template used. All attributes defined for base definition are inserted in the new template `otherLayout.jsp`.

Getting Tiles 2 API

There are different versions of Tiles 2 available for download and implementation. The latest version is 2.0.4. But here, we are using the version 2.0.1, which is supported by Struts 2 Tiles plugin for integrating Tiles 2 with Struts 2 Framework. So, download the distribution JAR for Tiles 2.0.1 from following links:

- <http://tiles.apache.org/download.html>
- <http://archive.apache.org/dist/tiles/v2.0.1/>

Unpack the downloaded binary distribution (tiles-core-2.0.1-bin.zip) and copy the following two JARs into your WEB-INF/lib folder from the extracted folder::

- tiles-core-2.0.1.jar
- tiles-api-2.0.1.jar

Also, copy the following Tiles dependencies JARs into your WEB-INF/lib folder after copying it from the lib folder of binary distribution of Tiles 2.0.1:

- commons-beanutils-1.7.0.jar
- commons-digester-1.8.jar
- commons-logging-api-1.1.jar

The API includes a set of interfaces and class which together make the Tiles Framework. The important classes include TilesListener, TilesServlet, TilesFilter, TilesContainer, BasicTilesContainer, MutableTilesContainer, TilesContainerFactory, TilesAccess, TilesException, TilesRequestContext, TilesContextFactory, etc.

The TilesListener Class

The TilesListener (`org.apache.tiles.listener.TilesListener`) class is used as a listener for the initialization of Tiles container. A Tiles container is represented by an interface `org.apache.tiles.TilesContainer`, which encapsulates Tiles Framework and exposes tiles features to other frameworks using it through some plugin. The `createContainer()` method of `TilesListener` class is used to get a reference of `TilesContainer`. Table C.1 lists the methods of `TilesListener` class.

Table C.1: Methods of TilesListener class

Method	Description
<code>void contextDestroyed (ServletContextEvent event)</code>	It removes TilesContainer from the service.
<code>void contextInitialized (ServletContextEvent event)</code>	It initializes TilesContainer and also places it into service.
<code>TilesContainer createContainer (javax.servlet.ServletContext context)</code>	It returns a reference of the TilesContainer.

The **TilesContainerFactory** Class

The `createContainer()` method of `TilesListener` class basically uses the `createContainer()` method on the object of `org.apache.tiles.factory.TilesContainerFactory` class. Here's the code of `TilesListener.createContainer()` method:

```
protected TilesContainer createContainer(ServletContext context)
    throws TilesException
{
    TilesContainerFactory factory = TilesContainerFactory.getFactory(context);
    return factory.createContainer(context);
}
```

The `TilesContainerFactory` is provided to create the default container implementation and its initialization before putting it into service. The `TilesContainerFactory.createContainer()` may return the reference of `org.apache.tiles.impl.BasicTilesContainer` or `org.apache.tiles.mgmt.MutableTilesContainer` after initializing it. A context parameter `org.apache.tiles.CONTAINER_FACTORY.mutable` can be set to `true` to get a `MutableTilesContainer`, which supports runtime definitions.

Here's the code for `TilesContainerFactory.createContainer()` method:

```
public TilesContainer createContainer(Object context) throws TilesException{
    String value =
        getInitParameter(context, "org.apache.tiles.CONTAINER_FACTORY.mutable");
    if(Boolean.parseBoolean(value)){
        return createMutableTilesContainer(context);
    } else{
        return createTilesContainer(context);
    }
}
```

The **TilesServlet** and **TilesFilter** Class

The `TilesServlet` (`org.apache.tiles.servlet.TilesServlet`) is a simple Servlet provided for backward compatibility to be used as Tiles initialization Servlet. This Servlet uses the `TilesListener` class and simply invokes the `contextInitialized()` method over listener object. Further proceedings, like getting appropriate `TilesContainer` using `TilesContainerFactory`, are, as usual, described earlier.

Similarly, the `org.apache.tiles.filter.TilesFilter` class can be used to process the reloadable tiles definitions. The `org.apache.tiles.filter.TilesFilter` class extends the `org.apache.tiles.servlet.TilesServlet` class and implements the `javax.servlet.Filter` interface.

Configuring Tiles 2

Earlier the Tiles Framework was used with Struts Framework only, but the evolution of Tiles 2 made this technology independent. Now, we can develop Web applications with Tiles 2 only in a Servlet-based environment and can use it in the Web application development integrating it with other frameworks,

like Struts 2. Before discussing its integration with Struts 2, we'll describe the different ways to configure Tiles Framework in our application. We have basically three options to start Tiles. We can use any of the three available options to configure Tiles. These are as follows:

- ❑ Loading `org.apache.tiles.servlet.TilesServlet`
- ❑ Loading `org.apache.tiles.filter.TilesFilter`
- ❑ Loading `org.apache.tiles.listener.TilesListener`

Loading TilesServlet Class

We can configure the `org.apache.tiles.servlet.TilesServlet` in `web.xml` file of our Web application, which can be loaded at startup. This Servlet can be defined similar to other Servlets with some optional `init` parameters to it. The `TilesServlet` is just a startup Servlet and do not need any Servlet mapping to serve any request. This can be configured in `web.xml` file as shown in Listing C.5:

Listing C.5: TilesServlet Mapping in `web.xml` file

```
<servlet>
    <servlet-name>tiles</servlet-name>
    <servlet-class>org.apache.tiles.servlet.TilesServlet</servlet-class>
    <init-param>
        <param-name>org.apache.tiles.DEFINITIONS_CONFIG</param-name>
        <param-value>/WEB-INF/tiles-defs.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

The `init-param org.apache.tiles.DEFINITIONS_CONFIG` (or `definitions-config`) defines the path of Tiles configuration file. So, the name and location of Tiles configuration file can be customized and defined here. We can also define this parameter as a context parameter:

```
<context-param>
    <param-name>org.apache.tiles.DEFINITIONS_CONFIG</param-name>
    <param-value>/WEB-INF/tiles-defs.xml</param-value>
</context-param>
```

The default Tiles configuration file is `WEB-INF/tiles.xml` and we need to define it as `init` parameter if we are using the `tiles.xml` file as Tiles configuration file. We can customize the different tiles setting, defining classes to be used by defining some other context parameters, which are as follows:

- ❑ `org.apache.tiles.CONTAINER_FACTORY`
- ❑ `org.apache.tiles.CONTAINER_FACTORY.mutable`
- ❑ `org.apache.tiles.CONTEXT_FACTORY`
- ❑ `org.apache.tiles.DEFINITIONS_FACTORY`
- ❑ `org.apache.tiles.PREPARER_FACTORY`

Loading **TilesFilter** Class

We can also define a filter with filter mapping. The filter used here is `org.apache.tiles.filter.TilesFilter`. It is usually used when the definition files need to be changed and reloaded frequently. Here's Listing C.6 showing the filter mapping given in `web.xml` file:

Listing C.6: Filter mapping for TilesFilter in `web.xml` file

```
<filter>
    <filter-name>Tiles Filter</filter-name>
    <filter-class>org.apache.tiles.filter.TilesFilter</filter-class>

    <init-param>
        <param-name>org.apache.tiles.DEFINITIONS_CONFIG</param-name>
        <param-value>/WEB-INF/tiles-defs.xml</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Tiles Filter</filter-name>
    <url-pattern>*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Loading **TilesListener** Class

The third and final option to configure tiles is to register a listener in `web.xml` file and define optional parameters as context parameters. The listener to be registered here is `org.apache.tiles.listener.TilesListener`.

Here's Listing C.7 that shows how `TilesListener` is registered with a context parameter in `web.xml` file:

Listing C.7: Registering `TilesListener` listener class in `web.xml` file

```
<context-param>
    <param-name>org.apache.tiles.DEFINITIONS_CONFIG</param-name>
    <param-value>/WEB-INF/tiles-defs.xml</param-value>
</context-param>
. . .
. . .
<listener>
    <listener-class>org.apache.tiles.listener.TilesListener</listener-class>
</listener>
```

REMEMBER

All classes used here are according to *Tiles 2 version 2.0.1*. In case you are trying some newer version, like 2.0.3 or 2.0.4, then the classes and parameter names used may change according to different APIs provided by different versions. But when trying to integrate with *Struts 2*, always use version 2.0-SNAPSHOT or 2.0.1 as the new *Tiles 2* version may not be fully supported by *Struts 2 Tiles Plugin 2.0.6* which we are going to discuss here.

Creating and Using Tiles Pages

After the brief introduction about getting Tiles 2 APIs and configuring Tiles 2 Framework, let's create some JSP pages using Tiles. The creation of tiles pages includes creating a template page, creating the page definition in Tiles configuration file, and generating tiles pages using Tiles tags from tag library bundled with the Tiles distribution. The `tiles-core.tld` tag file defines all the Tiles tags, which can directly be used in our JSP page after providing a taglib directive, as shown here:

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
```

Here's a simple JSP page, given in Listing C.8, that uses a page definition, say `welcome.page`, defined in the Tiles configuration file:

Listing C.8: A sample tiles page (JSP) using welcome.page definition

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertDefinition name="welcome.page"/>
```

Instead of using the page definition provided in the Tiles configuration file, we can also define the page structure in the JSP page itself. It is similar to giving a page definition in Tiles configuration file.

Here's Listing C.9 that shows the template to be used inserted using the `insertTemplate` tag and all attributes filled using `putAttribute` tag:

Listing C.9: A sample Tiles page

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertTemplate template="/layout.jsp">
<tiles:putAttribute name="title" value="Welcome" type="string"/>
<tiles:putAttribute name="header" value="/mainheader.jsp"/>
<tiles:putAttribute name="body" value="/mainbody.jsp"/>
<tiles:putAttribute name="footer" value="/footer.jsp"/>
</tiles:insertTemplate>
```

This JSP page, shown in Listing C.9, uses `layout.jsp` (see Listing C.1) as a template and does not use any definition from Tiles configuration file. So, we have both the options, either to define the page in Tiles configuration file or use tiles tags to create page structure at runtime, as shown in Listing C.9. We can directly insert a template filling different attributes at runtime.

The tiles tags allow the filling of templates, modifications of the definitions, and creation of new definitions at runtime. We have seen how the template is inserted and filled (Listing C.9) and how definitions are inserted in JSP page (Listing C.8). We can also modify the definitions, while using it in a JSP page as shown in Listing C.10:

Listing C.10: A JSP page using and modifying the definition

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertDefinition name="welcome.page"/>
<tiles:putAttribute name="body" value="/otherBody.jsp" />
</tiles:insertDefinition>
```

Similarly, we can create definitions at runtime. This needs our application to be configured to use Mutable container. This can be achieved by providing a context parameter named `org.apache.tiles.CONTAINER_FACTORY.mutable` and setting its value to `true` as shown here:

```
<context-param>
    <param-name>org.apache.tiles.CONTAINER_FACTORY.mutable</param-name>
    <param-value>true</param-value>
</context-param>
```

Now, we can create definition in our JSP pages using tiles tags. This definition will be available during the request and expires after that.

Here's the code, given in Listing C.11, for a JSP page creating a runtime definition:

Listing C.11: A sample JSP page creating runtime definition

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:definition name="runtime.def" template="/layout.jsp">
    <tiles:putAttribute name="title" value="Welcome" type="string"/>
    <tiles:putAttribute name="header" value="/header.jsp"/>
    <tiles:putAttribute name="body" value="/body.jsp"/>
    <tiles:putAttribute name="footer" value="/footer.jsp"/>
</tiles:definition>
<tiles:insertDefinition name="runtime.def" />
```

After this discussion about Tiles 2 Framework, we can now move on to its integration with Struts 2 Framework. Struts 2 comes with a plugin to support its integration with Tiles 2.

Struts 2 Tiles Plugin

The Struts 2 Tiles plugin is used to integrate Tiles Framework within a Struts 2 pages application where the action execution consequently gives the next output screen defined as one of the results for a given action. This plugin allows the actions to return tiles pages to the user. This plugin is bundled with Struts 2 distribution in the form of a JAR file named `struts2-tiles-plugin-2.0.6.jar`. Struts 2 tiles plugin basically supports two versions of Tiles 2, i.e. 2.0. SNAPSHOT and 2.0.1. So, it is suggested that this plugin should be used carefully, while working with another versions of Tiles 2, because it may give some errors in its implementation. This plugin introduces some new packages and classes with a `struts-plugin.xml` file. The JAR for this plugin includes the following:

- ❑ `org.apache.struts2.tiles` package:
 - `ConfiguredServletContext.class`
 - `StrutsTilesContainerFactory.class`
 - `StrutsTilesListener.class`
 - `StrutsTilesRequestContext.class`
- ❑ `org.apache.struts2.views.tiles` package:
 - `TilesResult.class`
- ❑ `struts-plugin.xml`

All classes bundled with this plugin are to integrate Tiles 2 with Struts 2 so that Tiles can be implemented with Struts 2 actions. The two most important classes to be discussed here are

StrutsTilesListener listener class and TilesResult class. The StrutsTilesListener class is an extension of the traditional TilesListener listener class bundled with Tiles 2 distribution and is used to provide tight integration with Struts 2 features. The StrutsTilesListener class uses StrutsTilesContainerFactory class and StrutsTilesRequestContext class. The StrutsTilesContainerFactory and StrutsTilesRequestContext classes are again an extension of TilesContainerFactory class and TilesRequestContext class, respectively.

The main objective of this plugin is to enable Struts 2 actions to return web pages rendered using tiles definitions. This is supported by the use of a new result class introduced, i.e. TilesResult class. Let's have some brief discussion on the StrutsTilesListener class and TilesResult class.

StrutsTilesListener Class

The Struts 2 Tiles plugin brings a replacement to the standard org.apache.tiles.listener.TilesListener class.

This is the org.apache.struts2.tiles.StrutsTilesListener class, which extends the TilesListener class. The StrutsTilesListener listener class gives a tighter integration with Struts 2 features, such as Freemarker integration. If there is no tiles container factory set explicitly using context parameter org.apache.tiles.CONTEXT_FACTORY, the StrutsTilesListener class sets it to StrutsTilesContainerFactory class.

The org.apache.struts2.tiles.StrutsTilesContainerFactory class extends org.apache.tiles.factory.TilesContainerFactory class and behaves as a wrapper over this class and creates tiles container, initializes it, and puts it into service.

The StrutsTilesContainerFactory further defines an inner class StrutsTilesContextFactory extending the TilesContextFactory and provides a method to create TilesApplicationContext and TilesRequestContext. When using Struts 2 Tiles plugin, we'll register this listener.

TilesResult Class—New Result Type

The addition of this plugin provides a new result type, i.e. TilesResult. The org.apache.struts2.views.tiles.TilesResult class extends the org.apache.struts2.dispatcher.ServletDispatcherResult, which happens to be our default result type. This result type enables our actions to return a result code to render a tiles definition. Now, in addition to using page locations to be rendered to the user, we can provide page definition that is to be rendered for a result, given the result type used here is TilesResult.

Here's the doExecute() method of TilesResult, given in Listing C.12, that shows how it uses the TilesContainer to render the definition:

Listing C.12: doExecute() method of TilesResult class

```
public void doExecute(String location, ActionInvocation invocation) throws
Exception {
    setLocation(location);
    ServletContext servletContext = ServletActionContext.getServletContext();

    TilesContainer container = TilesAccess.getContainer(servletContext);
```

```
HttpServletRequest request = ServletActionContext.getRequest();
HttpServletResponse response = ServletActionContext.getResponse();
container.render(request, response, location);
}
```

A default tiles package is defined in struts-plugin.xml bundled with this plugin JAR. The package has TilesResult configured as a new result type. This package can be extended by your own package in struts.xml file.

Here's the code, given in Listing C.13, showing the content of struts-plugin.xml () file contained in tiles plugin JAR:

Listing C.13: struts-plugin.xml

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="tiles-default" extends="struts-default">
        <result-types>
            <result-type name="tiles" class=
                "org.apache.struts2.views.tiles.TilesResult"/>
        </result-types>
    </package>
</struts>
```

Using Struts 2 Tiles Plugin

The tiles plugin can be used in any Struts 2 based Web application to integrate it with Tiles. The process of using this plugin and making tiles pages in our Struts 2 Web application can be described in the following simple steps:

- Registering StrutsTilesListener
- Registering TilesResult
- Configuring action to use tiles

Registering StrutsTilesListener

To provide a tight integration of Struts 2 features with Tiles Framework features, the plugin provided listener is registered in web.xml file, replacing the standard TilesListener, as shown here:

```
<listener>
<listener-class>org.apache.struts2.tiles.StrutsTilesListener</listener-class>
</listener>
```

Registering TilesResult

The new result type introduced by plugin must be registered before it is used. Here's Listing C.14 that provides a result-types element in struts configuration file, i.e. struts.xml file:

Listing C.14: Registering TilesResult as result type

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <package name="mydefault" extends="struts-default">
        <result-types>
            <result-type name="tiles" class=
                "org.apache.struts2.views.tiles.TilesResult"/>
        </result-types>
        <action . . .></action>
    </package>
</struts>
```

The struts-plugin.xml file included in this plugin JAR defines a package named tiles-default. The TilesResult is registered as a result type in this package. The struts-plugin.xml file is loaded before struts.xml and, hence, we can use the tiles-default package and use the TilesResult without registering it, as shown in Listing C.15:

Listing C.15: Using tiles-default package

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <package name="mydefault" extends="tiles-default">
        <action name="someAction" class="com.kogent.action.SomeAction">
            <result type="tiles">welcome.page</result>
        </action>
    </package>
</struts>
```

Configuring Action to use Tiles

After registering the listener in the web.xml and TilesResult as a result in the struts.xml file, we are now ready to use this result type to render a definition provided in the Tiles configuration file (tiles.xml, by default). To enable tiles page rendering through results configured for an action, the type attribute of the <result> element must be defined as tiles. This invokes the TilesResult class to process the result for the user, which will be rendered according to the definitions provided here and defined in Tiles configuration file.

Here's example, given in Listing C.16, where an action mapping is provided with result of type tiles:

Listing C.16: Using tiles result

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
```

```

<package name="mydefault" extends="tiles-default">
    <action name="someAction" class="com.kogent.action.SomeAction">
        <result name="login" type="tiles">login.def</result>
        <result name="success" type="tiles">welcome.def</result>
    </action></package>
</struts>

```

After this discussion about the Tiles 2 related concepts and APIs, we can go for real implementation in a working application which is fully discussed in the chapter.

By now you are familiar with Tiles 2, including various concepts, like template, definition, and attribute along with a thorough discussion on the creation of tiles pages with examples implementing the different tags from the tiles tag library. Most importantly, we saw how the integration of Struts 2 Framework with Tiles using the Struts 2 Tiles plugin has made these two powerful concepts together to be used in a single Web application. This way the Struts 2 application has the capability to render a page using its definition provided in Tiles configuration file. Now, we'll see how Struts 2 Tiles plugin is implemented and how Struts 2 action can be integrated with Tiles through an application.

Creating an Application with Struts 2 Tiles Plugin

We going to create an application which implements Struts 2 Tiles plugin and describes how Struts 2 action can be integrated with Tiles. For this we'll implement different concepts discussed in a working Web application. The application is simple Struts 2 application and, hence, its directory structure is kept standard like the ones we are using earlier. The only extra implementation in this application is the use of Struts 2 Tiles plugin and using a new result type, i.e. `TilesResult`. Before starting our discussion, let's see the directory structure of the application being developed here in Figure C.3. Figure C.3 shows the different directories, code files, and JARs.

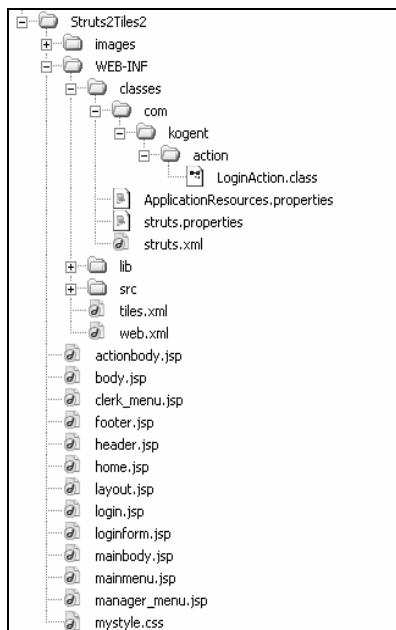


Figure C.3: Directory structure of application.

NOTE

You can create a directory structure similar to Figure C.3 and copy the required files, like `ApplicationResources.properties`, `struts.properties`, `struts.xml`, `tiles.xml`, `web.xml` and other JSP pages from the CD, or alternatively you can create your own copy while we are discussing them step by step in this section.

Getting Prepared

Before we start coding for our application, we must have all the JARs required in our `WEB-INF/lib` folder. In addition to having Struts 2 related JARs, like `struts2-core-2.0.6.jar` and `xwork-2.0.1.jar`, we must have Tiles JAR and most importantly the Struts 2 Tiles Plugin JAR added to our `WEB-INF/lib` folder. Therefore, add the following JARs to your lib folder, which includes some Tiles dependency JARs:

- ❑ Tiles JARs (version 2.01) with dependency JARs
 - `tiles-core-2.0.1.jar`
 - `tiles-api-2.0.1.jar`
 - `commons-beanutils-1.7.0.jar`
 - `commons-digester-1.8.jar`
 - `commons-logging-api-1.1.jar`
- ❑ Struts 2 Tiles Plugin JAR
 - `struts2-tiles-plugin-2.0.6.jar`
- ❑ Struts 2 JARs
 - `struts2-core-2.0.6.jar`
 - `xwork-2.0.1.jar`
 - `freemarker-2.3.8.jar`
 - `ognl-2.6.11.jar`

Now, we are ready with all the required APIs to develop a working application, based on Struts 2 and integrated with Tiles.

Configuring Struts 2 Tiles Plugin

A Struts 2 based application has filter mapping for `org.apache.struts2.dispatcher.FilterDispatcher` class. In addition to this we also need to register a listener provided by the Tiles plugin being used here to integrate Tiles with Struts 2. The class `org.apache.struts2.tiles.StrutsTilesListener` provides a better support for Struts 2 features and hence it is preferred over the traditional `TilesListener`.

Here' the configuration provided in `web.xml` file, as shown in Listing C.17 (you can find `web.xml` file in `Code\Appendix C\Struts2Tiles2\WEB-INF` folder in CD):

Listing C.17: `web.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<display-name>Tiles 2 Plugin Example</display-name>
<filter>
<filter-name>struts2</filter-name>
<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
<listener-class>org.apache.struts2.tiles.StrutsTilesListener</listener-class>
</listener>
<welcome-file-list>
<welcome-file>home.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

Here, we have not provided any context parameter to configure the CONTEXT_FACTORY and Tiles configuration file. Hence, the defaults are implied here. The default Tiles configuration file is tiles.xml and the context factory used here is StrutsTilesContainerFactory.

Creating Template

We are designing pages for a Web application and it is always good if all of the web pages from an application have similar design with the same look and feel. This purpose can be solved if all pages of the application share a common template. Tiles supports the usage of a template when giving a page definition. A template can also be defined as a group of tiles/fragments. These tiles can be filled dynamically according to the different attributes set for the definitions being used to render a page. A template can be created using the tiles tags, which provide a basic structure for all the pages sharing this template. Let's design a pattern for our pages in the form of tiles using the most common design—design having header, footer, menu and body, as shown in Figure C.4.

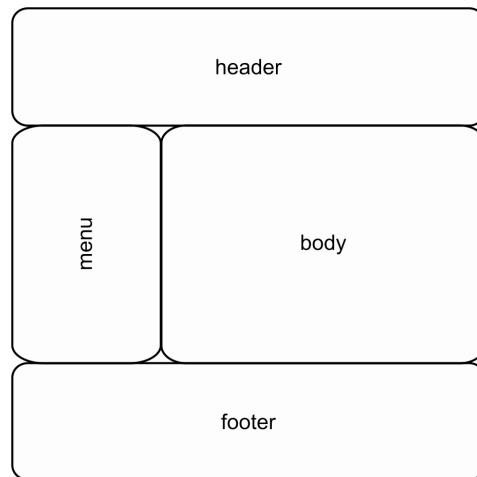


Figure C.4: The layout of a template.

Appendix: C

A template is represented by a JSP page which uses tiles tags to define the basic structure. The template created here is `layout.jsp`. Observe the code and the attribute names defined to be inserted in the template to complete the page.

Here's the code, given in Listing C.18, for `layout.jsp` (you can find `layout.jsp` file in `Code\Appendix C\Struts2Tiles2` folder in CD):

Listing C.18: `layout.jsp`

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<html>
<head>
<title>
    <tiles:getAsString name="title"/>
</title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body topmargin="0">
<table width="800" cellspacing="0" align="center" >
<tr>
    <td colspan="2" height="70">
        <tiles:insertAttribute name="header"/>
    </td>
</tr>
<tr height="300">
    <td width="200" valign="top">
        <tiles:insertAttribute name="menu"/>
    </td>
    <td width="600">
        <tiles:insertAttribute name="body"/>
    </td>
</tr>
<tr>
    <td colspan="2" height="70">
        <tiles:insertAttribute name="footer"/>
    </td>
</tr>
</table>
</body>
</html>
```

The tags—`<tiles:getAsString/>` and `<tiles:insertAttribute/>`—are used to define the attributes to be inserted in the page. The attributes can be changed to generate a new page, but using the same template assures that all of them have the same position for the reusable tiles, like header, footer, menu, and body.

Definitions in `tiles.xml`

The Tiles configuration file used by this application is `tiles.xml`, which is the default one. All the definitions used in this application are provided in this file. So let's create the `tiles.xml` file and save it in `WEB-INF` folder.

Here's the code, given in Listing C.19, for `tiles.xml` (you can find `tiles.xml` file in `Code\Appendix C\Struts2Tiles2\WEB-INF` folder in CD):

Listing C.19: tiles.xml with a definition

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
"http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
<definition name="login.def" template="/layout.jsp">
    <put-attribute name="title" value=
        www.simplylogin.com/Login type="string"/>
    <put-attribute name="header" value="/header.jsp"/>
    <put-attribute name="menu" value="/mainmenu.jsp"/>
    <put-attribute name="body" value="/loginform.jsp"/>
    <put-attribute name="footer" value="/footer.jsp"/>
</definition>
</tiles-definitions>
```

The only definition entered in this file is `login.def`. See this definition and observe the other code files, which are used here. A definition may define a template to be used with all the required attribute, filled partially or fully.

The definition here uses `layout.jsp` as the template and fills all its attributes with different view components, like `header.jsp`, `mainmenu.jsp`, `loginform.jsp` and `footer.jsp`. These files are yet to be created. This definition can be inserted in a JSP page or can directly be rendered for the user using `TilesResult` without creating any new JSP page. We'll see both implementations in the coming sections.

Creating Tiles Enabled JSPs

A JSP page can easily be changed to a tiles page (fragmented into tiles) using tiles tags. This type of page design using tiles tags helps in reducing duplication of code to design a number of pages having the same structure in an application. The basic structure of the template is used. We have the two different ways to create a tiles page:

- Inserting Template using `<tiles:insertTemplate/>`
- Inserting Definition using `<tiles:insertDefinition/>`

The template can be directly inserted into a JSP page and all the attributes filled using the `<tiles:putAttribute/>` tag giving the name and value for the attribute. We do not need to provide a definition in the `tiles.xml` file here.

Here's the code, given in Listing C.20, of the `home.jsp` page which uses `<tiles:insertTemplate/>` and fills all the attributes for the template (`layout.jsp`) used (you can find `home.jsp` file in `Code\Appendix C\Struts2Tiles2` folder in CD):

Listing C.20: home.jsp

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertTemplate template="/layout.jsp">
<tiles:putAttribute name="title"
    value="www.simplylogin.com - Home page" type="string"/>

<tiles:putAttribute name="header" value="/header.jsp" />
```

Appendix: C

```
<tiles:putAttribute name="menu" value="/mainmenu.jsp"/>
<tiles:putAttribute name="body"   value="/mainbody.jsp" />
<tiles:putAttribute name="footer" value="/footer.jsp" />
</tiles:insertTemplate>
```

The output of this page can be seen when we start the application, as it is defined `welcome-list-file` in `web.xml`. The output of `home.jsp` page is shown in Figure C.5.

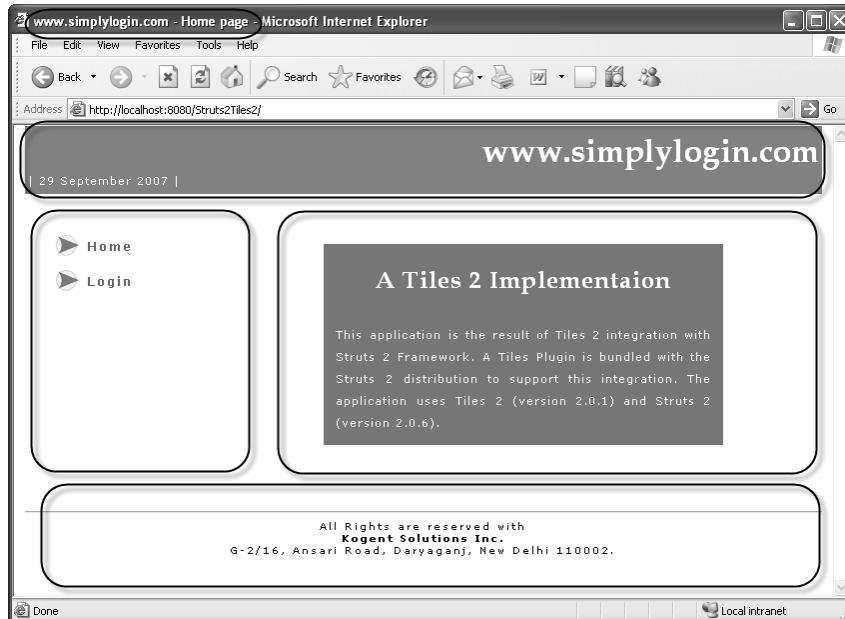


Figure C.5: The `home.jsp` designed using template `layout.jsp`.

The five round-corner rectangles are used here to represent five different attributes used to fill the `template layout.jsp`. The first attribute is of type `String` and the content directly gets displayed here using `<tiles:getAsString/>` tag in `template layout.jsp`. The basic structure of the JSP is rendered according to the structure of the template being used, i.e. `layout.jsp` page. Can you guess the source of contents for these different fragments/tiles?. These are simple JSP pages, like the `header.jsp`, the `footer.jsp`, the `mainmenu.jsp`, and the `mainbody.jsp` as defined by `<tiles:putAttribute/>` tag in Listing C.20. So before you access `home.jsp`, make sure that you have all these four JSP pages saved in your project folder. Before accessing the `home.jsp` file, create the following view components:

- `header.jsp`
- `footer.jsp`
- `mainmenu.jsp`
- `mainbody.jsp`

The `header.jsp` creates a banner to be used in all pages designed in this application. Here, the code, given in Listing C.21, for `header.jsp` (you can find these JSP files in `Code\Appendix C\Struts2Tiles2` folder in CD):

Listing C.21: header.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table width="100%" height="100%" bgcolor="#5a84da">
<tr>
<td align="left" valign="bottom" class="text">
| <s:date name="new java.util.Date()" format="dd MMMM yyyy"/> | </td>
<td align="right">
<h1 style="color:#ffffff;font-family:Book
Antiqua">www.simplylogin.com</h1></td></tr>
</table>
```

Similarly, footer.jsp is the content to be displayed at the bottom of every page. Here's the code, given in Listing C.22, for footer.jsp (you can find these JSP files in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.22: footer.jsp

```
<hr>
<div align=center style="font-size:10;letter-spacing:2">
All Rights are reserved with <br><b>Kogent Solutions Inc.</b><br>
G-2/16, Ansari Road, Daryaganj, New Delhi 110002.
</div>
```

Here's Listing C.23 that shows the mainmenu.jsp page providing two hyperlinks (you can find these JSP files in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.23: mainmenu.jsp

```
<p>&nbsp;</p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left">
<a href="home.jsp">Home</a></td>
</tr>
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left">
<a href="login.jsp">Login</a></td>
</tr>
</table>
```

The mainbody.jsp page is a simple JSP page showing some message. Here's the code, given in Listing C.24, for mainbody.jsp (you can find these JSP files in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.24: mainbody.jsp

```
<table width="400" align="center" cellpadding="10" bgcolor="#a362b3" >
<tr><td class="text">
<h2 align="center" style="font-family:Book Antiqua">A Tiles 2 Implementaion</h2>
<p align="justify">
This application is the result of Tiles 2 integration with
Struts 2 Framework. A Tiles Plugin is bundled with the Struts 2
distribution to support this integration.
The application uses Tiles 2 (version 2.0.1) and Struts 2 (version 2.0.6).
</p>
</td></tr>
</table>
```

You can see the output of these reusable JSP pages in different pages, as shown in Figure C.5. The four round shaped rectangles separate the output of header.jsp, mainmenu.jsp, mainbody.jsp and footer.jsp.

Similarly, we can design a new Tiles page using the definition provided in tiles.xml file. We can directly insert the definition in the JSP page using `<tiles:insertDefinition/>` tag.

Here's the code, given in Listing C.25, for login.jsp, which uses page definitions login.def given in tiles.xml of Listing C.19 (you can find login.jsp file in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.25: login.jsp

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertDefinition name="login.def"/>
```

The `<put-attribute/>` element in tiles.xml file sets attributes for the page definition login.def (see Listing C.19). The view components used for login.def definition are header.jsp, footer.jsp, mainmenu.jsp and loginform.jsp. All JSP pages have already been created and only the JSP page required here is loginform.jsp. This JSP page simply gives a form with three input fields which have been named as 'loginid', 'password' and 'type' (See Listing C.26). Here's the code, given in Listing C.26, for loginform.jsp page (you can find loginform.jsp file in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.26: loginform.jsp page

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
<tr><td class="text" align="center">
| Login |
<br><s:actionerror/>

<s:form action="login" method="post" cssClass="text">
<s:textfield name="loginid" key="app.loginid"/>
<s:password name="password" key="app.password"/>
<s:select key="app.type"
name="type"
list="#{'manager':'Manager', 'clerk':'Clerk'}"/>
<s:submit value="Login"/>
```

```
</s:form>
</td></tr>
</table>
```

The `loginform.jsp` uses Struts tags to create a form and input fields. Before we see the output of `loginform.jsp` and `login.jsp`, we need to discuss over the action required to process date through this form. This is where Struts 2 and Tiles will integrate to work together.

Using **TilesResult**

The `login.jsp` page shows you a login form which, when submitted, has to be handled by some action class. The action class execution may give some result code and the next view is decided accordingly. The integration of Struts 2 Framework with Tiles gives a new result type to be used, which enables the action to return a new page rendered to the user created through definitions in `tiles.xml` file. The new result class is `TilesResult` and you have the option to either register it in your package defined in `struts.xml` file or extend `tiles-default` package defined in `struts-plugin.xml` file.

But before jumping over the action and result configuration details to be provided in `struts.xml` file, see the code of a simple action class, i.e. `LoginAction` with its `execute()` and `validate()` method.

Here's the code, given in Listing C.27, for `LoginAction.java` (you can find `LoginAction.java` file in `Code\Appendix C\Struts2Tiles2\WEB-INF\src\com\kogent\action` folder in CD):

Listing C.27: `LoginAction.java`

```
package com.kogent.action;

import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    String loginid;
    String password;
    String type;

    public String getLoginid() {
        return loginid;
    }
    public void setLoginid(String loginid) {
        this.loginid = loginid;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}
```

```
public String execute() throws Exception {  
    if(loginid.equals(password)){  
        if("manager".equals(type)){  
            return "manager";  
        }else{  
            return "clerk";  
        }  
    }else{  
        this.addActionError(getText("app.invalid"));  
        return ERROR;  
    }  
}  
public void validate(){  
    if( (loginid == null) || (loginid.length() == 0) ) {  
        this.addFieldError("loginid", getText("app.loginid.blank"));  
    }  
    if( (password == null) || (password.length() == 0) ) {  
        this.addFieldError("password", getText("app.password.blank"));  
    }  
}  
}
```

The possible result codes returned by `LoginAction` class are manager, clerk, error, and input. Let's see the action mapping provided in `struts.xml` file to process the login form submission.

Here's the code, given in Listing C.28, for showing the action mapping for login action (you can find `struts.jsp` file in `Code\Appendix C\Struts2Tiles2\WEB-INF\classes` folder in CD):

Listing C.28: struts.xml file with action mapping and tiles result

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE struts PUBLIC  
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  
"http://struts.apache.org/dtds/struts-2.0.dtd ">  
<struts>  
    <package name="default" extends="tiles-default">  
        <action name="login" class="com.kogent.action.LoginAction">  
            <result name="error"/>/login.jsp</result>  
            <result name="input" type="tiles">login.def</result>  
            <result name="manager"  
                type="tiles">manager.welcome.def</result>  
            <result name="clerk"  
                type="tiles">clerk.welcome.def</result>  
        </action>  
    </package>  
</struts>
```

The package created here extends tiles default package, which is available because of the loading of `struts-plugin.xml` file prior to `struts.xml` file. Hence, we do not need to register a new result type here and can directly use tiles as result type.

Observe the action mapping for action named `login` in Listing C.28. There are four results configured with this action. See the types and location of these results. If a JSP page is being used as the location

then the result type should be the default dispatcher. But, the powerful feature added by Tiles plugin here is the use of tiles as result type, which helps in rendering a page definition from Tiles configuration file. We can use a JSP page and a page definition, instead of generating a view to the user, as shown here:

```
<result name="error">/login.jsp</result>
<result name="input" type="tiles">login.def</result>
```

The second result configured here with name="input" will consequently show you the page reading its definition from tiles.xml file and the definition used here is login.def. This further eliminates the need for creating a new JSP page. We can, instead, give a page definition easily in tiles.xml file.

The action may return two more result codes other than input and error. These result codes are manager and clerk, which are returned according to the type selected by the user while logging. The user should get two different consoles as their type is different. This can easily be implemented by giving new definitions using the same template, but different attributes. The two new definitions used here are manager.welcome.def and clerk.welcome.def. Here are the two results:

```
<result name="manager" type="tiles">manager.welcome.def</result>
<result name="clerk" type="tiles">clerk.welcome.def</result>
```

We need to provide these new definitions in tiles.xml file to generate an appropriate view for two different types of users. The templates used by these definitions too will be same, i.e. layout.jsp. We only need to define different attributes to be used in these two definitions.

Here's the code, given in Listing C.29, for the two new definitions added in tiles.xml file:

Listing C.29: Two new definitions added in tiles.xml file

```
<definition name="manager.welcome.def" template="/layout.jsp">
<put-attribute name="title"
    value="www.simplylogin.com/Login Successfull - Manger" type="string"/>
<put-attribute name="header" value="/header.jsp"/>
<put-attribute name="menu" value="/manager_menu.jsp"/>
<put-attribute name="body" value="/body.jsp"/>
<put-attribute name="footer" value="/footer.jsp"/>
</definition>

<definition name="clerk.welcome.def" extends="manager.welcome.def">
<put-attribute name="title"
    value="www.simplylogin.com/Login Successfull - Clerk" type="string"/>
<put-attribute name="menu" value="/clerk_menu.jsp"/>
</definition>
```

The definition manager.welcome.def is provided with all the attributes filled. The clerk.welcome.def definition is created by extending the manager.welcome.def definitions with the required attributes overridden for the clerk type of user. These definitions use some previously created and used JSP page and some yet to be created JSP pages, like body.jsp, manager_menu.jsp, and clerk_menu.jsp,.

Here's the code, given in Listing C.30, for body.jsp (you can find these JSP pages in Code\Appendix C\Struts2Tiles2 folder in CD):

Appendix: C

Listing C.30: body.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
<tr><td class="text" align="center">
| Successful Login!|
<br><br><br><br>
You have logged in as : <b><s:property value="loginid"/></b>
<br><br><br><br>
| Use Navigation Links provided in Menu Bar. |
</td></tr>
</table>
```

The manager_menu.jsp and clerk_menu.jsp page provide different set of navigational links and is designed for two different types of user.

Here's the code, given in Listing C.31, for manager_menu.jsp (you can find these JSP pages in Code\Appendix C\Struts2Tiles2 folder in CD):

Listing C.31: manager_menu.jsp

```
<p>&nbsp;</p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left"><a href="home.jsp">Home</a></td>
</tr>
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left"><a href="manager.action">New</a></td>
</tr>
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left"><a href="manager.action">Edit</a></td></tr>
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left"><a href="manager.action">Delete</a></td>
</tr>
<tr height="30" valign="middle">
<td valign="middle" width="10">

</td>
<td align="left"><a href="logoff.action">Logoff</a></td>
</tr>
</table>
```

Here's the code, given in Listing C.32, for `clerk_menu.jsp` (you can find these JSP pages in `Code\Appendix C\Struts2Tiles2` folder in CD):

Listing C.32: `clerk_menu.jsp`

```
<p>&nbsp;</p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            
        </td>
        <td align="left"><a href="home.jsp">Home</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            
        </td>
        <td align="left"><a href="clerk.action">Deposit</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            
        </td>
        <td align="left"><a href="clerk.action">Withdraw</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            
        </td>
        <td align="left"><a href="clerk.action">Transfer</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            
        </td>
        <td align="left"><a href="logoff.action">Logoff</a></td>
    </tr>
</table>
```

Now, we are ready to see how the login process works and how different page definitions are rendered through `TilesResult` result class. Click on the 'Login' hyperlink shown in Figure C.5. The output of `login.jsp` page, which uses the definition `login.def`, is shown in Figure C.6.

Appendix: C

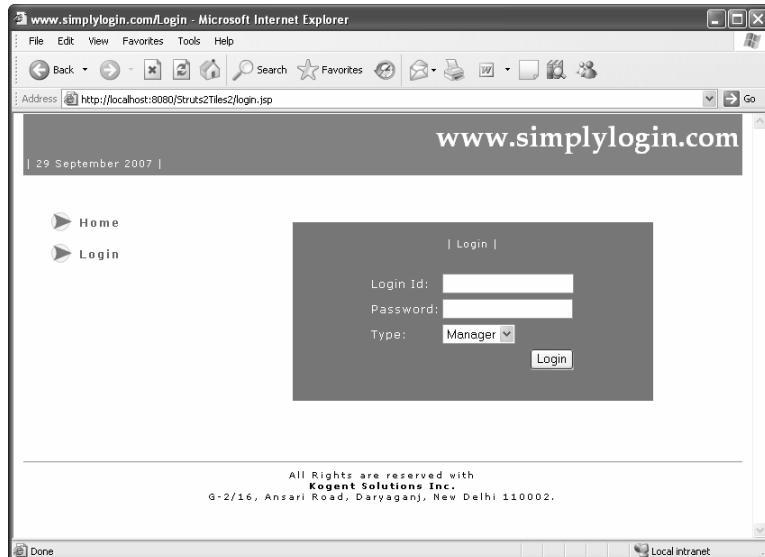


Figure C.6: The login.jsp page using login.def definition.

We can get the result code, `input`, if we leave any field blank, which results in the invocation of `TilesResult` again to render the `login.def` definitions (see action mapping in Listing C.28). But this time the field errors, set by `LoginAction` class, are available and displayed, as shown in Figure C.7.

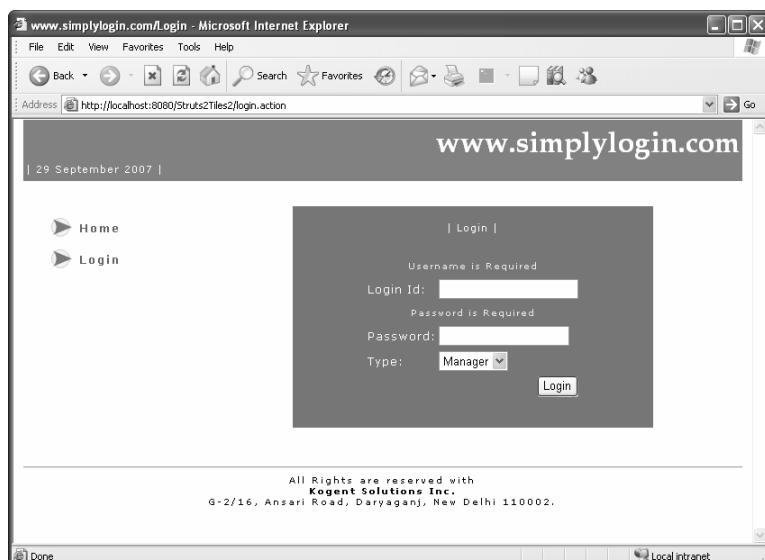


Figure C.7: The login.jsp showing field errors when "input" returned from action.

Similarly, we get `manager` as the result code when the same string is passed as login id and password and the type selected is 'Manger'. This results in the rendering of the definition `manager.welcome.def` with its output shown in Figure C.8. Observe the list of options shown in the menu bar created using `manager_menu.jsp` page.

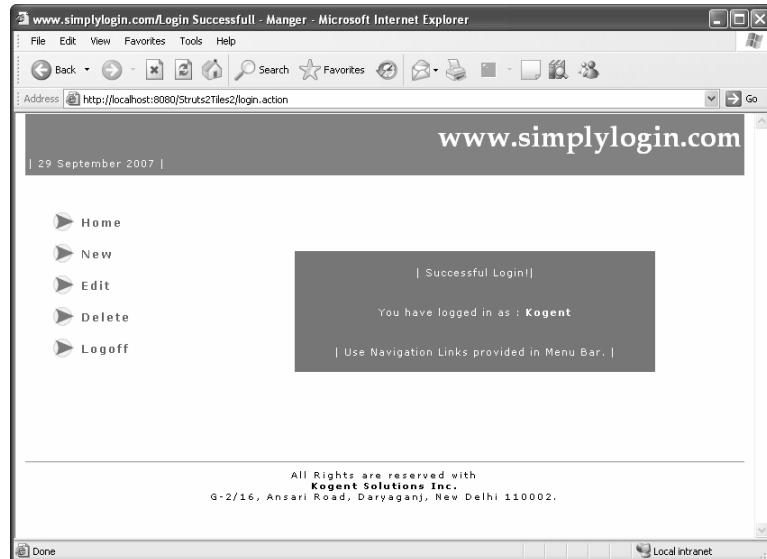


Figure C.8: Output of definition manager.welcome.def.

Similarly, if the type selected is ‘Clerk,’ you’ll get a different output using `clerk.welcome.def` definition which uses a different value for `menu` attribute and the list of options, as displayed in Figure C.9.

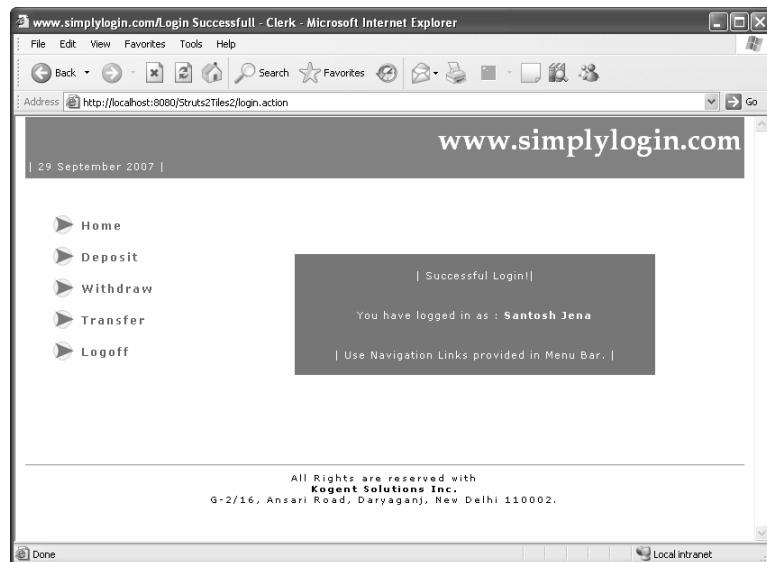


Figure C.9: Output of definition clerk.welcome.def.

Now compare Figure C.8 and Figure C.9. These figures are designed using the same template `layout.jsp` and, hence, have a similar structure. They both use reusable tiles, like header, footer and body. However, they are different for the `title` and `menu` attributes, which simply make the two pages distinct from each other, even though the look and feel of the pages are same. This all appear just by

Appendix: C

writing few lines describing the page definitions, which are quite simple in comparison to writing the whole code to design a new page.

The new hyperlinks provided here may not be working and so we can simply provide some new action mappings and new page definitions to make them work.

Here's the new action mappings, given in Listing C.33, to be added in `struts.xml` file to make these hyperlinks work:

Listing C.33: New action mappings in struts.xml

```
<action name="clerk">
    <result type="tiles">clerk.def</result>
</action>

<action name="manager">
    <result type="tiles">manager.def</result>
</action>

<action name="logoff">
    <result type="tiles">login.def</result>
</action>
```

The new definitions required here are `clerk.def` and `manger.def`, which can be added to our `tiles.xml` file similar to other definitions added.

Here are these two definitions, given in Listing C.34, to be added to your copy of `tiles.xml` file:

Listing C.34: New definitions of manager.def and clerk.def

```
<definition name="manager.def" extends="manager.welcome.def">
    <put-attribute name="title" value="www.simplylogin.com - Manager" type="string"/>
    <put-attribute name="body" value="/actionbody.jsp"/>
</definition>

<definition name="clerk.def" extends="manager.def">
    <put-attribute name="title" value="www.simplylogin.com - Clerk" type="string"/>
    <put-attribute name="menu" value="/clerk_menu.jsp"/>
</definition>
```

The only new view component used in these definitions is `actionbody.jsp`. Create this JSP page before using new definitions.

Here's Listing C.35 that shows the new component used in definitions (you can also find `actionbody.jsp` file in `Code\Appendix C\Struts2Tiles2` folder in CD):

Listing C.35: `actionbody.jsp` page

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
<tr><td class="text" align="center">
<br><br>

<h2>| Under Construction |</h2>
<br><br><br><br>
```

```
This page is underconstruction. <br>Please try after some time.  
</td></tr>  
</table>
```

Click on the ‘Logoff’ hyperlink, shown in Figures C.8 and C.9, to return to the page rendered using definitions `login.def`.

The new menu options, shown in Figures C.8 (like ‘New,’ ‘Edit’ and ‘Delete’) and C.9 (like Deposit, Withdraw and Transfer), can be clicked to invoke associated actions, i.e. ‘manger’ and ‘clerk’ which results in the display of a page using the definition `manager.def` (defined in Listing C.34 and output shown in Figure C.10) and `clerk.def` (defined in Listing C.34 and output not shown here).

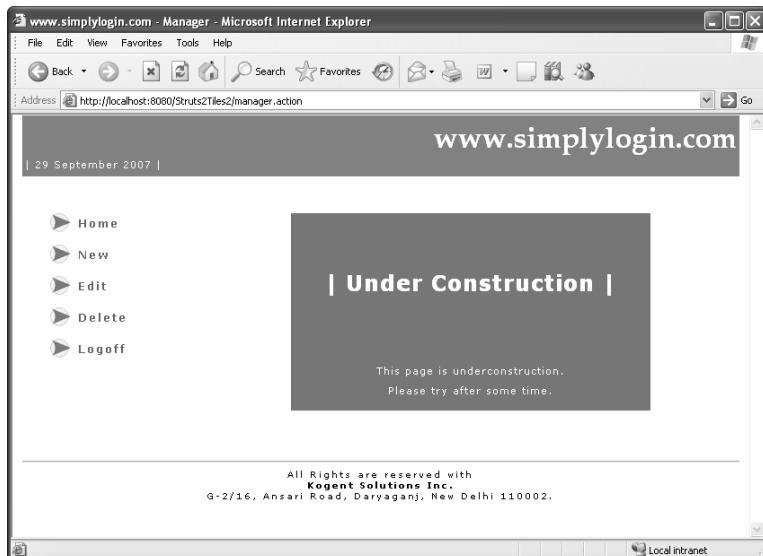


Figure C.10: Output of definition manager.def.

In this chapter, we have implemented Struts 2 Tiles plugin to integrate our Struts 2 based application with Tiles. The Struts 2 action was then capable of rendering page definitions using result type `TilesResult`. This integration resulted in the development of web pages having consistent look and feel throughout the application. The web page development using Tiles surely makes life easy for web page designers, as it reduces the duplicated codes and, hence, the efforts to design a new page using concepts of reusable tiles and definitions.



D

Configuring Struts 2 Application

In this chapter, we'll discuss about the configuration of Struts 2 application. Here we'll understand the configuration files and their structures supported in Struts 2. If you understand the basic configuration available in Strut 1 thoroughly, then it would be easy for you to learn the different types of configurations for different components in Struts 2. This chapter is about the structures of various Struts 2 configuration files and their respective purposes.

So what are configuration files? In simple layman terms, a configuration file is a file that stores the total description of the components that are associated with an application and required by the Web container to run the application. These configuration files are required for the initial setting of an application. In Struts 2 application, they perform a very crucial role in the execution of a program. The file that contains the configuration information for a particular program is known as the configuration file. When a program starts, the configuration files are used to see what parameters or attributes are in effect.

A Struts 2 configuration file can be defined as an XML document that describes the parts of a Struts 2 application, including actions, results, and Interceptors. A Struts 2 configuration file contains information about many types of resources and configures the interaction among them. In addition to some XML file used to configure Struts 2 application, we have some .properties file, which also affects the functioning of the application. The properties files also contain different properties and values set for them to customize the application. Therefore, before getting into what and how it is configured, we can describe where exactly the configuration details are to be defined.

Configuration Files

As we have seen in Struts 1, one of the important configuration files is the Web application Deployment Descriptor or web.xml file. In Struts 2 too this file plays an important role in the configuration of Web application. This file gives full control to the developer for configuring Struts-based application. By default, Struts loads a set of internal configuration files to configure it, and then another set of configurations for configuring your application.

The files that are used to configure an application in Struts 2 are as follows:

- ❑ web.xml
- ❑ struts.xml
- ❑ struts.properties
- ❑ struts-default.xml
- ❑ struts-plugin.xml

Apart from these, as Struts allows dynamic reloading of XML configuration file, some configuration files can be reloaded dynamically. This dynamic reloading makes interactive development possible. Using dynamic reloading, you can reconfigure your action mapping of your application during development. As there is a possibility of slight performance penalty, this is not implemented, in general. But if you want to use this feature, you need to set struts.configuration.xml.reload to 'true' in your copy of struts.properties file. We'll go deep into studying the struts.properties file later in this chapter.

Let's now discuss the important configuration files in detail in the subsequent topics.

Configuration File—web.xml

It is already defined that web.xml configuration file plays an important role in building a Struts applications. As Servlet programs are included in your Struts applications, it is necessary to keep the mapping functions inside the web.xml file. The web.xml file must reside in the WEB-INF folder of your Web application. This file, web.xml Web application Deployment Descriptor file, represents the core of the Web application. So, we consider this file as an important file for the Web applications based on Struts Framework. Here, in the web.xml file, Struts defines its Front Controller, i.e. the FilterDispatcher. FilterDispatcher is a Servlet filter class that initializes the Struts Framework and handles all the requests. This filter can contain initialization parameters, which answer this question "what will happen if additional configuration files are loaded and how the framework will react correspondingly?"

Along with FilterDispatcher, Struts also offers an ActionCleanupFilter class. The main responsibility of the ActionCleanupFilter class is to handle a special clean-up task, when other filters need access to an initialized Struts Framework.

Key Initialization Parameters

The common key initialization parameters for the <filter> element of the web.xml file are as follows:

- ❑ config—a comma-delimited list of XML configuration files to load
- ❑ actionPackages—a comma-delimited list of Java packages to scan for Actions
- ❑ configProviders—a comma-delimited list of Java classes that implement the ConfigurationProvider interface that should be used for creating the configuration
- ❑ *—any other parameter is treated as framework constants

Here's the code, given in Listing D.1, that shows you the filter and the filter mapping inside the Deployment Descriptor web.xml:

Listing D.1: A sample web.xml file

```
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://j
```

```

ava.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <filter>
        <filter-name>struts</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
        <init-param>
            <param-name>actionPackages</param-name>
            <param-value>com.mycompany.myapp.actions</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>struts</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- ... -->
</web-app>

```

The mapping shown in Listing D.1 makes your Web application Struts enabled. The `<url-pattern>`, which is set to `/*`, makes sure that all requests are handled by the Front Controller, i.e. `FilterDispatcher` class. This class further handles the request and uses another configuration detail given in different configuration files. The next configuration files to be discussed in detail is `struts.xml`.

*Configuration File—**struts.xml***

The `struts.xml` file is a default file for the Struts 2 Framework, also known as the core configuration file for this Framework. The appropriate place for `struts.xml` file is `WEB-INF/classes` folder in your Web application. It contains various configuration details for framework specific components, like actions, results, and Interceptors.

If it is required, you can also insert other configuration files to the default file present. This helps in dividing large `struts.xml`.

Here the question arises, whether we can divide the large `struts.xml` file into smaller parts. There are two approaches for doing so:

- You can include other `struts.xml` files, those you want, from a bootstrap or
- You can package a `struts.xml` files in a JAR.

So, a `struts.xml` can contain more than one include elements as shown in the following code snippet:

```

<struts>
    <include file="struts-default.xml"/>
    <include file="config-browser.xml"/>
    <package name="default" extends="struts-default">
        .....
    </package>
    <include file="other.xml"/>
</struts>

```

The first `include` statement tells the framework to load the `struts-default.xml` file, which is generally found in `struts2-core-2.0.6.jar` file. This `struts-default.xml` defines all the default bundled results and Interceptors and many Interceptor stacks. You can put your own

<include> elements in your struts.xml interchangeably with <package> elements. These files will be loaded in the order of their appearance.

According to the second approach, you can insert a module to your application, by placing a struts.xml and related classes into a JAR on the CLASSPATH.

Listing D.2, shows the basic structure of struts.xml file and the important tags that are used during the configuration of an application along with their attributes.

Listing D.2: struts.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <result-types>
            <result-type name="result1" class="SomeResultClass1"/>
            <result-type name="result2" class="SomeResultClass2"/>
            .
            .
            .
        </result-types>
        <interceptors>
            <interceptor name="interceptor1" class="SomeInterceptorClass1"/>
            <interceptor name="interceptor2" class="SomeInterceptorClass2"/>
            .
            .
            .
        </interceptors>
        <default-interceptor-ref name="defaultStack1"/>
        <global-results>
            <result name="resultname" type="resulttype1">/xxx</result>
            .
            .
            .
        </global-results>
        <action name="action" class="SomeActionClass">
            <result name="success">some_page.jsp</result>
            .
            .
            .
        </action>
        .
        .
    </package>
</struts>
```

In Listing D.2 you can observe the various components of Struts 2 and their configuration settings in the struts.xml file. The various components or elements are configured under the opening and closing tags of <struts> element. The default configuration file, struts-default.xml, has also been

included in this file. Only those elements, which are usually configured in the struts.xml are currently shown in the file. These include the <result-types>, <interceptors>, <interceptor-stack>, <global-results> and <action>. These elements are required during the configuration of a Struts 2 Web application.

Configuration File—struts.properties

The framework of a Struts application uses a number of properties, which can be changed according to the requirement of a user. The location of the struts.properties file is WEB-INF/classes folder of your Web application. This property file is similar to other resource bundle files having the extension .properties and which contains key/value pairs representing various properties and their values.

You can place the struts.properties file anywhere on the CLASSPATH, but it is typically found under the WEB-INF/classes directory of the application. You can find the total list of properties in the default.properties file, which is present inside the struts2-core-2.0.6.jar.

In other words, the struts.properties can be used to override the default values set for the different keys in default.properties file. This setting of keys help in customizing Struts application. For example, setting of struts.devMode to ‘true’ helps in debugging the Struts-based applications.

All the properties can also be set using Constant Configuration in a XML configuration file, which offers a simple way to customize a Struts application by defining key settings that is able to restructure the framework and plugin behavior.

Let’s go into the details of the properties available inside the default.properties. Table D.1 displays different properties, which are defined in default.properties and can be overridden in struts.properties file.

Table D.1: Properties to customize a Struts 2 application

Properties	Description
struts.configuration = org.apache.struts2.config.DefaultConfiguration	It defines the configuration class required to configure Struts and gets the configuration parameter into Struts
struts.locale=en_US struts.i18n.encoding=UTF-8	This is required to set the default locale and encoding scheme in your Strut application
struts.objectFactory=spring	This line indicates that the default object factory can be overridden here. Short-hand notation is supported in some cases such as ‘spring’. In other ways, you can use com.opensymphony.xwork2.ObjectFactory sub-class name here
struts.objectFactory.spring. autoWire = name	It defines the autoWiring logic when using the SpringObjectFactory. There are four types of values used like ‘name’, ‘type’, ‘auto’, and ‘constructor’. By default, it takes the value as ‘name’
struts.objectFactory.spring. useClassCache = true	It describes the Struts-spring integration. If the class instance should be cached, then it should be left as true, until a future Spring release makes it possible. You can specify either true or false. By default it is true

Table D.1: Properties to customize a Struts 2 application

Properties	Description
<code>struts.objectTypeDeterminer = tiger</code> <code>struts.objectTypeDeterminer = notiger</code>	It describes that the default object type determiner can be overridden here. The shorthand notation ‘tiger’ and ‘notiger’ is given in some cases, alternatively you can use <code>com.opensymphony.xwork2.util.ObjectTypeDeterminer</code> implementation name and ‘notiger’ value is used to disable the tiger support
<code>struts.enable.SlashesInActionNames = false</code>	If you want to allow slashes in your action names, you need to set the value to be ‘true’ in this property. If ‘false’, action names cannot have slashes and will be accessible via any directory prefix. The value ‘true’ is useful when you use wildcards and store value in the URLs
<code>struts.tag.altSyntax= true</code>	It uses alternative syntax that requires <code>%{ } </code> in most places to evaluate expression for String attributes for tags
<code>struts.devMode = false</code>	The development mode can be set to false or true by setting this property. When it is set to true, Struts behaves closer and friendlier to the developer.
<code>struts.configuration.xml.reload=false</code>	This property supports reloading of the configuration. It reloads configuration of the <code>struts.xml</code> , when it is changed
<code>struts.url.http.port = 80</code> <code>struts.url.https.port = 443</code>	These are used to build URLs, such as URLTag
<code>struts.url.includeParams=get</code> or <code>all</code>	Three possible values for this property are ‘none’, ‘get’ or ‘all’
<code>struts.custom.i18n.resources =testmessages, testmessages2</code>	It loads the custom default resource bundles
<code>struts.xslt.nocache=false</code>	When user wants to make the use of style sheet caching, he needs to configure the <code>XSLTResult</code> class, using this property and setting the value to be ‘true’, otherwise ‘false’
<code>struts.configuration.files= struts-default.xml, struts-plugin.xml, struts.xml</code>	This property shows a list of configuration files automatically loaded by Struts
<code>struts.mapper.alwaysSelectFullNamespace=false</code>	It defines whether to always select the namespace before the last slash or not

Configuration File—**struts-default.xml**

The struts-default.xml is generally found in struts2-core-2.0.6.jar. This file may be included at the top of the struts.xml file to include the standard configuration setting without having to copy them. This file is automatically loaded from the struts2-core-2.0.6.jar file placed in your WEB-INF/lib folder. The struts-default.xml file contains default configuration for the framework results, Interceptors and Interceptor stacks. All the default configuration regarding results and Interceptors is defined inside the package, struts-default. The package defined in our struts.xml file can extend the <package> defined in struts-default.xml file. This makes all the default results, Interceptors, and Interceptor stacks available to be used in our struts.xml file.

Here's the basic structure, given in Listing D.3, of struts-default.xml file:

Listing D.3: Structure of struts-default.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <bean class="com.opensymphony.xwork2.ObjectFactory" name="xwork" />
    .
    .
    <bean class="com.opensymphony.xwork2.ObjectFactory"
        static="true" />

    <package name="struts-default">
        <result-types>
            <result-type name="chain"
                class="com.opensymphony.xwork2.ActionChainResult"/>
            <result-type name="dispatcher"
                class="org.apache.struts2.dispatcher.ServletDispatcherResult"
                default="true"/>
            .
            .
            .
            </result-types>

        <interceptors>
            <interceptor name="alias"
                class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
            <interceptor name="chain"
                class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
            <interceptor name="params"
                class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>
            <interceptor name="workflow"
                class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>
            <interceptor name="store"
                class="org.apache.struts2.interceptor.MessageStoreInterceptor" />
            <interceptor name="checkbox"
                class="org.apache.struts2.interceptor.CheckboxInterceptor" />
            <interceptor-stack name="basicStack">
                <interceptor-ref name="exception"/>

```

```
<interceptor-ref name="servlet-config"/>
    . . .
</interceptor-stack>

<interceptor-stack name="validationWorkflowStack">
    . . .
</interceptor-stack>

//Some other interceptor stacks. . . .
</interceptors>
<default-interceptor-ref name="defaultStack"/>
</package>
</struts>
```

In all the Struts 2 applications created in the previous chapters of this book, most of the time we have used the default result and Interceptor stack. For example, if we declare some result as `<result name="success"></result>` without giving a value for its `type` attribute, the default result type taken is ‘dispatcher’. Similarly, when there is no Interceptor configured in an action mapping, the default set of Interceptor executed is `defaultStack`. All these default results and Interceptor are extended to our `strut.xml` file by extending `struts-default` package after including the `struts-default.xml` file in `struts.xml` file.

NOTE

All the default Interceptors and Results are covered in detail in Chapter 4 and Chapter 8 respectively.

Configuration Elements

All Web applications need a Deployment Descriptor to make the initialization of resource, like Servlets and taglibs. Similarly, the Struts Framework also needs a different configuration (`struts.xml`) file to initialize its own resources. These resources include Interceptors to preprocess request, Action classes to execute business logic, and results to prepare views for the end user. The default configuration file (`struts.xml`) for Struts Framework has a set of valid elements, like `<package>`, `<bean>`, `<action>`, and many more. Following are the elements used for different types of configuration:

- Bean configuration
- Constant configuration
- Package configuration
- Namespace configuration
- Include configuration
- Interceptor configuration
- Action configuration
- Result configuration
- Exception configuration

Bean Configuration

Inside bean configuration, you can find the attributes, like `class`, `type`, `name`, `scope`, `static`, and `optional`. Among these, the only required attribute is ‘`class`’. It specifies the Java class to be

created or manipulated. A framework container can generate a bean and the bean is applied to the internal framework object or have values injected to its static methods.

The object injection can be implemented by the **type** attribute present in beans, which gives the information about which interface the object will implement.

The value injection can be implemented through the **static** attribute present in bean. It is good for allowing objects, which are not created by the container to receive framework constants. The functions of the different attributes present in bean are as follows:

- ❑ **class**—It gives the name of the bean class.
- ❑ **type**—It places the primary Java interface this class implements.
- ❑ **name**—It gives the name of the bean, it is always unique.
- ❑ **scope**—It offers the scope for the bean, it can be default, singleton, request, session, or thread.
- ❑ **static**—It specifies whether to inject static method or not; it will be “false” when the type is specified.
- ❑ **optional**—It tells whether the bean is optional?

Constant Configuration

The constant configuration offers a simple way to customize a Struts application by defining key settings that are able to restructure the framework and plugin behavior in XML files. The default constants are declared and given default values in `default.xml` and they can be overridden in `struts.properties` as discussed earlier in this chapter. In addition, we can define different values for these keys in XML file, like `web.xml` as well as `struts.xml`. In various XML variants, to create the constant configuration, we require two attributes—`name` and `value`.

Here's how the constant configuration is generated in different XML files (Listing D.4):

Listing D.4: Configuring constants

```
//Constant configuration inside struts.xml file
<struts>
<constant name="struts.devMode" value="true"/>
.....
</struts>

//Constant configuration in web.xml
<web-app>
    <filter>
        <filter-name>struts</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
        <init-param>
            <param-name>struts.devMode</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
</web-app>
```

In this way, we can define different name/value pairs using `<constant>` element in `struts.xml`.

Package Configuration

We know packages are something like container, which are useful to group actions, result, result types, Interceptor, and Interceptor-stacks into a logical configuration unit. In a general way, we can say packages are similar to the objects in that they can be extended and have some part which can be overridden by sub-packages. The four attributes used with this element are **name**, **extends**, **namespace**, **abstract**. The description of these four is given here:

- ❑ **name**—This attribute specifies a unique name for the package and it acts as a key to other packages to reference.
- ❑ **extends**—It inherits properties of other packages, to which it extends.
- ❑ **namespace**—This is used to ensure only authorized users can access the actions in a given namespace.
- ❑ **abstract**—It is useful to make the package to be abstract. This attribute creates a base package, which can avoid the action configurations.

Among the preceding four attributes, the only required attribute is ‘name’ whose value behaves as a key for the future reference by other packages.

One thing we have to remember here is that the configuration file is processed sequentially downwards the document. So, we need to keep the package which is referenced by an **extends**, before the package which extends it. The format of the package configuration inside the **struts.xml** is as shown here:

```
<struts>
    <package name="package1" extends="default" namespace="/package1">
        . . .
        .
        .
    </package>
</struts>
```

Namespace Configuration

Inside any package configuration, there is some basic role of namespace configuration. We know that there are many big Struts applications, which contain hundreds of pages or may be more than that. In such a case, there is the big chance of creation of action name conflict. To organize such large applications and to remove this (action name conflict) problem, Struts configuration makes the use of ‘packages’ and ‘namespaces’. Each package can set its own defaults, including a namespace setting.

Generally what it does is it subdivides all the action configurations into logical modules, each having its own identifying prefix. Each Namespace can have their own set of actions with same action name, along with their own implementation. Namespace prefix does not need to be embedded in forms and links. If you’ll not provide any namespace, the default namespace is an empty String. When an action configuration is not found in a specified namespace, then the default namespace is also searched. The root namespace is denoted as /. The root behaves as namespace, when the request, directly under the context path, is received. Like others, it will access the default namespace, if a local action is not found.

Here’s Listing D.5 showing the three different packages in a single **struts.xml** file:

Listing D.5: Configuring package in struts.xml file

```
<package name="default">
    <action name="useraction" class="somepack.someAction">
```

```

        <result name="success">welcome.jsp</result>
    </action>
        <action name="manageraction" class="somepack.someAction">
            <result name="success">bar1.jsp</result>
        </action>
    </package>

<package name="userpack" namespace="/">
    <action name="useraction" class="somepack.someAction">
        <result name="success">user.jsp</result>
    </action>
</package>

<package name="managerpack" namespace="/manager">
    <action name="manageraction" class="somepack.someAction">
        <result name="success">manager.jsp</result>
    </action>
</package>

```

Consider the preceding example. Here's three types of package declaration are shown. The first package declaration does not have any namespace; the second has root namespace declaration; whereas the third has namespace /manager. Consider the third one. If the request is made upon /manager/manageraction.action, then the /manager namespace is searched. If it is found, the appropriate action will be executed. In the example, the manageraction action is present inside the namespace /manager. So the manager.manageraction action will be executed, and if it returns success, then the request will be forwarded to manager.jsp. Now observe the second package declaration; it is using root namespace. If the request is made for useraction.action, then it will start search for the root namespace. In case the root namespace is not found, it will search for the default namespace. Here, in this case, useraction.action is found in the root namespace and the request will be forwarded to user.jsp. In case of the first package, there is no namespace defined here. So, if the request is made upon /manager.action, the root namespace will be searched. If found, the root action will be executed, otherwise it will search for the default namespace. Here, the manageraction does not exist in the root namespace, so it will search the default namespace and the default manageraction will be executed.

Include Configuration

Include configuration are very much similar to that of tags that we are using in JSPs. It is used to include files; the included file must always be in the same format to that of struts.xml, as well as the doctype. You can place the included file anywhere on the CLASSPATH. The configuration of include is shown here:

```

<struts>
    <include file="somefile.xml"/>
    <include file="another.xml"/>
    . . .
</struts>

```

In case of a large project, it is useful to use the include files to organize different modules of the application that are being developed and used by different team members.

Interceptor Configuration

In the developing strategy, many actions need to share common things, like some actions require validated input, some require files to be uploaded to be pre-processed, some other may require drop-down lists and other controls to be pre-populated before the page displays. The Struts Framework makes these operations easy by introducing the *Interceptor* concept. It allows you to define code to be executed before and after the execution of an action method. It behaves as a powerful tool in the developing strategy. Every request to some Action class goes through some Interceptors configured for it. Generally, there are many use cases of Interceptor, which include validation, property population, security, logging, debugging and more.

We can create the Interceptor stack by chaining a number of Interceptors. Each Interceptor has a unique class name, as Interceptors are implemented as Java classes. The configuration of registering Interceptor is shown here:

```
<interceptors>
<interceptor name="secure" class="com.kogent.SomeInterceptor"/>

    <interceptor-stack name="secureStack">
        <interceptor-ref name="secure"/>
        <interceptor-ref name="defaultStack"/>
    </interceptor-stack>
</interceptors>
```

The Interceptors can be configured in struts.xml file within a package element using `<interceptors>`, `<interceptor>` elements. An Interceptor stack, which is a collection of Interceptors, can be configured by using `<interceptor-stack>`.

You can place the Interceptor and Interceptor stack in any order while defining the Interceptor stack. One Interceptor stack can have reference of another Interceptor stack also. The framework will invoke each Interceptor on the stack in the order you have defined them. The struts-default.xml file defines all framework Interceptors and default Interceptor stacks, like basicStack, defalutStack, modelDrivenStack, etc. We can make all framework Interceptors available in our package by including struts-default.xml and extending struts-default package. For all action configurations, the required set of Interceptors is configured by using `<interceptor-ref>` element.

Action Configuration

If we take the framework as a whole, the importance of the action element will always be more than the other elements. When the request matches the action's name, the framework uses the mapping to determine how to process the request. An action mapping can contain a set of exception handlers, Interceptor stack, and a set of result types, but the only required element is 'name', others are optional. It is up to the user to use them. The other attributes can also be specified at package scope.

Here's an example of a login action:

```
<action name="login" class="LoginAction">
    <result name="success">menu.jsp</result>
    <result name="input">login.jsp</result>
</action>
```

When the user requests for an action, the request path is matched with the name attribute of all the actions mapped in struts.xml file. The hostname, application name, and the extension from the request URI are dropped, and the remaining string is searched. For example, if the address in your browser ends with `http://localhost:8080/projectname/login.action`, the request maps to the login action. Within an application, a Struts Tag usually generates the link to an action. The tag can specify the action by name, and the framework will render the default extension and anything else that is needed.

You can see a `<s:form>` defining its `action` attribute to a action configured in `struts.xml` file with name `login`, as shown in following code snippet:

```
<s:form action="login">
    <s:textfield label="Please enter your name" name="name"/>
    <s:submit/>
</s:form>
```

This form has action name `login` and its submission will invoke the `LoginAction` class. If your action name is being provided by slashes like `name="manager/login"`, then you need to activate the functionality via `struts.xml` by defining a constant `struts.enable.SlashesInActionNames` with value true.

By default, the first method of the Action class to be executed is its String `execute()` method. This method works as an entry point. Sometimes the user wants to generate more than one entry point to the action. Suppose you want separate the entry point to create, update, and delete. In that case, you can create different methods with signature String `methodName()`. The action can be configured to invoke a specific method other than `execute()` by giving a `method` attribute, which invokes the String `removeEmployee()` method of the Action class as shown here:

```
<action name="delete" class="com.kogent.EmployeeAction" method="removeEmployee ">
```

Another thing left for discussion is Action Default. Suppose an action is requested and the framework is not able to map the request to the respective action name. In that case, generally, the page will show '404 - page not found'. If you would like to give a default action, in case of any unmatched request, you can specify a default action. In case of there is no action to match, the default action will be executed. The syntax for configuring the default action is shown here:

```
<package name="Hello" extends="action-default">
    <default-action-ref name="underConstruction">
        <action name="underConstruction">
            <result>/underConstruction.jsp</result>
        </action>
    </default-action-ref>
</package>
```

Result Configuration

After completion of an Action class method, it returns a String and the value of the String is used to select a result element. An action mapping will often have a set of results representing different possible outcomes. The different types of result tokens that are defined by the `ActionSupport` base class are as follows:

```
String SUCCESS = "success";
String NONE = "none";
String ERROR = "error";
String INPUT = "input";
String LOGIN = "login";
```

A default result type can be set as part of the configuration for each package. In case a package extends another package, the child package can set its own default result or can inherit that from its parent package. Setting of a result type is shown here:

```
<package name="mypackage">
    <result-types>
        <result-type name="dispatcher"
            class="org.apache.struts2.dispatcher.ServletDispatcherResult"
            default="true"/>
        <result-type name="chain"
            class="com.opensymphony.xwork.ActionChainResult"/>
        .
        .
        .
    </result-types>
    .
    .
</package>
```

Different result types to be used can be defined in a similar way. All framework results are defined in struts-default.xml file, which can be included in our strut.xml and the packages defined in struts.xml can extend the struts-default package to make all framework result available to be used. Other results are chain, redirect, redirect-action, stream, freemarker, velocity and others.

The set of results can be configured for each action mapping with `<result>` element. The attributes `name` and `type` are optional for this element. The result can be configured for an action as shown here:

```
<action name="login">
    <result>success.jsp</result>
    <result name="error">error.jsp</result>
    <result name="input">login.jsp</result>
</action>
```

In case of the absence of the `type` attribute, the framework will use `dispatcher` result by default. The default result type, i.e. `dispatcher`, simply forwards to another Web resource, which is usually a JSP/HTML page. If the resource is a JavaServer Page, then the container will render it, using its JSP engine. Similarly, if the `name` attribute is not specified, the framework will take it as ‘success’, by default. Here, we have shown a result element using `name` and `type` attribute and an alternative is shown using defaults for these attributes, as shown in the following code snippet:

```
<result name="success" type="dispatcher">welcome.jsp</result>
OR
<result>welcome.jsp</result>
```

In some applications, you’ll find some results are being applied with multiple actions. For example, we may have a set of actions which can use the same result configured with `name=login` to consequently

authorize the user. Here, the sharing of results among different actions is introduced, which can be implemented by configuring some global results. The framework will search a local request first and if not found, it starts searching for global results. The configuration of global results is as shown here:

```
<global-results>
    <result name="error">/error.jsp</result>
    <result name="login" type="redirect-action">/login.jsp</result>
</global-results>
```

Exception Configuration

The methods of Action class being invoked may throw different types of exceptions. We can handle these exceptions in a traditional way using try/catch block inside our code. Struts 2 provides an easier way of handling exceptions through mapping them to some JSP/HTML page. This way of declarative exceptions handling helps in handling different exceptions thrown by Action class methods in a simple and more manageable way. The exceptions are handled here by automatically catching and mapping them to some pre-defined results. Usually this pattern is useful for the framework, which throws `RuntimeException`. The configuration of exception can be implemented in two ways, such as globally or for a specific action mapping, and it takes two elements—exception and result.

Here's Listing D.6 that shows the configuration of exception handling:

Listing D.6: Exception configuration in struts.xml file

```
<struts>
    <package name="default">
        ...
        <global-results>
            <result name="login">/login.jsp</result>
            <result name="exception">/exception.jsp</result>
        </global-results>

        <global-exception-mappings>
            <exception-mapping exception="java.sql.SQLException"
                result="sqlException"/>
            <exception-mapping exception="java.lang.Exception"
                result="exception"/>
        </global-exception-mappings>
        ...

        <action name="someAction" class="com.kogent.SomeAction">
            <exception-mapping exception="com.kogent.CustomException"
                result="login"/>
            <result name="sqlException">/exception.jsp</result>
            <result>/success.jsp</result>
        </action>
        ...
    </package>
</struts>
```

In the preceding example, the first exception mapping is mapped globally, and the second mapping is for a particular Action class `someAction`. Any exception thrown is searched for the exception mapping to find the result configured for it in the local action mapping first and globally thereafter.

Zero Configuration Applications

Zero configuration Struts 2 applications are those applications, which do not use additional XML and properties file to configure different components in the application like we are doing so far in our applications. Instead, some metadata is expressed using annotations to provide details regarding the Action class to be executed, result to be used, validations, and type conversions. We do not need XML file to configure all these important issues and, hence, such applications are known as *Zero configuration* applications.

Annotations can be defined as extra information, which is associated with a particular document or, in other words, it is a new word for metadata (data of data). In case of Java, annotations can be defined as a code fragment that elaborates upon the described Java classes, methods, and fields. These annotations are implemented as a special kind of interface and can be packaged, compiled, and imported like any other classes. Most of the time, the user wants to give more information to a particular piece of code or more than just a single attribute. This is where, annotation plays an important role. One of the benefits of the new annotation specification is that annotation enables you to create quite complex structures, while maintaining type safety.

Annotation in Struts 2 is generally divided into four different types:

- Action annotation
- Interceptor annotation
- Validation annotation
- Type Conversion annotation

Inside the preceding four types of annotations, some more annotations are defined here one by one.

Action Annotation

These annotations are used, while creating an Action class. There are different annotations available to configure namespace and results for the given action. These Action classes need not to be configured in `struts.xml` for their associated mapping. This results in Zero configuration for actions, i.e. no required configuration for actions. There are different annotations, which help in defining namespace for the action, the parent package to extend from, and results associated with the action. Generally, these annotations are collectively known as Action annotations. Read on to know more about them.

@Namespace Annotation

Its responsibility is to override the namespace of an action. We need not provide action mapping for Action class now when Action annotations are used, i.e their configuration is created automatically, which is based on some naming conventions. The `@Namespace` annotation defines the namespace for automatically configured actions.

@ParentPackage Annotation

It defines an existing configuration package for the action package to extend. An action can extend some other package to reuse the configurations provided there. We can define the parent package for the action using `@ParentPackage` annotation.

@Result Annotation

It defines an action Result. There are four types of parameters available inside the Result and Results annotation. These four parameters are name, type, value and param.

Here's the example which shows @Result defining a single request:

```
@Result(name="success", value="/home.page", type=TilesResult.class)
public class HomeAction extends ActionSupport {
    // ...
}
```

@Results Annotation

It defines multiple results. The @Results annotation helps in defining set of results for a given action. Here's the example, which shows @Results defining multiple results:

```
@Results({
    @Result(name="success", value="/home.page", type=TilesResult.class),
    @Result(name="homeError", value="/homeError.page", type=TilesResult.class)
})
public class HomeAction extends ActionSupport {
    // ...
}
```

These annotations allow definition of Action results, namespace, and parent package in the Action class itself, rather than in XML file. The only required things here are the consideration of some naming conventions and defining of a initial parameter named actionPackage for FilterDispatcher filter. We can set the actionPackages filter init-param to a comma-separated list of packages containing the annotated Action classes, as shown in the following example:

```
<filter>
<filter-name>struts2</filter-name>
<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
<init-param>
<param-name>actionPackages</param-name>
<param-value>com.kogent.action, com.kogent.employee</param-value>
</init-param>
</filter>
```

The location set as actionPackage is searched for the action being invoked. For example, the com.kogent.action.ShowAction Action class can be invoked using request URL <http://localhost:8080/someAppName/show.action>. The results and parent package for show action can simply be configured using @Results, @Result, @ParentPackage annotations.

Interceptor Annotation

These annotations are used to configure different methods of the Action class. This is for making sure that the particular Action class method is invoked before or after the execute() method, thus intercepting the execution path. There are three types of Interceptor annotations:

Appendix: D

- @After annotation
- @Before annotation
- @BeforeResult annotation

To get the functionality of the preceding three Interceptor annotations, you need to specify `AnnotationWorkflowInterceptor` in your Interceptor stack.

@After Annotation

This marks the method which is required to be executed after the result. The method annotated with this annotation is invoked after other Action class method and result execution. The action annotation can be applied at method level.

Here's an example of @After annotation:

```
public class SomeAction extends ActionSupport {

    @After
    public void isValid() throws validationException {
        // validate model object, throw exception if failed
    }

    public String execute() {
        // perform action
        return SUCCESS;
    }
}
```

@Before Annotation

The method having this annotation is invoked before the main action method. Here's an example of @Before annotation:

```
public class SomeAction extends ActionSupport {

    @Before
    public void isAuthorized() throws AuthenticationException {
        // authorize request, throw exception if failed
    }

    public String execute() {
        // perform secure action
        return SUCCESS;
    }
}
```

@BeforeResult Annotation

The method annotated with this annotation will be invoked after the action method, but before the result execution. It can also be applied at method level.

Here's an example of @BeforeResult:

```
public class SomeAction extends ActionSupport {  
  
    @BeforeResult  
    public void isValid() throws validationException {  
        // validate model object, throw exception if failed  
    }  
  
    public String execute() {  
        // perform action  
        return SUCCESS;  
    }  
}
```

Validation Annotation

The Validation annotations help in implementing validation rules on different fields without configuring them in some XML files. Different Validation annotations are provided here for corresponding validation rules. We can use different Validation annotation at method level to validate the data being set. Similar to other annotations, these annotations have their own set of attributes. The names of Validation annotation are self-descriptive for what they are made for. The most frequently used Validation annotations are described here with their syntax:

```
@ConversionErrorFieldValidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true)  
  
@DateRangeFieldvalidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true, min = "2005/01/01",  
    max = "2005/12/31")  
  
@DoubleRangeFieldvalidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true, minInclusive = "0.123",  
    maxInclusive = "99.987")  
  
@Emailvalidator(massage = "Default message",  
    key = "i18n.key", shortCircuit = true)  
  
@IntRangeFieldvalidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true, min = "0", max = "42" )  
  
@RequiredFieldvalidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true)  
  
@RequiredStringValidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true, trim = true)  
  
@StringLengthFieldvalidator(message = "Default message",  
    key = "i18n.key", shortCircuit = true, trim = true,  
    minLength = "5", maxLength = "12")
```

Whenever you want to use annotation-based validation, you have to annotate the class or interface with Validation annotation.

Here's an example, given in Listing D.7, of an Annotated class for @Validation(), @RequiredFieldValidator() and @IntRangeFieldValidator():

Listing D.7: Sample class using Validation annotation

```
@Validation()
public class SimpleAnnotationAction extends ActionSupport {

    @RequiredFieldValidator(type = ValidatorType.FIELD, message =
        "You must enter a value for bar.")
    @IntRangeFieldValidator(type = ValidatorType.FIELD, min = "6", max = "10",
        message = "bar must be between ${min} and ${max}, current value is
        ${bar}.")
    public void setBar(int bar) {
        this.bar = bar;
    }

    public int getBar() {
        return bar;
    }

    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

NOTE

The Struts 2 Validation support has been discussed in Chapter 9.

Type Conversion Annotation

Generally, type conversion for maps and collections using generics is directly supported. Instead of specifying the types found in collections and maps, the collection's generic type is used. By using annotations, an application should be able to avoid the use of any `ClassName-conversion.properties` files. Type Conversion annotations have the following annotations:

- ❑ `@Conversion` annotation
- ❑ `@CreateIfNull` annotation
- ❑ `@Element` annotation
- ❑ `@Key` annotation
- ❑ `@KeyProperty` annotation
- ❑ `@TypeConversion` annotation

@Conversion Annotation

It is a marker annotation for type conversions. This annotation is used at the type level. An example of this annotation is as follows:

```
@Conversion()  
public class ConversionAction implements Action {  
}
```

@CreateIfNull Annotation

This annotation sets the CreateIfNull for type conversion. It must be applied at field level. It has the only parameter that is value. An example of this type of annotation is as follows:

```
@CreateIfNull (value = true)  
private list <User> users;
```

@Element Annotation

It sets elements for type conversion and should be applied at field level. It has the parameter, value, which specifies the element property value. An example of this type of annotation is as follows:

```
@Element( value = com.acme.User )  
private Map<Long, User> userMap;  
@Element( value = com.acme.User )  
public List<User> userList;
```

@Key Annotation

It sets the key for the type conversion. This annotation must be applied at the field level. It has one parameter, value, which specifies the key property value. An example of this type of annotation is as follows:

```
@Key( value = java.lang.Long.class )  
private Map<Long, User> userMap;
```

@KeyProperty Annotation

It sets the key property for type conversion. This annotation must be applied at the field level. This annotation should be used with Generic types, if the key property of the key element needs to be specified. The one parameter that it uses is value, which specifies the key property value. An example of this type of annotation is as follows:

```
@KeyProperty( value = "userName" )  
protected List<User> users = null;
```

@TypeConversion Annotation

This annotation is used for the class and application-wide conversion rules. In case of class-wide conversion, the conversion rules will be assembled in a file called XXXAction-

conversion.properties within the same package as the related Action class. Here you have to set the type parameter as type = ConversionType.CLASS, whereas in case of application-wide conversion, the conversion rules will be assembled within the xwork-conversion.properties file within the CLASSPATH root. You have to set the type parameter as type = ConversionType.APPLICATION. It has the parameters, like key, type, rule, converter, and value.

Here's an example, given in Listing D.8, for this annotation:

Listing D.8: Sample class using TypeConversion annotation

```
@Conversion()
public class ConversionAction implements Action {

    private String convertInt;

    private String convertDouble;
    private List users = null;

    private HashMap keyvalues = null;

    @TypeConversion(type = ConversionType.APPLICATION, converter
        = "com.opensymphony.xwork2.util.XWorkBasicConverter")
    public void setConvertInt( String convertInt ) {
        this.convertInt = convertInt;
    }

    @TypeConversion(converter = "com.opensymphony.xwork2
        .util.XWorkBasicConverter")
    public void setConvertDouble( String convertDouble ) {
        this.convertDouble = convertDouble;
    }

    @TypeConversion(rule = ConversionRule.COLLECTION, converter =
        "java.util.String")
    public void setUsers( List users ) {
        this.users = users;
    }

    @TypeConversion(rule = ConversionRule.MAP, converter =
        "java.math.BigInteger")
    public void setKeyValues( HashMap keyvalues ) {
        this.keyvalues = keyvalues;
    }

    @TypeConversion(type = ConversionType.APPLICATION, property = "java.util.Date",
        converter = "com.opensymphony.xwork2.util.XWorkBasicConverter")
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

You must have understood the rich set of annotations available in Struts 2, which can be used with the code to consequently reduce the declaration in configuration files. Now, we can take you to the real implementation of the different configuration issues and concepts through an applications.

Implementing Struts 2 Application Configuration

Here we are going to explain the configuration of some important elements in the `struts.xml` file with the help of an application. We would develop a simple Struts 2 Web application in order to understand the configuration settings of different elements in the `struts.xml` file. The name of our Web application is `strutsproject`. This application consists of the following components:

- ❑ **Two JSP files**—`index.jsp` and `success.jsp`
- ❑ **An Action class**—`DisplayAction.java`
- ❑ **Web application Deployment Descriptor**—`web.xml`
- ❑ **Struts 2 configuration file**—`struts.xml`
- ❑ **Two properties files**—`ApplicationResources.properties` and `struts.properties`

The details of all these components, along with their source codes, are covered here. In order to deploy the `strutsproject` Web application on your Web server follow the similar steps as taken for previous applications in the book.

We'll now explain the preceding mentioned components one by one.

Here we have created two JSP pages, namely `index.jsp` and `success.jsp`. The `index.jsp` is our first page where the user enters his information. The `index.jsp` page provides a form with input fields `username` and `city`.

Here's the code, given in Listing D.9, for the `index.jsp` page (you can find `index.jsp` file in `Code\Appendix D\strutsproject` folder in CD):

Listing D.9: `index.jsp`

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>

<html>
    <head>
        <title>Personal Info</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>

    <body>
        <h2><s:text name="app.welcome"/></h2>
        <s:form action="displayAction">
            <s:textfield key="app.username" name="username" />
            <s:textfield key="app.city" name="city" />
            <s:submit key="button.save"/>
        </s:form>
    </body>
</html>
```

Now, save `index.jsp` in the root directory of your application. Look at the keys, which are used in the fields, and these keys are set to some message values in the `ApplicationResources.properties` file.

Appendix: D

Now, we'll give you the coding for the success.jsp page, which is our final page on which the result will be displayed to the user. Here's the code, given in the Listing D.10, for success.jsp (you can find success.jsp file in Code\Appendix D\strutsproject folder in CD):

Listing D.10: success.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>Personal Info.</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h2><s:text name="app.welcome"/></h2>
    Your Name: <b><s:property value="username"/></b><br><br>
    Your City: <b><s:property value="city"/></b><br><br>
    <s:a href="index.jsp">Back</s:a>
</body>
</html>
```

This file will also be stored at the same location of index.jsp. These are two views that we need for our Web application. Now, we'll show you the details of the two properties files used in the application. The first file to be mentioned is ApplicationResources.properties.

Here's the source code, given in Listing D.11, for ApplicationResources.properties (you can find ApplicationResources.properties file in Code\Appendix D\strutsproject\WEB-INF\classes\com\kogent folder in CD):

Listing D.11: ApplicationResources.properties

```
# Resources for parameter 'com.kogent.ApplicationResources'
# Project strutsproject
app.title=Struts 2 Application
app.welcome=Personal Information
app.username=User Name
app.name=Enter Name
app.age=Enter Age
app.city=City
button.save=Save
```

Observe the code given in the file in Listing D.11; we have set all the key values that we have used inside the index.jsp and success.jsp. The second properties file that we are using in our application is struts.properties. This file contains only a single line of code:

```
struts.custom.i18n.resources=com.kogent.ApplicationResources
```

The struts.properties file is placed in WEB-INF/classes folder and the ApplicationResources.properties file is placed in WEB-INF/classes/com/kogent folder.

Now, we'll create our Action class DisplayAction.java. Listing D.12 shows the source code for DisplayAction.java class (you can find DisplayAction.java file in Code\Appendix D\strutsproject\WEB-INF\src\com\kogent folder in CD):

Listing D.12: DisplayAction.java

```

package com.kogent;

import com.opensymphony.xwork2.ActionSupport;
public class DisplayAction extends ActionSupport {

    private String username;
    private String city;
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String execute() throws Exception {
        if("".equals(username))
            username="Blank";
        if("".equals(city))
            city="Blank";
        return SUCCESS;
    }
}

```

The `DisplayAction.java` file is our Action class. This file contains a `execute()` method, which returns a String type value `SUCCESS`. The file also contains getters and setters for the `username` and `city` fields.

Next, we move on to the most important phase of the development of a Web application, i.e. the configuration of a Web application. The first configuration file that we are going to discuss is the `web.xml` file. As mentioned in the earlier chapters of the book, the `web.xml` file for Struts 2 based application needs only two configuration settings—filters and filter-mapping. In our `web.xml` file, we have one more entry for welcome-file-list, which gives the name of the welcome file.

Here's Listing D.13 showing the `web.xml` for strutsproject application (you can find `web.xml` file in `Code\Appendix D\strutsproject\WEB-INF` folder in CD):

Listing D.13: web.xml file

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>Struts 2 Application</display-name>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    
```

```
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Now, let's discuss the most important file for any Struts 2 based application. This is the `struts.xml` file; it defines the configuration settings of different components of Struts 2.

Here's the code, given in Listing D.14, showing the `struts.xml` file for our sample Web application (you can find `struts.xml` file in `Code\Appendix D\strutsproject\WEB-INF\classes` folder in CD):

Listing D.14: struts.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>

    <package name="default" >
        <result-types>
            <result-type name="dispatcher"
class="org.apache.struts2.dispatcher.ServletDispatcherResult"/>
        </result-types>

        <interceptors>
            <interceptor name="exception"
class="com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor"/>
            <interceptor name="servlet-config"
class="org.apache.struts2.interceptor.ServletConfigInterceptor"/>
            <interceptor name="prepare"
class="com.opensymphony.xwork2.interceptor.PrepareInterceptor"/>
            <interceptor name="checkbox"
class="org.apache.struts2.interceptor.checkbox.Interceptor" />
            <interceptor name="params"
class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>
            <interceptor name="conversionError"
class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"/>

            <interceptor-stack name="MyBasicStack">
                <interceptor-ref name="exception"/>
                <interceptor-ref name="servlet-config"/>
                <interceptor-ref name="prepare"/>
                <interceptor-ref name="checkbox"/>
                <interceptor-ref name="params"/>
                <interceptor-ref name="conversionError"/>
            </interceptor-stack>
        </interceptors>
    </package>
</struts>
```

```
<action name="displayAction" class="com.kogent.DisplayAction">
    <interceptor-ref name="MyBasicStack"/>
    <result name="success" type="dispatcher">
        <param name="location"/>/success.jsp</param>
    </result>
</action>
</package>
</struts>
```

We'll discuss the configuration of the different elements of struts.xml file one by one. The important thing about this file is its placing. It should be placed inside the WEB-INF/classes directory of your Web application. The web.xml file will be stored, as usual, inside the WEB-INF directory of your Web application.

Now that all the coding related to our strutsproject application is over, it is time to run our Web application. The output of index.jsp page is as shown in Figure D.1.



Figure D.1: The index.jsp page showing some input fields.

The users need to enter information in the required fields inside the index.jsp. On clicking the 'Save' button, this information is submitted and the final output will be shown in the success.jsp page. Suppose you have entered your username as 'Kogent' and the city as 'Delhi'. In that case, the output will be like the one shown in Figure D.2.

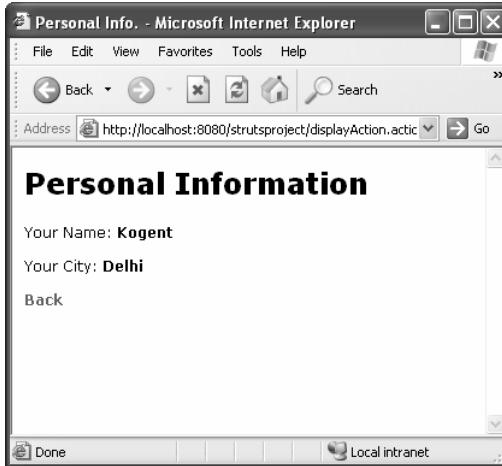


Fig D.2: The success.jsp page showing details passed in index.jsp page.

Based on the preceding application, let's understand the configuration of the following elements present in the struts.xml file:

- Package configuration
- Interceptor configuration
- Action configuration
- Result configuration

In addition to these configurations, we'll also understand the include, namespace, and exception configuration. So let's start our discussion.

Implementing Package Configuration

As discussed earlier in this chapter, packages are used to group the various configuration elements, such as actions, result-types, interceptors, etc. into a single logical configuration unit. It means all the configuration elements present in the struts.xml file are configured under the opening and closing tags of the package element. The package element has four attributes, which are discussed in the earlier part of this chapter.

In our example, we have mentioned only one attribute for the package element. This attribute is name, which is the only compulsory attribute out of the four attributes available with the package element. The configuration of the package element in the struts.xml file, given in the Listing D.14, has the name attribute equal to default. It is under this package name that we are configuring other results, Interceptors, and actions. The basic structure package configuration looks like the following code:

```
<package name="default">
    . . .
    . . .
</package>
```

In our example, we have given only one attribute for the package element, i.e. the name of the package. We have used only one attribute because in this particular example we have given configuration of all the elements directly in a single file. In case, we used the configuration settings of the struts-

default.xml file by including this file in the struts.xml (Include configuration will be discussed later this chapter), we need another attribute `extends` to be mentioned in the opening `<package>` tag. If we mention the `extends` attribute, then the opening package tag will look like the following code:

```
<include file="struts-default.xml"/>
<package name="default" extends="struts-default">
    . . .
    . . .
</package>
```

This line tells that the package default is extending all the configurations given in struts-default package, which is defined in struts-default.xml file. The struts-default.xml is loaded from struts2-core-2.0.6.jar, which must be present in your WEB-INF/lib folder. By inheriting the struts-default package, this package is able to use framework results and Interceptors of the struts-default package. This package contains all the result types, Interceptors, Interceptor stacks, which are the basic requirements for a Struts 2 application.

Implementing Interceptor Configuration

In this section, we'll discuss the practical aspects of the configuration of Interceptors. Interceptors are one of the most important components of the Struts 2 Framework. Interceptors are used both before and/or after the execution of the Action class. The configuration of different Interceptors is done by using the `<interceptors>` element. The configuration of Interceptors in the struts.xml file comprises of the configuration of two elements, first the Interceptors and then the configuration of Interceptor stack.

When we configure the Interceptors, we usually group different Interceptors (in some cases we may add some Interceptor stack also) in the form of a stack. The purpose behind this formation of Interceptor stack is that it can easily be referenced in the configuration settings of other elements, i.e. without giving the configuration of individual Interceptors. The configuration settings of different Interceptors, which are included in the Interceptor stack, should be given before the declaration of the stack in the struts.xml. The Struts 2 Framework will invoke each Interceptor on the stack in the order it is defined.

Observe the struts.xml file carefully. The configuration of Interceptors is given between the `<interceptors>` and `</interceptors>` tags. We have defined our own Interceptor stack. The name of this stack is MyBasicStack containing six Interceptors. The configuration of these six Interceptors is given before the definition of the stack. The individual Interceptors have two attributes—name and class—as shown in the Listing D.14. If our package does not extend struts-default package from struts-default.xml, we need to configure all results and Interceptors in our struts.xml file.

Here's the code, given in Listing D.14, showing the result and Interceptor configuration:

```
<result-types>
    <result-type name="dispatcher"
        class="org.apache.struts2.dispatcher.ServletDispatcherResult"/>
</result-types>

<interceptors>
    <interceptor name="exception"
        class="com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor"/>
```

```
<interceptor name="servlet-config"
class="org.apache.struts2.interceptor.servletConfigInterceptor"/>
. . .
<interceptor-stack name="MyBasicStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="servlet-config"/>
. . .
</interceptor-stack>
</interceptors>
```

Implementing Action Configuration

The configuration setting of the Actions in the Struts 2 Framework is mainly done by using the action mappings. These are the basic building structure for the configuration of Actions in the Struts Framework. Essentially, the action maps an identifier to a handler class. Actions are used for handling of the request, and processing of data to generate the output. When a request matches the action's name, the framework uses the mapping to determine the procedure of processing the request.

Action mapping can specify a set of result types, a set of exception handlers, and an Interceptor stack. The only required attribute is the name attribute. The other attributes can also be specified at the package scope. Every action present in our application is configured separately in the struts.xml file.

In our sample Web application, strutsproject, we have a single action displayAction. The configuration of this action includes the mapping of our action with the required result-type. The action element has two attributes—name and class. The name attribute gives the name of the action, and class attribute gives the name of the Action class to be executed. The <interceptor-ref> gives the reference to the Interceptor stack used in the configuration of action. The <result> element gives the type of the result used to generate the output. For example, in Listing D.14, the result-type is dispatcher and the interceptor-ref is MyBasicStack.

The action mapping from **Listing D.14** is shown here:

```
<action name="displayAction" class="com.kogent.DisplayAction">
    <interceptor-ref name="MyBasicStack"/>
    <result name="success" type="dispatcher">
        <param name="location">/success.jsp</param>
    </result>
</action>
```

Implementing Result Configuration

In our application, we have used only a single result type dispatcher. Therefore, we only need to configure this dispatcher result type in the struts.xml file. The various result-types are configured under the opening and closing tags of <result-types> element. We can configure any number of result-types in a single struts.xml file. The result-type element has two attributes—name and class of the result. The result type configuration defined in our struts.xml file is as follows:

```
<result-types>
    <result-type name="dispatcher"
```

```
        class="org.apache.struts2.dispatcher.ServletDispatcherResult"/>
</result-types>
```

We can use a number of result-types in our application depending upon the requirement of our Web application. A number of inbuilt result-types are provided with the Struts 2 package.

NOTE

The different types of results and their configuration have been discussed in the Chapter 8 of this book.

Implementing Include Configuration

Before going to the depth of the include configuration, let's consider the `struts.xml` file for the preceding application once again. Let's replace the `struts.xml` file for the previous application with the one shown in Listing D.15.

Listing D.15: struts.xml using struts-default package

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <action name="FirstAction" class="FirstAction">
            <result>/Success.jsp</result>
        </action>
    </package>
</struts>
```

We have included `struts-default.xml` file to our `strut.xml`. The `struts-default.xml` file contains the configuration for Interceptors, Interceptor stacks, and result-types. When we include this `struts-default.xml` file in our `struts.xml` file, we also need to extend our package from the `struts-default` package using the `extends` attribute, as shown in the Listing D.15.

If you compare both the `struts.xml` files given in Listing D.14 and Listing D.15, respectively, you may clearly observe that the later one is precise and compact, and easily understandable. The `struts.xml` file, given in the Listing D.15, is also the standard way of writing this file. The `struts.xml` file, given in Listing D.14, gave you an idea about the various configuration elements present in the `struts.xml` configuration file.

NOTE

The users are advised to use the standard notation, given in Listing D.15, in their application for developing the struts.xml file.

Implementing Namespace Configuration

In enterprise-level applications, you'll find that more than one team is working on different modules, independently of each other. The main motto is to give different Namespaces to different team. As their work involves hundreds of pages or more, Struts configuration is designed in the concept of 'packages' and 'namespace'.

Appendix: D

Each package can set their own defaults, including a namespace setting format as shown here:

```
<struts>
    <package name="example" namespace="/example" extends="struts-default">

        <action name="someAction" class="example.SomeAction">
            <result>/example/success.jsp</result>
        </action>

    </package>
</struts>
```

We can give some Namespace for our package, say after providing /example as the Namespace attribute value the package defined in Listing D.14 will look like the one shown here:

```
<struts>

    <package name="default" namespace="/example">
        <result-types>..</result-types>

        <interceptors>
            . . .
        </interceptors>

        <action name="displayAction" class="com.kogent.DisplayAction">
            <interceptor-ref name="MyBasicStack"/>
            <result name="success" type="dispatcher">
                <param name="location">/success.jsp</param>
            </result>
        </action>
    </package>
</struts>
```

The action `displayAction` can be invoked using url `/example/displayAction.action`, instead of `/displayAction.action`.

Implementing Exception Configuration

The exception, thrown during the Action method, can be automatically caught and mapped to a predefined Result. This exception handling helps in avoiding the user getting stack trace of the exception caused. Instead, the user now gets a simple JSP page describing some of the problems and hyperlinks to escape the scenario. Let's create a simple Action class with its `execute()` method throwing a `NullPointerException`.

Here's the code, given in Listing D.16, for `MyAction.java` (you can find `MyAction.java` file in `Code\Appendix D\strutsproject\WEB-INF\src\com\kogent` folder in CD):

Listing D.16: MyAction.java

```
package com.kogent;
import com.opensymphony.xwork2.ActionSupport;
public class MyAction extends ActionSupport {
```

```
public String execute() throws Exception {  
    throw new NullPointerException();  
}
```

Now add the following syntax in your struts.xml enabling your application to handle exceptions thrown by MyAction Action class. Now add a global result and provide an action mapping for MyAction Action and see the use of <exception-mapping> element, as shown in Listing D.17.

Listing D.17: Configuring exceptions in struts.xml file

```
<struts>  
  
    <package name="default" namespace="/example">  
        <result-types>..</result-types>  
  
        <interceptors>  
            . . .  
        </interceptors>  
  
        <global-results>  
            <result name="exception" type="dispatcher"/>/exception.jsp</result>  
        </global-results>  
  
        <action name="displayAction" class="com.kogent.DisplayAction">  
            . . .  
        </action>  
  
        <action name="myaction" class="com.kogent.MyAction">  
            <exception-mapping exception="java.lang.Exception" result="exception"/>  
                <interceptor-ref name="MyBasicStack"/>  
                <result name="success" type="dispatcher"/>/success</result>  
            </action>  
  
    </package>  
</struts>
```

Observe that the <exception-mapping> element has two attributes—exception and result. On the occurrence of some exception, the global result exception will be executed. Create exception.jsp page first which is to be shown in case some exception (java.lang.Exception or other subclasses of it like java.lang.NullPointerException) is thrown during the execution of MyAction class.

Here's the simple JSP page, given in Listing D.18, for exception.jsp (you can find exception.jsp file in Code\Appendix D\strutsproject folder in CD):

Listing D.18: exception.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>  
<html>  
<head>  
    <title>Exception...</title>  
    <link rel="stylesheet" href="mystyle.css" type="text/css" />  
</head>
```

Appendix: D

```
<body>
    <h2>Some Exception!</h2>
    Some Exception has caused you to see this page.
</body>
</html>
```

Now execute the MyAction Action class by entering URL <http://localhost:8080/strutsproject/myaction.action> in the address bar of your browser after redeploying your application to reflect changes. This will display the exception.jsp page, as shown in Figure D.3.



Figure D.3: The exception.jsp showing message.

Using Annotations

The user of annotation has reduced the need of configurations provided in the XML files. All types of different configuration details can automatically be generated by using annotation for results, intercepting methods, validation rules, etc. Let us design an Action class using some annotations and find what additional configuration details are required to execute it.

We can create zero configuration Struts application, which do not need any additional XML configuration. But, there should be some procedure or some other implementation, which helps the controller in finding proper Action class that is not configured in `strut.xml` file. The solution is to set a filter `init` parameter in `web.xml`, named `actionPackage`, to a comma-separated list of packages, which contains Action classes. Now change your `web.xml` for this additional `init-param`, as shown here:

```
<web-app ... . . . >
    .
    .
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
        <init-param>
            <param-name>actionPackages</param-name>
            <param-value>com.kogent.action</param-value>
        </init-param>
    </filter>
```

```
</filter>
<filter-mapping>
. . .
</filter-mapping>
. . .
</web-app>
```

The package `com.kogent.action` will be scanned for the classes, which implements Action interface or ends with Action. If the Action class has a suffix Action, it is dropped before creating the action name for action mapping for this. For example, the `com.kogent.action.ShowAction` Action class can be accessed using `http://localhost:8080/strutsproject/show.action`. Any further sub-package, created inside the `com.kogent.action` package, will be treated as namespace for action mapping.

The Action class, which we are going to create here, is `ShowAction`. The action will not be configured in `struts.xml` file and all the configuration details required will be given by using an annotation. Here's the code, given in Listing D.19, for `ShowAction.java` (you can find `ShowAction.java` file in `Code\Appendix D\strutsproject\WEB-INF\src\com\kogent\action` folder in CD):

Listing D.19: `ShowAction.java`

```
package com.kogent.action;
import org.apache.struts2.config.Result;
import org.apache.struts2.config.Results;
import org.apache.struts2.dispatcher.ServletDispatcherResult;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.validator.annotations.*;

@Results({
    @Result( name="success", value="/user_success.jsp",
        type=ServletDispatcherResult.class),
    @Result( name="input", value="/user.jsp", type=ServletDispatcherResult.class)
})
public class ShowAction extends ActionSupport{
    String name;
    int age;

    @IntRangeFieldValidator(type = ValidatorType.FIELD,min="10", max="30",
        message = "Allowed Age is between 10 and 30.")
    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    @RequiredStringValidator(type = ValidatorType.FIELD, message =
        "Name field can't be empty.")
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```
public String execute(){
    return "success";
}
```

Compile and get ShowAction.class file in the WEB-INF/classes/com.kogent/action package. The Action class is ready to be used without any action mapping done in struts.xml file. The two results have been declared here using @Results and @Result annotations. The Action class implements some validation annotations at method level. We'll see their consequences, while running the application. Design two JSP pages to use this Action class. The first JSP page is user.jsp, which gives two input fields 'name' and 'age'.

Here's the code, given in Listing D.20, for user.jsp (you can find user.jsp file in Code\Appendix D\strutsproject folder in CD):

Listing D.20: user.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>User Info</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h2>User Information</h2>
    <s:form action="show">
        <s:textfield key="app.name" name="name" />
        <s:textfield key="app.age" name="age" />
        <s:submit key="button.save"/>
    </s:form>
</body>
</html>
```

See the action attribute of the <s:form> tag which is set to show. The submission of this form will invoke the action having name="show" with URL <http://localhost:8080/strutsproject/show.action>. This will invoke the ShowAction Action class. The user.jsp can be accessed by using the URL <http://localhost:8080/strutsproject/user.jsp> and its output is shown in Figure D.4.



Figure D.4: The user.jsp showing two input fields.

Design another JSP page which will be shown when the Action class returns success. We have configured two results for the action—user_success.jsp on ‘success’ and user.jsp on ‘input’. On the occurrence of some validation error, the action returns INPUT and the error is displayed in the user.jsp page, as shown in Figure D.5.



Figure D.5: The user.jsp showing validation error messages.

The page, which is displayed on the submission of form with valid data in both the fields is user_success.jsp page.

Here's the code, given in Listing D.21, for user_success.jsp page (you can find user_success.jsp file in Code\Appendix D\strutsproject folder in CD):

Listing D.21: user_success.jsp

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>User Info.</title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
    <h2>User Information</h2>
    Your Name: <b><s:property value="name"/></b><br><br>
    You are <b><s:property value="age"/></b> year old.<br><br>
    <s:a href="user.jsp">Back</s:a>
</body>
</html>
```

Enter a valid entry in the fields, shown in Figure D.5, say ‘John’ as name and ‘25’ as age to display the output screen like the one shown in Figure D.6.

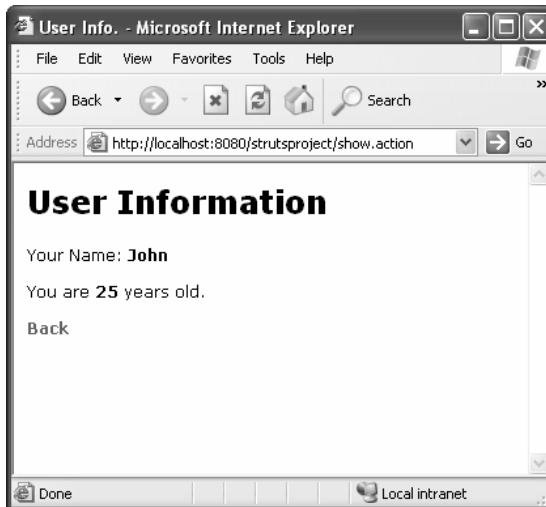


Figure D.6: The user_success.jsp showing name and age entered.

By using annotation, we have invoked an Action class and implemented different validation rules on different fields of Action class. The application created using ShowAction Action class, and two JSP pages—user.jsp and user_success.jsp—is configured fully using annotation without adding any configuration details in your struts.xml file. With this you must have got an idea of zero configuration applications using Struts annotations.

In this chapter, we discussed different configuration details related to a Struts 2 based application, different components associated—Action class, results, Interceptors, constants, and beans, etc.—and the configuration done in struts.xml, web.xml, strut.properties and other default files—struts-default.xml and default.properties.



E

Type Conversion in Struts 2

We know that the HTTP protocol, HTML and the Servlet specifications doesn't deal with the concept of data types. In a Web application, everything is transferred in the form of String or an array of Strings from the client to server. Although this makes the specification simpler, it leaves work for the developer to convert this String data to another required data type. In this appendix, we'll discuss how Struts 2 Type Conversion support solves this problem of developers so that they can focus only on the Business logic, which makes the application development faster and easier. Before discussing about type conversion, let's first discuss why type conversion is necessary.

Need of Type Conversion

Let's suppose that you create a user interface without any type conversion support. The user interface is a JSP page which prompts the user to enter his username, age, and birthdate. This page is then submitted to the server using the http POST method. The http POST request has no information about the type of data each input has. Since the data type of username is type String, there is no need for converting it to String type. Let's consider the other input values, such as age, and birthdate. The values are entered in String form and therefore, it is necessary to convert these data types into proper data types, e.g. age must be converted to int (or Integer) and birth date into Date data type. Let's create an Action without type conversion support that will handle this request.

Here's the code, given in Listing E.1, for the action ActionwotypeConver.java:

Listing E.1: ActionwotypeConver.java

```
import java.util.Date;
import java.text.DateFormat;
import com.opensymphony.xwork2.ActionSupport;
public class ActionwotypeConver extends ActionSupport {
    String username;
    String age;
    String birthdate;
    public String execute () throws Exception {
```

Appendix: E

```
int convertAge = Integer.parseInt (age);
DateFormat df = DateFormat.getDateInstance (DateFormat.SHORT);
Date convertBirthDate = df.parse (birthdate);

// do business logic

return SUCCESS;
}
//getter and setter methods.
}
```

Listing E.1 shows that we have to explicitly convert String data types into their proper data types format, if our action doesn't support type conversion features. Now let's take a look at the view displaying the entered values.

Here's the code, given in Listing E.2, for view_wo_type.jsp:

Listing E.2: view_wo_type.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>
<%@ page import="com.opensymphony.xwork2.ActionContext, java.util.Date,
java.text.DateFormat"%>
<html>
<head>
<title>Login Details</title>
</head>
<body>
Your Login Information
<p/>
Username: <s:property value="username"/><br/>
Age: <s:property value="age"/><br/>
Birth Date: <s:property value="birthdate"/><br/>
<%
Date birthDate = (Date)
ActionContext.getContext ().getvaluestack () .
findValue ("birthdate");
DateFormat df =
DateFormat.getDateInstance(DateFormat.SHORT);
%>
Birth Date: <%= df.format (birthdate) %><br/>
</body>
</html>
```

Listing E.2 shows that while extracting values from the ValueStack we have to explicitly convert data types into String data types format if our action doesn't support type conversion features. We retrieve the birthdate from the Value Stack as date type and then convert it into String type.

Now we'll discuss how Struts 2 Type Conversion solves the problem of explicit conversion from String to other data type or vice versa and automatically does the conversion from String data type to other data types or vice versa. This makes the developers write lesser code and, thus, make application development easier.

Let's first look at our action. So rewrite the action, but this time our action would use the Struts 2 Type Conversion support.

Here's the code, given in Listing E.3, for the action ActionwithtypeConver.java:

Listing E.3: ActionwithtypeConver.java

```
import com.opensymphony.xwork2.ActionSupport;
public class ActionwithtypeConver extends ActionSupport {
    String username;
    String age;
    String birthdate;

    public String execute () throws Exception {

        // do business logic

        return SUCCESS;
    }
    //getter and setter methods.
}
```

As you can see, all the lines that are used for preparing the data are deleted. There is no need to explicitly convert one date type to another. This is automatically done by Struts 2 Type Conversion. The developers have to focus only on the Business logic.

The same things happen when we retrieve the value from the Value Stack. Let's rewrite the view but this time our view would use Struts 2 Type Conversion support.

Here's the code, given in Listing E.4, for the view view_with_type.jsp:

Listing E.4: view_with_type.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>
<%@ page import="com.opensymphony.xwork2.ActionContext, java.util.Date,
java.text.SimpleDateFormat"%>
<html>
<head>
<title>Login Details</title>
</head>
<body>
Your Login Information
<p/>
Username: <s:property value="username"/><br/>
Age: <s:property value="age"/><br/>
Birth Date: <s:property value="birthdate"/><br/>
Birth Date: <%= df.format (birthdate) %><br/>
</body>
</html>
```

As you can see, all the lines that are used for converting the date type to String type are deleted. The conversion is automatically done by the Struts 2 Type Conversion.

Type conversion occurs whenever Struts 2 attempts to apply a value to a type that it cannot convert. In our example, all the conversions are handled automatically because they are performed by the built-in type converters that come with Struts 2. For other conversions, such as Email to String type or vice versa, you have to create your own custom converters.

Struts 2 Built-in Type Conversion

Struts 2 performs the most common type conversion automatically. This is done by the built-in type conversion that comes with Struts 2. This includes support for converting String data type to and from for each of the following types.

- ❑ **String**—As all values submitted by the form are in the form of String arrays, this is the most common conversion. It includes conversion of String arrays into String and vice versa.
- ❑ **Primitives**—Primitives types include integer, double, float, char, and so on. These are also handled automatically.
- ❑ **Date**—Dates are automatically converted for both input and output using the SHORT format. In United States, this format is MM/dd/yyyy.
- ❑ **Array**—Individual strings can be converted to individual items of an Array.
- ❑ **Collections**—Conversion to a collection is done automatically. Struts 2 takes all the String array values, creates a new ArrayList of the same size and places all the values in the ArrayList.

So far, we discussed how Struts 2 built-in type conversion converts String data type to other types, and vice versa. Now, in the next section, we'll discuss how you can create your own type converter using Struts 2 Framework.

Creating a Custom Type Converter

Type conversion is really useful in situations where we want to convert a String into a complex object. For example, if we prompt a user to enter the edge of square in the form of ‘S:e?’, where ? is any number, then you will require Struts 2 to do the conversion—both from String to edge and from edge to String. For this *square* example, you have to configure your converter. To configure a converter, you have to create a property file which corresponds to your action. In this case, if your action name is ‘MyAction’ then it has a corresponding ‘MyAction-conversion.properties’ file. This file should be placed in the same package as the ‘MyAction’ class.

Let's start creating our custom converter for the *square* example just discussed. For that first we'll create a view which prompts the user to enter edge of Square in the form of ‘S:e?’, where ? is an integer.

Here's the code, given in Listing E.5, for the view, index.jsp.(You can also find index.jsp file in Code\Appendix E\Struts2TypeConverter folder in CD):

Listing E.5: index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>

    <head>
        <title>Struts 2 Custom TypeConverter</title>
    </head>

    <body>
        <hr/>
        <h4>Enter Edge (String) of Square to convert it into Square object</h4>
        <s:form action="CustomTypeConverter" method="POST">
            <s:textfield name="square" label='Square "S:e?"'/>
            <s:submit/>
        </s:form>
    </body>
</html>
```

```
</s:form>
<hr/>

</body>

</html>
```

After creating the view, we'll create a Java Bean class which represents our Square object. The class Square.java is a simple Java Bean, which has one property field 'edge' and getter and setter method for this field.

Here's the source code, given in Listing E.6, for Square.java (You can also find Square.java file in Code\Appendix E\Struts2TypeConverter\WEB-INF\src\com\kogent folder in CD):

Listing E.6: Square.java

```
package com.kogent;

public class Square {

    private int edge;

    public int getEdge() {
        return edge;
    }

    public void setEdge(int edge) {
        this.edge = edge;
    }
}
```

After creating a class in which the String is to be converted, we'll create our converter class. The converter class is used for converting String to Square type and from Square to String type as well. Therefore, we have to split the conversion method in two parts: one that converts String type into Square type and one that converts Square type into String type. The converter class contains another function for parsing the String entered by the user, i.e. if the user does not enter the value in the required format, then the converter class returns a TypeConversionException exception.

Here's the code, given in Listing E.7, for the converter class, SquareTypeConverter.java (You can also find SquareTypeConverter.java file in Code\Appendix E\Struts2TypeConverter\WEB-INF\src\com\kogent folder in CD):

Listing E.7: SquareTypeConverter.java

```
package com.kogent;
import java.util.Map;
import org.apache.struts2.util.StrutsTypeConverter;
import com.opensymphony.xwork2.util.TypeConversionException;

public class SquareTypeConverter extends StrutsTypeConverter {
    public Object convertFromString(Map context, String[] values, Class toclass)
    {
        String userString = values[0];
```

Appendix: E

```
        Square newSquare = parseSquare( userString );
        return newSquare;
    }

    public String convertToString(Map context, Object o) {

        Square square = (Square) o;
        String userString = "S:e" + square.getEdge();

        return userString;
    }

    private Square parseSquare( String userString ) throws TypeConversionException
    {
        Square square = null;

        int edgeIndex = userString.indexOf('e') + 1;
        System.out.println("userString = " + userString);

        if (!userString.startsWith( "S:e" ) )
            throw new TypeConversionException ( "Invalid Syntax");

        int edge;
        try {
            edge = Integer.parseInt( userString.substring( edgeIndex ) );
        }catch ( NumberFormatException e ) {
            throw new TypeConversionException ( "Invalid Integer Value for Edge"); }

        square = new Square();
        square.setEdge( edge );

        return square;
    }
}
```

The SquareTypeConverter class extends the StrutsTypeConverter class and implements two methods—one for converting from String to Square (convertFromString) and one for converting from Square to String (convertToString). The class also has a method parseSquare(), which is used for parsing the String value entered by the user.

Now we'll create our action class. The name of our Action class is CustomTypeConverter.java and it defines a field 'edge' of type Square along with its getter and setter methods. Here's the source code, given in Listing E.8, for our action (You can also find CustomTypeConverter.java file in Code\Appendix E\Struts2TypeConverter\WEB-INF\src\com\kogent folder in CD):

Listing E.8: CustomTypeConverter.java

```
package com.kogent;

import com.opensymphony.xwork2.ActionSupport;

public class CustomTypeConverter extends ActionSupport {
```

```

public String execute(){
    System.out.println("Square = " + square.getEdge() );
    return SUCCESS;
}

/* Square Property */

private Square square;

public Square getSquare() {
    return square;
}

public void setSquare(Square square) {
    this.square= square;
}

}

```

Next, we'll configure our converter corresponding to the action class. The name of our action is CustomTypeConverter.java and we'll create a property file with the name CustomTypeConverter-conversion.properties, as already discussed.

Here's the source code, given in Listing E.9, for this property file (You can also find CustomTypeConverter-conversion.properties file in Code\Appendix E\Struts2TypeConverter\WEB-INF\classes\com\kogent folder in CD):

Listing E.9: CustomTypeConverter-conversion.properties

```
square=com.kogent.SquareTypeConverter
```

The entry defined in this file tells where the converter is placed.

Next, we'll create a view, which simply displays that our custom converter is working successfully. The name of the view is success.jsp. Here's the code, given in Listing E.10, for success.jsp (You can also find success.jsp file in Code\Appendix E\Struts2TypeConverter folder in CD):

Listing E.10: success.jsp

```

<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>

    <head>
        <title>Custom TypeConverter Successfully Implemented</title>
    </head>

    <body>
        <hr/>
        <h4>Congratulations! You have used a custom converter to create a
        Square object
        from a string and back to a string. </h4>

```

Appendix: E

```
<p> You created a square with edge equal to <s:property  
value="square.edge"/></p>  
  
Just to check the outgoing data conversion, here's the square back  
in the string syntax <s:property value="square"/>  
<hr/>  
</body>  
  
</html>
```

Now we'll create the struts configuration file struts.xml, which provides mapping for our action. Here's the source code, given in Listing E.11, for struts.xml (You can also find struts.xml file in Code\Appendix E\Struts2TypeConverter\WEB-INF\classes folder in CD):

Listing E.11: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE struts PUBLIC  
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  
"http://struts.apache.org/dtds/struts-2.0.dtd">  
  
<struts>  
    <include file="struts-default.xml"/>  
    <package name="default" extends="struts-default">  
        <action name="CustomTypeConverter" class="com.kogent.CustomTypeConverter">  
            <result>/success.jsp</result>  
            <result name="input">index.jsp</result>  
        </action>  
    </package>  
</struts>
```

When you run the application, the index.jsp page will be displayed as shown in Figure E.1.

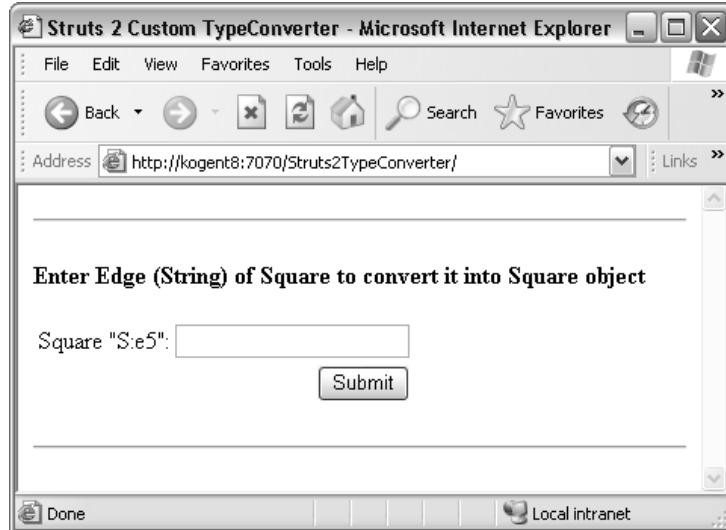


Figure E.1: Converting String to Square Object.

Now enter the value ‘S:e28’ in the textbox and press the ‘Submit’ button. On pressing the ‘Submit’ button, Figure E.2 will be displayed.

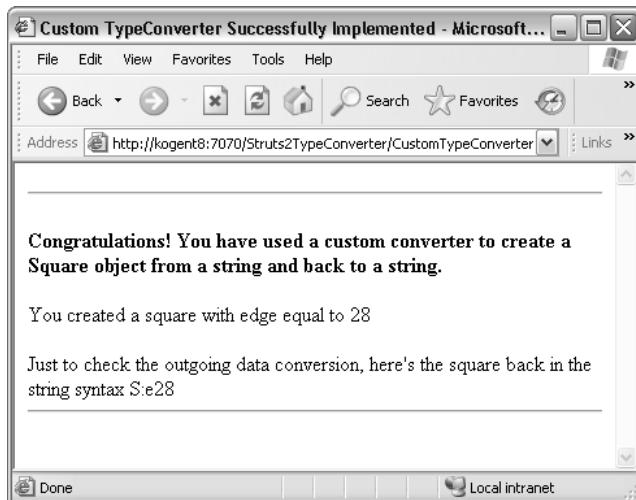


Figure E.2: Successful Creation of Square Object.

Now we learned how type conversion removes most of the code in our action classes and in our views, and makes the codes easier to develop and understand. We also learned that Struts 2 allows us to simplify our actions and views by removing all the codes we might normally put in our actions and views, for preparing the data and allowing type conversion to do that work for us. Finally, we learned about the built-in type conversion and what they do for us.



F

Migrating Struts 1 to Struts 2

Struts 1 Framework is widely used in the industry to develop Java Web applications and most of the programmers are comfortable with Struts 1 Framework, its architecture, and the way components are created, configured and used in it. The earlier versions, which we are collectively referred as Struts 1, have already proved their usability as a framework in designing, developing and maintaining Java Web applications. Struts 1 is recognized as one of the most popular application framework for Java Web applications, which implements the Model-View-Controller (MVC) design pattern in the development of Web applications.

The new release of Apache Struts Framework is Struts 2.0.6, also referred as Struts 2. The Struts 2 is a result of a merger of Struts Framework with WebWork2 Framework. Struts 2 is designed to be simpler, smarter, and easier. This is done by keeping it closer to what Struts was always meant to be in terms of Model 2 MVC pattern implementation, easier component creation and configuration, internationalization support, and rich Tag Libraries. To understand Struts 2 Framework in a better way and to discuss the key changes in the framework, a simple Web application developed in Struts 1 can be migrated to Struts 2 application. This migration helps in explaining how things are put together to function in Struts 2, and how the development with Struts 2 has become easier as compared to Struts 1.

Before we start migrating a Struts 1 application to Struts 2, let's have a look at the new architecture and the request life-cycle in Struts 2.

Struts 2 is a Front Controller Framework, similar to its earlier counterpart and the number of concepts which were supported in Struts 1 is same as in Struts 2.

The request life-cycle in the application, developed by using Struts 2 Framework, is quite different from Struts 1 Framework (see the different type of components being used like Interceptors, actions, and results). For processing a request, a new request is matched by framework with a configuration to know the Interceptors, action and results to be used for the request. The request passes through the stack of Interceptors, which are configured for this particular action match. The Interceptors work as pre-processors for the request and are analogous to RequestProcessor class of Struts 1. The new instance of the Action class is created and the method of the Action class providing logic for the request is executed. The result (usually a JSP page) corresponding to the result code returned from the method executed is obtained and used to display some output to the client. The request once again passes

through the Interceptors for some additional processing, if any. Finally, the response is returned to the user, which is normally in some HTML content.

One of the most significant architecture differences between Struts 1 and Struts 2 is the use of ActionForm. In Struts 2, the data can be handled by the Action itself. To display the required information the data can be fetched from Action. In other words, ActionForm are now obsolete in Struts 2. In addition, the POJO (Plain Old Java Object) classes can be used as Action and ActionForm. The directory structure and the configuration style is changed which are adopted from WebWork2.

For the people working with Struts 1, the concept of Interceptors and result is new. The migration of a simple Struts 1 application to Struts 2 will be needful for the readers to know where these new concepts fit in the framework and how they make the development of Web application flexible and easy.

Migration Strategy

There are many developers who are using Struts 1 in the production and will continue to do so in the coming years to support the maintenance of the existing Struts 1 projects. So, the question of choosing Struts 1 or Struts 2 as a framework comes only when we are planning a new project. The migration of a Struts 1 application to Struts 2 application is not difficult, but requires a good amount of effort. The migration of an application requires the changing of Actions, JSP pages, and configuration in order to support the new framework, which is time-consuming. But for some Struts 1 applications which will continue to grow and change in the future, the time spent on its migration to a Struts 2 application is of worth.

The migration strategy used in this chapter for migrating Struts 1 to Struts 2 helps in getting a clear comparison between the two frameworks. Only those components need to be modified or rewritten where these frameworks differ. An existing Struts 1 application can be migrated to Struts 2 by just adding Struts 2 JARs and changing other components, like Action classes, JSP pages, and configuration files one by one. Both these versions can be used in the same Web application.

Required Changes

For changing an existing Struts 1 application into Strut 2, you are required is to know the basic changes in the framework. The new configuration style is to be followed and the obsolete things have to be removed. The basic application architecture will remain unchanged. The real change comes in the design of various components with the new architectural support provided by the framework. For example, the different Struts Tag Libraries provided with Struts 2 are different from what is provided with Struts 1. This fact forces updating of all JSP files, so that they can support Struts 2 tags. Let's understand the basic changes in framework, which a reader must know to migrate a simple Struts 1 application to Struts 2 application.

Changes in Front Controller

Struts has always been a Front Controller Framework. Being Front Controller means that all the requests to a Web application are handled and processed by a single and common Controller component. In Struts 1, the Front Controller was `org.apache.struts.action.ActionServlet` class. This Controller Servlet managed all requests and searched for matching action mapping in `struts-config.xml` to find the Action class to be executed. Struts 2 is also not an exception and have Front Controller component, like Struts 1.

But the Front Controller has changed from Servlet `org.apache.struts.action.ActionServlet` to a filter `org.apache.struts.action2.dispatcher.FilterDispatcher`. In a Struts 2 application, the `FilterDispatcher` looks at the request and determines the proper Action in the mappings provided in `struts.xml`.

Configuring Framework

With the change in the Front Controller component, the way a Struts Web application framework is enabled within the Servlet container is changed. In Struts 1, the Controller Servlet mapped with its Servlet class and url-pattern. Similarly, to make `web.xml` Struts 2 enabled, the Servlet filter is mapped with the filter class `FilterDispatcher` and url-pattern.

Here's the code, given in Listing F.1, that shows how `web.xml` was made Struts 1 enabled:

Listing F.1: Struts 1 enabled `web.xml`

```
<web-app>

    <servlet>
        <servlet-name>struts1</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>

    <servlet-mapping>
        <servlet-name>struts1</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

</web-app>
```

Similarly, in Struts 2, the clear mapping of its Front Controller component is provided which happens to be `FilterDispatcher` class. The Servlet configuration of Struts 1 is replaced with a filter configuration in Struts 2. The package structure has also been changed in Struts 2 with the addition of some new packages from WebWork2.

Here's the code, given in Listing F.2, for `<filter>` and `<filter-mapping>` elements:

Listing F.2: Struts 2 enabled `web.xml`

```
<web-app>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts.action2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Now we have two different Controller components in Struts 1 and Struts 2. In Struts 1, `org.apache.struts.action.ActionServlet` was configured as Controller, while in Struts 2, we need to configure `org.apache.struts.action2.dispatcher.FilterDispatcher` filter class as Controller. It is a better to use `/*` as the url-pattern in comparison to `/*.do`. The default extension for Struts 2 actions is `.action` and is defined in `default.properties` file, which reside in Struts 2 JAR file. This default extension can be set by overriding the value of `struts.action.extension` property in `struts.properties` file and adding it in WEB-INF/classes folder.

Structure of Actions

An Action class and its methods are basically designed for the logic implementation required for some request. An Action class is responsible for executing business logic. In Struts 1, all Action classes had to extend `org.apache.struts.action.Action` base class, while in Struts 2 we have different ways available to create our Action class. A Struts 2 Action class can be created by implementing `com.opensymphony.xwork2Action` interface with some other interfaces to enable various services. The other way of creating a Struts 2 Action class is to extend `com.opensymphony.xwork2.ActionSupport` class. In addition, any POJO object can be used as a Struts 2 action object.

Listing F.3 shows various examples of Action classes:

Listing F.3: Struts 1 and Struts 2 Action classes

```
//Struts 1 Action class

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class MyAction extends Action {
    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        return mapping.findForward("success");
    }
}

//Struts 2 action class

import com.opensymphony.xwork2.ActionSupport;

public class MyAction extends ActionSupport {
    public String execute() throws Exception {
        //some statements
        return "success";
    }
}

//POJO as Struts 2 Action class
```

```
public class MyAction{  
    public String execute() throws Exception{  
        //some statements  
        return "success"  
    }  
}
```

In Struts 1, the initial entry point for the framework into the Action class was its `execute()` method. Though some other methods of Action class can be invoked by `DispatchAction` class, the first method to be executed in a Struts 1 Action class was the `execute()` method. This is not the case in Struts 2. In Struts 2, any method of your Action class, having the signature `public String methodName()`, can be invoked through configuration.

The signature of `execute()` method has totally changed in Struts 2. The `execute()` method of Struts 1 Action class took four arguments and returned a reference of type `ActionForward`. On the other hand, the `execute()` method of Struts 2 Action class can be defined as the simplest one having no argument and returning a `String`. Observe the difference between the prototype of `execute()` methods of Struts 1 and Struts 2 in the following code:

```
//execute() method for struts 1 action  
  
public ActionForward execute(ActionMapping mapping,  
                           ActionForm form,  
                           HttpServletRequest request,  
                           HttpServletResponse response) throws Exception {  
    return mapping.findForward("success");  
}  
  
//execute() method for struts 2 action  
  
public String execute() throws Exception {  
    return "success";  
}
```

Struts 1 actions depended on Servlet API as reference of `HttpServletRequest` and `HttpServletResponse`, which were passed in the `execute()` method. This created a problem in the testing of Struts 1 actions. On the other hand, the Struts 2 actions are not coupled to a container and can still access the original request and response. This came into reality with the implementation of Dependency Injection and Inversion of Control pattern. The Dependency Injection support makes testing of Struts 2 action simpler.

In Struts 2, for every new request a new instance of the action is created and, hence, the Action classes need not to be thread safe. On the other hand, Struts 1 actions were singleton and had to be thread safe.

Combining Action and ActionForm

Struts 1 `ActionForm` objects were used to capture and hold input values. All `ActionForms` had to extend `org.apache.struts.action.ActionForm` base class. Now the `ActionForms` are obsolete in Struts 2. Struts 2 Action classes can work as `ActionForm` and we do not need to create an extra class here. Struts 2 Action class properties can be used as input properties. The Action class itself can define getter and setter methods for all input properties. The Action class properties can be accessed by using tablibs.

Appendix: F

The property fields of Action class automatically get populated similar to Struts 1 ActionForms. In addition to POJO Actions, Struts 2 supports POJO form objects also.

Here's the code, given in Listing F.4, which shows an example of Action class that can behave like Struts 1 ActionForm:

Listing F.4: Struts 2 Action class as ActionForm

```
import com.opensymphony.xwork2.ActionSupport;

public class MyAction extends ActionSupport {

    public String execute() throws Exception {
        setMessage("Welcome!");
        return "success";
    }

    //input property 'message'
    private String message;

    //getter and setter method for property 'message'

    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Changes in Action Mapping

All Struts 1 and Struts 2 actions must be configured before they are requested and executed. The action mapping is required by the Front Controller to match the request and find the Action class, which is to be executed for a given request. The first two differences in Struts 1 and Struts 2 action mapping are the name and location of configuration file. Struts 1 actions were configured in `struts-config.xml` file present in the `WEB-INF` directory of the application. Similarly, Struts 2 actions are configured in `struts.xml` file, which is located in the `WEB-INF/classes` folder of the application.

Here's the sample Struts 1 action mapping as shown in Listing F.5:

Listing F.5: Struts 1 action mapping in struts-config.xml

```
<struts-config>
    <form-beans>
        <form-bean name="login" type="pack.form.LoginForm" />
    </form-beans>

    <action-mappings>
        <action
            path="/login"
            type="pack.action.LoginAction"
            name="login"
            scope="request">
```

```
<forward name="success" path="/jsp/admin.jsp" />
<forward name="error" path="/jsp/errorpage.jsp" />
</action>
</action-mappings>
</struts-config>
```

The struts-config.xml file contains mapping for the actions and form-beans. One `<action>` element maps a single request path (/login) to a single Action class (LoginAction), an action form name (login) and a set of forward elements providing url to forward to, in case of success or error returned from the `execute()` method of Action class.

In a similar way, Struts 2 actions are mapped in struts.xml file. Here's a sample action mapping of Struts 2 action as shown in Listing F.6:

Listing F.6: Struts 2 action mapping in struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="hello-default" extends="struts-default">
        <action name="Hello" class="Hello">
            <result>/Hello.jsp</result>
        </action>
    </package>
</struts>
```

The first thing, which can be noticed from the comparison of Listing F.5 and Listing F.6, is that there is no `<action-mapping>` node in struts.xml. We have include and package nodes which are new for Struts programmers. Struts 2 allows the modularization of configuration in a number of files. The include node inserts the content of a file into the current file. The actions can be grouped together in a package node with a unique value for name attribute. The package node provides namespace separation and structure. We can extend a package to gain access to its configuration of actions, results, Interceptors and exception. In Listing F.6, we have extended a package named `struts-default` which is defined in the included file, `struts-default.xml`.

Using Struts 2 Tag Library

Struts 1 came with a set of Tag Libraries used to create JSP pages. Though the use of these custom Tag Libraries was not required to use the framework itself, it proved useful in the creation of JSP pages for the application. This helped in writing more precise and readable code for the view component in an easy way. These custom Tag Libraries included various tags for creating dynamic HTML forms, tags for accessing data from the FormBean, and some tags for conditional generation of output. The set of Struts 1 Tag Libraries included `struts-html.tld`, `struts-bean.tld`, `struts-logic.tld` and `struts-template.tld`. Because the library is already written, all we have to do was to tell the Servlet engine about it. In Tomcat, you use the `<taglib>` tag in the `web.xml` file to specify the URI of the tag library, and the location of the tag library descriptor file on the Web server system.

Here's the code, given in Listing F.7, for a JSP page using Struts 1 tags:

Appendix: F

Listing F.7: A sample JSP page using Struts 1 tags

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><bean:message key="app.title"/></title>
</head>
<body>
<div align=center>
    <h2><bean:message key="app.welcome"/></h2><br><br>
</div>
</body>
</html:html>
```

Struts 2 provides a different set of tags to be used in JSP pages. These set of Struts 2 tags are not only equivalents of Struts 1 tags (shown in Listing F.8), but also provide significant improvement over what Struts 1 tags provide.

Listing F.8: A sample JSP page using Struts 2 tags

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><s:text name="app.title"/></title>
</head>
<body>
<div align="center">
    <h2><s:text name="app.welcome"/></h2>
</div>
</body>
</html:html>
```

Support for Internationalization

Both Struts 1 and Struts 2 Framework support internationalization, which is about customizing user interfaces to support various formats and messages for different locales and regions. In Struts 1 application, the `<message-resources>` element was configured in `struts-config.xml` as follows:

```
<message-resources parameter="com.kogent.ApplicationResources" />
```

This defined the base name of resource bundles (.properties files) used to find localized messages. The default resource bundle, as configured here was `ApplicationResources.properties` file and was found in the `WEB-INF\classes\com\kogent` folder of the application.

In Struts 2, the resource bundle can be registered in `struts.properties`. We can set the value for `struts.custom.i18n.resources=ApplicationResources` in the `struts.properties` file to declare that the base name of resource bundles to be used will be `ApplicationResoucces`, i.e.

ApplicationResources_xx.properties files. The xx stands for the standard locale code, like 'en' for English and 'fr' for French, etc.

Both Struts 1 and Struts 2 provide some tags to access the localized text messages. In Struts 1, we have `<bean:message key="somekey" />` tag and Struts 2 we have `<s:text name="somekey" />` tag.

We are going to change a Struts 1 sample login application into Struts 2 application. This migration will go step by step, changing components one by one, with due consideration to the basic framework changes from Struts 1 to Struts 2 that are discussed by now this chapter.

Migrating a Struts 1 Application to Struts 2

We need a Struts 1 application first which will be migrated to Struts 2 further. So the first application being developed in this section is based on Struts 1 (version 1.2) Framework and the same application will be used for migrating to Struts 2 application, implementing all changes required according to the changes in the construction of Action class, application configuration, action mapping and tag library support.

Struts 1—Login Application

The application which is being discussed in this section is a simple application built using Struts 1 Framework. The application consists of three JSP pages, an Action class, a FormBean class, few configuration files and Tag Libraries. Three basic view components designed are `index.jsp`, `login.jsp` and `login_success.jsp`. The basic logic implemented in the application is to provide a login user interface (`login.jsp`) to the user where the username and password is entered. The form submission invokes the Controller which searches the configuration file for a match to know the FormBean (`LoginForm`) and populates it with the entered data before invoking the Action class (`LoginAction`). The user is forwarded to the next JSP page according the `ActionForward` object returned from the `execute()` method of Action class.

But to make all these work together in the framework, all components are developed and configured according to what framework supports. We'll go through the code files for all these components before getting a look at how these are configured in `web.xml` and `struts-config.xml`.

Here we'll discuss only the basic structure of components designed for this Struts 1 application. This component structure and syntax will be compared with what is provided in Struts 2 Framework. This comparison will help in the migration process of Struts 1 application to a Struts 2 application.

Directory Structure

The directory structure of the application is shown in Figure F.1.

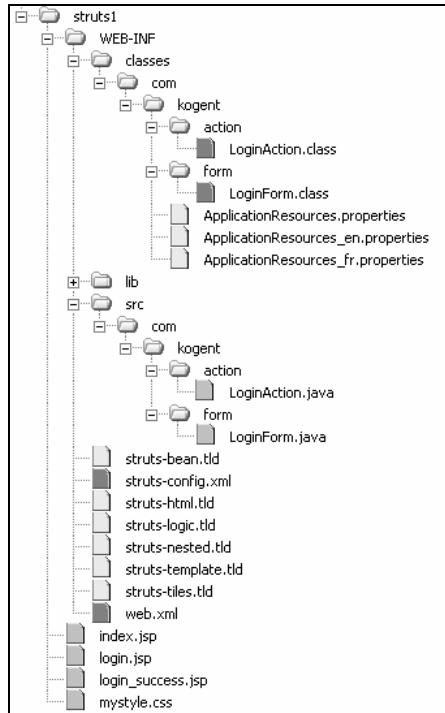


Figure F.1: Directory structure of a Struts 1 application.

Create a folder for your application, say **struts1**, in **D:\Code\Appendix F** folder. Create some more folders, as shown in Figure F.1, like **WEB-INF**, **WEB-INF/classes**, and **WEB-INF/lib**. Place these different types of files at the proper location in the directory structure as described in the following statements:

- All packages containing class files to be placed in **WEB-INF/classes** folder.
- The **WEB-INF/lib** folder contains all Struts 1 JAR files.
- All Struts Tag Libraries to be placed in **WEB-INF** folder.
- All configuration files, like **web.xml** and **struts-config.xml**, to be placed in **WEB-INF** folder.
- All JSP files are saved inside project folder (**struts1**) directly.
- All message-resource property files are searched file in the **WEB-INF/classes** folder within the appropriate package.

You can put a **WEB-INF/src** folder containing source files (.java files) for all class files created. This folder is optional in your application and you can put your sources files at any other location.

Creating **LoginForm** Class

This is a FormBean class which is extended from **org.apache.struts.action.ActionForm** class. This FormBean class declares input properties named **username** and **password**. For each property, there is a pair of getter-setter methods. In addition to these methods, a **LoginForm** class override methods like **validate()** method. This FormBean is automatically populated with the data entered in the input fields provided in a JSP page, which happens to be **login.jsp** in our case. We can put our

logic to check the validity of this data in validate() method. Our validate() method, here, checks only for the blank fields.

To see the connection between LoginForm FormBean and login.jsp designed to populate its input properties, observe the form-bean and action mapping provided in Listing F.15.

Compile the code, given in Listing F.9, for LoginForm.java and put the LoginForm.class file in WEB-INF/classes/com/kogent/form folder (you can find LoginForm.java file in Code\Appendix F\struts1\WEB-INF\src\com\kogent\form folder in CD):

Listing F.9: LoginForm.java

```
package com.kogent.form;
import org.apache.struts.action.*;
import javax.servlet.http.*;
public class LoginForm extends ActionForm {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    public ActionErrors validate(ActionMapping mapping,
        HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if ( (username == null) || (username.length() == 0) ) {
            errors.add("username", new ActionError("app.username.blank"));
        }
        if ( (password == null) || (password.length() == 0) ) {
            errors.add("password", new ActionError("app.password.blank"));
        }
        return errors;
    }
}
```

Creating LoginAction Class

LoginAction is an Action class and extends the org.apache.struts.action.Action class. One main method, defined in this Action class, is execute(). This class provides the business logic which is required for a particular request. The execute() method can get the reference of LoginForm FormBean class by typecasting ActionForm object passed to it as one of the four arguments. The execute() method gets an object of ActionForward class by using the findForward() method and

returns it. The next view or action is decided according to this returned object, which is mapped by using <forward> element of an action mapping provided in struts-config.xml. An object of ActionMessages can be set with request using saveErrors() method. This ActionMessages object contains some error messages which can be added by using add(String, ActionMessage) method.

Compile the code, given in Listing F.10, for LoginAction.java and keep the generated LoginAction.class in WEB-INF/classes/com/kogent/action folder (Figure F.1) (you can find LoginAction.java file in Code\Appendix F\struts1\WEB-INF\src\com\kogent\action folder in CD):

Listing F.10: LoginAction.java

```
package com.kogent.action;

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;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

import com.kogent.form.LoginForm;

public class LoginAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        ActionMessages messages=new ActionMessages();
        LoginForm loginForm = (LoginForm) form;
        if(loginForm.getUsername().equals(loginForm.getPassword()))
            return mapping.findForward("success");
        else{
            messages.add("invalid", new ActionMessage("app.invalid"));
            saveErrors(request, messages);
            return mapping.findForward("failure");
        }
    }
}
```

Creating JSP Views

In the MVC architecture implementation of Struts Framework, the view components are basically designed by using JSP pages. In this application, we'll design three JSP pages:

- index.jsp** – The first page for the application
- login.jsp** – The login user interface with fields for username and password
- login_success.jsp** – The page intimating user for successful login

Struts 1 Framework provides a set of Tag Libraries which help in designing JSP pages. In addition to helping in creating more interactive form-based applications, Struts tags make JSP pages more readable and manageable.

The **index.jsp** Page

This is the first page which appears when the application is accessed. It displays a message from the resource bundle, according to the current locale (English, French) using `<bean:message>` tag. This page provides a link to `login.jsp` page. The two Tag Libraries added to this JSP page are `struts-html.tld` and `struts-bean.tld`.

Here's the simple lines of code, given in Listing F.11, for `index.jsp` page (you can find `index.jsp` file in `Code\Appendix F\struts1\` folder in CD):

Listing F.11: `index.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><bean:message key="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<div align=center>
    <h2><bean:message key="app.welcome"/></h2><br><br>
    <a href="login.jsp">L o g i n</a>
</div>
</body>
</html:html>
```

All JSP pages use a style sheet `mystyle.css`. You can copy it from the CD provided with this book.

The **login.jsp** Page

This JSP page is designed to provide a login form with two fields to enter username and password. The form and the fields are designed by using `<html:form>`, `<html:text>` and `<html:errors>` tags defined in the `struts-html.tld` file, which is added to this JSP page using the taglib directive. The Struts Tag Libraries used here are `struts-html.tld` and `struts-bean.tld`. The action attribute sets with the `<html:form>` tag is used to match an action mapping in `struts-config.xml` to know which FormBean class and Action class are to be used. In addition to the input fields, this page also displays some error messages, which may be set by the Action class or the `validate()` method of FormBean class.

Here's the code, given in Listing F.12, for `login.jsp` (you can find `login.jsp` file in `Code\Appendix F\struts1\` folder in CD):

Listing F.12: `login.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>
<html>
    <head>
```

```
<title><bean:message key="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<html:form action="/login">
<table align=center>
<tr>
    <td colspan=2 align=center class=boldred>
        <html:errors property="invalid"/></td>
    </tr>
    <tr>
        <td colspan=2 align=center>Login!</td>
    </tr>
    <tr><td class="boldred" colspan=2 align=center>
        <html:errors property="username"/></td></tr>
    <tr>
        <td ><bean:message key="app.username"/></td>
        <td ><html:text property="username"/></td>
    </tr>
    <tr><td class="boldred" colspan=2 align=center>
        <html:errors property="password"/></td></tr>
    <tr>
        <td><bean:message key="app.password"/></td>
        <td><html:password property="password"/></td>
    </tr>
    <tr>
        <td colspan=2 align=right><html:submit value="Login"/></td>
    </tr>
</table>
</html:form>
<div align=center><a href="index.jsp">B a c k</a></div>
</body>
</html>
```

To see how the data entered from this JSP page automatically populates our `LoginForm` and how the `LoginAction` Action class is executed, we can see the relationship among these files in the action mapping provided in `struts-config.jsp`.

The `login_success.jsp` Page

This is again a simple page created to display a message intimating user for successful login. The Controller selects this JSP page to show it only when the username and password matches and `execute()` method of our Action class returns `mapping.forward("success")`.

Here's the code, given in Listing F.13, for `login_success.jsp` (you can find `login_success.jsp` file in `Code\Appendix F\struts1\` folder in CD):

Listing F.13: `login_success.jsp`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><bean:message key="app.title"/></title>
```

```

<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<div align=center>
<h2><bean:message key="app.success_login"/></h2><br><br>
<a href="index.jsp">L o g O u t</a>
</div>
</body>
</html>

```

After creating `LoginForm` class, `LoginAction` class and three JSP pages discussed earlier, let's configure these components in the Deployment Descriptors to relate them to work together.

Configuring Struts 1 – Login Application

To configure this application, the two configuration files used are `web.xml` and `struts-config.xml`. Read on to discuss them in detail.

The `web.xml` File

The Controller Servlet is declared in this file with all the parameters to initialize it. We need to provide a Servlet mapping for this Controller Servlet in `web.xml` file. This mapping is provided using `<servlet-name>`, `<servlet-class>` and `<url-pattern>` elements. The Controller Servlet class given for the Struts 1 application is `org.apache.struts.action.ActionServlet`. The url pattern for Struts 1 application is `*.do`. In addition, the `index.jsp` page is declared in the `<welcome-file-list>` making it the first page to appear.

Here's the code, given in Listing F.14, for `web.xml` file (you can find `web.xml` file in `Code\Appendix F\struts1\WEB-INF` folder in CD):

Listing F.14: `web.xml`

```

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="2.4" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Login App - Struts 1</display-name>
  <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>debug</param-name>
      <param-value>3</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>3</param-value>
    </init-param>
  </servlet>

```

```
<load-on-startup>0</load-on-startup>
</servlet>

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

<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

The **struts-config.xml** File

- ❑ This is the main configuration file in any Struts 1 application. This file contains `<form-bean>` and `<action-mapping>`. The application is Struts 1.2-based and the version is declared in the DTD of this configuration file shown in Listing F.15 (you can find `struts-config.xml` file in `Code\Appendix F\struts1\WEB-INF` folder in CD):

Listing F.15: struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<struts-config>

<form-beans >
    <form-bean name="loginForm" type="com.kogent.form.LoginForm" />
</form-beans>

<action-mappings >
    <action
        attribute="loginForm"
        input="/login.jsp"
        name="loginForm"
        path="/login"
        scope="request"
        type="com.kogent.action.LoginAction">
        <forward name="success" path="/login_success.jsp" />
        <forward name="failure" path="/login.jsp" />
    </action>
</action-mappings>
<message-resources parameter="com.kogent.ApplicationResources" />
</struts-config>
```

The following three important components are mapped in this file:

- ❑ `<form-bean>`—This element is used to declare a FormBean class and associate the FormBean class name with a name attribute. The FormBean class used here is `com.kogent.form.LoginForm` with name “`loginForm`”.
- ❑ `<action-mapping>`—This element includes action mappings. The mapping for only one action is given for this application and shown in Listing F.15. The attributes for this element are attribute,

input, name, path, scope and type. The FormBean to be populated with the data provided in the request is `loginForm`, which further maps to our `LoginForm` FormBean. The type attribute gives the name of Action class to be executed, which is `LoginAction` class. The two `<forward>` elements refer two JSP pages which are to be used in case of success and failure returned by `execute()` method of `LoginAction` class.

- `<message-resources>` – The `<message-resources>` elements define the base name to be used to find the locale specific resource bundle, i.e. properties file. These properties files or resource bundle contains key/message supporting different locales.

Properties Files

`PropertyResourceBundle` class is the Struts implementation of `java.util.ResourceBundle` class and this implementation allows Struts-based applications to use these resources which stores key/message pairs. In Struts 1 application, the `ApplicationResource.properties` file is added to the project, which prepares the resource bundle with all String, messages, and contains the messages in default language.

Two other files used in this application are `ApplicationResource_en.properties` (for English) and `ApplicationResource_fr.properties` (for French). The .properties file is text-based files and best for implementing internationalization.

Here's the code, given in Listing F.16, that gives all the key-message pairs in `ApplicationResource.properties` file (you can find all properties files in `Code\Appendix F\struts1\WEB-INF\classes\com\kogent` folder in CD):

Listing F.16: `ApplicationResource.properties`

```
# Resources for parameter 'com.kogent.ApplicationResources'
# Project struts1

app.title=Struts 1 Application
app.welcome=Welcome to Struts 1 Application
app.username=User Name
app.password=Password
app.success_login=You have successfully Logged In.

app.username.blank=User Name is Required
app.password.blank=Password is Required

app.invalid=Invalid User Name or Password.
```

The next file `ApplicationResource_en.properties` contains the same text as in `ApplicationResource.properties` file, as both support English language. The last property file `ApplicationResource_fr.properties` contains the same set of key-message pairs but the messages are translated in French language.

Here's the code, given in Listing F.17, for a taste of French:

Listing F.17: `ApplicationResource_fr.properties`

```
# Resources for parameter 'com.kogent.ApplicationResources'
# Project struts1

app.title=Application des Struts 1
```

Appendix: F

```
app.welcome=Faire bon accueil aux Struts 1 à l'application  
app.username=Nom d'utilisateur  
app.password=Mot de passe  
  
app.success_login=Vous avez avec succès entré  
app.username.blank=Le nom d'utilisateur est exigé  
app.password.blank=Le mot de passe est exigé  
app.invalid=Nom ou mot de passe inadmissible d'utilisateur
```

Running Struts 1—Login Application

After developing and configuring various components, we can now run our Struts 1 application to see the output of various JSP pages and other logic implementations. You need to create a WAR file, say struts1.war, and deploy it on your web server before running it as done with earlier applications in the book. Open your browser and access the application using url <http://localhost:8080/struts1/>, where 8080 is the port number of the Web server being used here. The first page which appears is the index.jsp and its output is shown in Figure F.2.



Figure F.2: The index.jsp page of Struts 1 application showing welcome message.

Click on the 'Login' hyperlink to display the login.jsp page, as shown in Figure F.3. This page gives two text input fields to enter 'User Name' and 'Password' along with a 'Login' button.

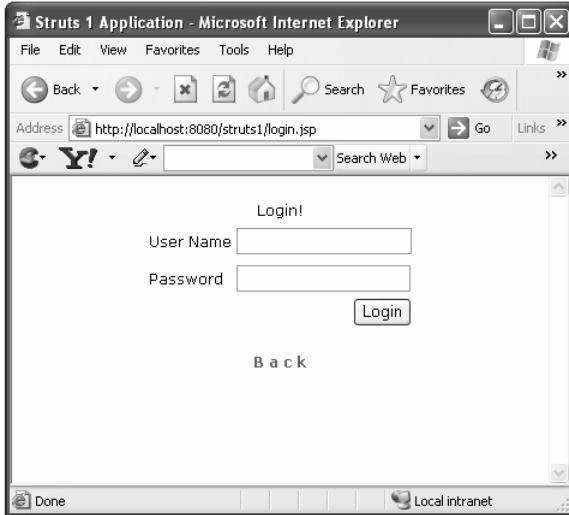


Figure F.3: The login.jsp of Struts 1 application showing login form.

If the form is submitted with any of the fields left blank, the error message, which is set by the validate() method, will be displayed using <html:text property=" " /> tag, as shown in Figure F.4.

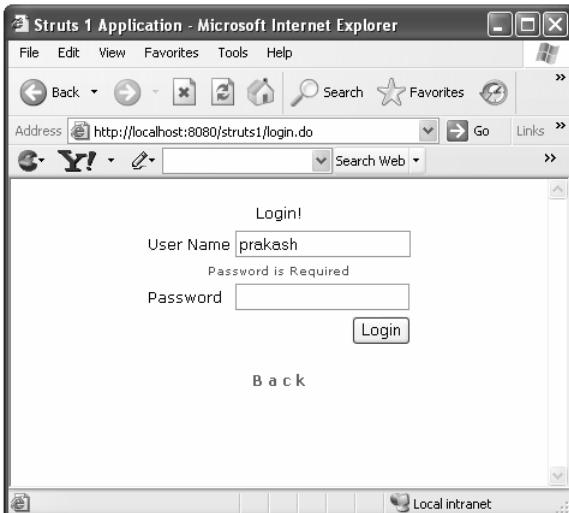


Figure F.4: The login.jsp of Struts 1 application showing error message.

This basic logic, which is implemented in execute () method of LoginAction class, approves a login successful only when the user name and password are same. So, enter a same string in 'User Name' and 'Password' field to see login_success . jsp (shown in Figure F.5) displaying a message.



Figure F.5: The login_success.jsp intimating user for successful login.

This application supports internationalization and we can run the application to see the text messages displayed in French language by changing the language preference of the browser to French. To set the language preference, select Tools→Internet Options→Languages and add language 'French (France)[fr]' to make it the first one in the list. Now, open a new browser and run your application. Your messages will be displayed in French language. The property file used then will be `ApplicationResource_fr.properties`.

Struts 2–Login Application

After creating an application in Struts 1 (1.2 specifically), let's migrate this application to a Struts 2 application. The best way to migrate a Struts 1 application to a Struts 2 application is to replace the Struts 2 JAR files in `WEB-INF/lib` folder of the application and start changing the different components step by step, like Action class, FormBean class, configuration files, and JSP pages to support Struts 2 Framework.

Having discussed the basic changes in the component designs for Struts 1 to Struts 2 earlier in this chapter, the components are changed accordingly—what new concepts and APIs are introduced in the new framework and where the Struts 1 code and structure need to be changed. So first of all, copy `struts1` application developed in the previous section as `struts2` at `D:\Code\Appendix F\` folder. The directory structure is changed to the one shown in Figure F.6.

The basic changes in the application, which can be observed by seeing Figure F.6, are as follows:

- The `LoginForm` is missing
- The `struts-config.xml` file is replaced with `struts.xml` and placed in `WEB-INF/classes` folder.
- A new property file `struts.properties` is introduced.
- Struts 1 JAR files are replaced to Struts 2 JARs.
- No more `.tld` files

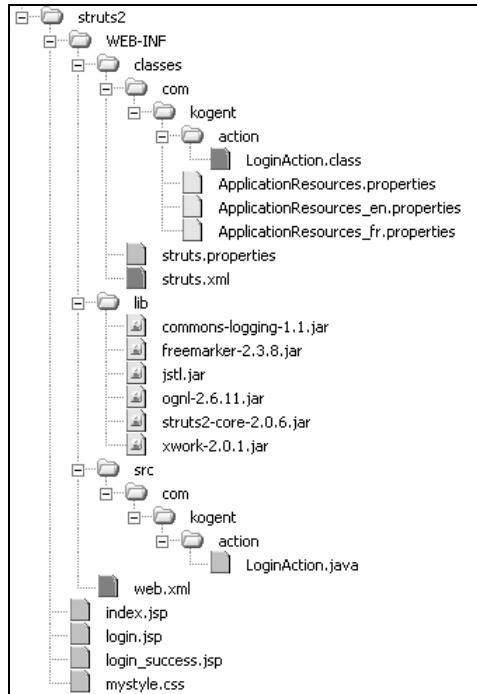


Figure F.6: Directory structure of a Struts 2 application.

Changes in **LoginAction** Class

In comparison to Struts 1, the Struts 2 Action classes are much simpler. We have already seen the structure of our `LoginAction` class in Listing F.10 which extended `org.apache.struts.action.Action` class and also seen the structure of its `execute()` method. The structure of an Action class can be as simple as shown in the following lines of code:

```
public class LoginAction{
    public String execute() throws Exception{
        //some logic
        return "success";
    }
}
```

The first thing which can be said here about Struts 2 Action class is that they can be POJO objects and doesn't require extension of a class or implementation of any interface. Further, the syntax of `execute()` method is the simplest one, i.e. `String execute (void)`. By convention, the method automatically invoked in the execution of an Action class is `execute()`. But, we can invoke any method of Action class having the signature `public String methodName()` by configuring it.

The `execute()` method of Struts 2 Actions does not take any arguments thus, making it free from Servlet APIs. But, sometimes, we need to access current requests `HttpServletRequest` object. In such a case, we can implement the `ServletRequestAware` interface to have `HttpServletRequest` object to use. This is a form of Dependency Injection.

Appendix: F

Our simple Action class after implementing the `HttpServletRequest` interface becomes like the one shown here:

```
public class LoginAction implements ServletRequestAware{
    private HttpServletRequest request;
    public void setServletRequest(HttpServletRequest request){
        this.request=request;
    }
    public String execute() throws Exception{
        //some logic
        return "success";
    }
}
```

Similarly, we have `com.opensymphony.xwork2.Action` interface that can be implemented to provide common results of ‘success’, ‘none’, ‘error’, ‘input’ and ‘login’ as constants. Some other interfaces are also available which can extend the functionality of your Action class.

NOTE

Struts 2 Actions has been discussed in Chapter 3.

The best way to create an action form is to extend `com.opensymphony.xwork2.ActionSupport` class, which implements some interfaces, like `Action`, `Validateable`, and `ValidationAware`. We know that the `action` attributes can be used as `input` properties. So our Action class can have a `validate()` method which contains the logic for validating values for the `input` properties. The `Validateable` interface references workflow Interceptor and its implementation assures that the `validate()` method of Action class is invoked before its `execute()` method. Keeping this logic, we have changed our `LoginAction` class, accordingly. It now extends `ActionSupport` class and defines two methods, i.e. `execute()` and `validate()` method.

Here's the code, given in Listing F.18 for `LoginAction` class for Struts 2 (you can find this `LoginAction.java` file in `Code\Appendix F\struts2\WEB-INF\src\com\kogent\action` folder in CD):

Listing F.18: `LoginAction.java` for Struts 2

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
}
```

```
public void setPassword(String password) {
    this.password = password;
}
public String execute() throws Exception {
    if(username.equals(password))
        return SUCCESS;
    else{
        this.addActionError(getText("app.invalid"));
        return ERROR;
    }
}
public void validate() {
    if ( (username == null) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
    }
    if ( (password == null) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
    }
}
}
```

Listing F.18 shows that the `LoginAction` class is also behaving like a `FormBean`. The input properties and public getter and setter methods for these properties are defined in this Action class. In addition, the Action class contains a `validate()` method also similar to `LoginForm` used in Struts 1 application. The new methods used are `addActionError()` and `addFieldError()`. These methods are used to set a message for some error in context which can be displayed on the JSP page. The `getText()` method obtains the message from resource bundle according to the Locale set. The `ActionSupport` class implements `Action` interface and, hence, the constants `SUCCESS`, `ERROR`, `INPUT`, and `NONE` can be used in place of string literals. The steps to change Struts 1 `LoginAction` to Struts 2 `LoginAction` class are as follows:

- Replace Struts 1 imports with Struts 2
- Move message property and public getter and setter methods from `LoginForm` to `LoginAction` class
- Remove `LoginForm` completely
- Change the `LoginAction` class to extend `ActionSupport` class
- Remove `ActionForm` typecasting and access properties directly

Changing `web.xml` File

The Front Controller is changed in Struts 2 from `org.apache.struts.action.ActionServlet` to `org.apache.struts2.dispatcher.FilterDispatcher`. So, the Servlet mapping given in `web.xml` is replaced with Filter mapping, as shown in Listing F.19 (you can find this `web.xml` file in `Code\Appendix F\struts2\WEB-INF` folder in CD):

Listing F.19: `web.xml` for Struts 2 application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" xmlns= "http://java.sun.com/xml/ns/j2ee
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
```

```
<display-name>Login App - Struts 2</display-name>

<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-
class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

With the Filters, it is better to use /* as url-pattern. The default extension for the Struts 2 Actions is .action, which can be set to something different in struts.properties available in Struts JARs.

Changing Struts Configuration File

The struts-config.xml file in Struts 1 application is replaced with the struts.xml, which provides action mappings. This configuration file, shown in Listing F.20, seems concise in comparison to struts-config.xml file shown in Listing F.15. For Struts 1 application, we map Action and ActionForm both, while in Struts 2 only Action classes are mapped. The single action mapping is provided with name="login" and type="com.kogent.action.LoginAction". Three results map JSP pages in case of three possible strings returned as result code from the execution of Action class. INPUT is returned implicitly if more data is required for the execution of execute() method. String constants—'success', 'error' and 'input'—can be returned explicitly also (see Listing F.18).

Here's the code, given in Listing F.20, for struts.xml (you can find struts.xml file in Code\Appendix F\struts2\WEB-INF\classes folder in CD):

Listing F.20: struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
    <include file="struts-default.xml"/>
    <package name="login-default" extends="struts-default">
        <action name="login" class="com.kogent.action.LoginAction">
            <result name="success">/login_success.jsp</result>
            <result name="error">/login.jsp</result>
            <result name="input">/login.jsp</result>
        </action>
    </package>
</struts>
```

Compare the Listing F.15 and Listing F.20 for the change in configuration file from Struts 1 to Struts 2. To change a Struts configuration file from Struts 1 to Struts 2, undertake the following steps:

- ❑ Rename the configuration file from `struts-config.xml` to `struts.xml` and put it in the WEB-INF/classes folder
- ❑ Change the DTD for Struts 2 (see Listing F.20)
- ❑ Replace `<struts-config>` with `<struts>`
- ❑ Insert `<include file="struts-default.xml" />`
- ❑ Remove `<form-bean>` elements as no form-bean mapping is required
- ❑ Replace `<action-mappings>` to `<package name="login-default" extends="struts-default">`
- ❑ In `<action>` element, replace the name of the attribute from path to name and from type to class
- ❑ Replace `<forward>` element with `<result>` element

The Struts 2 uses default values at a number of places. We can compare the following two lines, one from `struts-config.xml` and one from `struts.xml`:

```
<forward name="success" path="/login_success.jsp" />  
<result name="success">/login_success.jsp</result>
```

If we write `<result>` without the name attribute, it is taken as ‘success’ by default. The content of this element is taken as default to path. The default result type is `dispatch`. We can set the result type by setting `type` attribute with `<result>` element, as shown in the following line:

```
<result name="success" type="redirect">/login_success.jsp</result>
```

Now, we have the flexibility to modularize our configuration file and include different files in a single configuration file using `<include>` element. Further, we can create packages by using the `<package>` element, which provide namespace separation. A package can access actions, Interceptors, results and exceptions configured in other packages by extending them. The included file `struts-default.xml` resides in Struts 2 JAR, which contains the package `struts-default` configured, giving all access to default Interceptors and result types.

Configuring Resource Bundle

We are using `ApplicationResources.properties` as the resource bundle, which needs to be declared to the framework. To register the resource bundle in Struts 2 application, create a `struts.properties` file in WEB-INF/classes folder and set the value for a key `struts.custom.i18n.resources`, as shown here:

```
struts.custom.i18n.resources=com.kogent.ApplicationResources
```

This line in `struts.properties` overrides the value set for this key in `struts-default.xml` which is available in Struts 2 JAR and defines various properties.

Converting JSPs

We have used Struts 1 tags in the JSP pages designed for Struts 1 application designed earlier. Struts 2 Framework provides a different set of tags which can be used to create more flexible, elegant, and readable JSP pages. The basic requirement to make a JSP page Struts 2 tags enabled is to change the `uri` attribute in `taglib directive`. The following changes are required in the JSP pages designed in Struts 1 application:

- ❑ Change `<%@ taglib @%>` directive for the `uri` and `prefix` attribute. The value for `uri` attribute is `/struts-tags` and `prefix` is `s`.
- ❑ Replace `<bean:message key="app.title"/>` with `<s:text name="app.title"/>` to display localized message from properties file.
- ❑ The `<html:errors>` tags is replaced with `<s:actionerror />` which prints all action errors set in the context by the Action class using `addActionError()` method.
- ❑ Similarly, for all field errors set using `addFieldError()` method, we can use `<s:fielderrors/>`. Though, `<s:fielderrors/>` tag can be used to print field error messages at the desired location explicitly in the page, the field errors are displayed earlier the matching input fields implicitly.
- ❑ Replace `<html:form action="..." />` tag with `<s:form action="login" ... />`.
- ❑ For a single input text field, we can use single tag which displays its label according to the key provided:

```
<s:textfield name="username" key="app.username"/>
```

Here's the code, given in Listing F.21, for `index.jsp` page for Struts 2 application using Struts 2 tags (you can also find this `index.jsp` file in `Code\Appendix F\struts2\` folder in CD):

Listing F.21: `index.jsp` for Struts 2 application

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head><title><s:text name="app.title"/></title>
<link rel="stylesheet" href="mystyle.css" type="text/css" />
</head>
<body>
<div align="center">
    <h2><s:text name="app.welcome"/></h2>
    <a href="login.jsp">L o g i n</a>
</div>
</body>
</html:html>
```

In Listing F.21, a `<s:text ... />` tag is used to display a localized message from the respective resource bundle.

Here's the code, given in Listing F.22, for `login.jsp` page for our Struts 2 application which uses `<s:form>`, `<s:textfield>` and `<s:submit>` tag to build a form (you can find this `login.jsp` file in `Code\Appendix F\struts2\` folder in CD):

Listing F.22: login.jsp for Struts 2 application

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title><s:text name="app.title"/></title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <table align="center" width="300">
            <tr><td colspan="2"><div class="boldred"><s:actionerror/></div></td></tr>
            <tr><td align="center" colspan="2">Login!</td></tr>
            <tr><td align="center">
                <s:form action="login" method="post">
                    <s:textfield name="username" key="app.username"/>
                    <s:password name="password" key="app.password"/>
                    <s:submit value="Login"/>
                </s:form>
            </td>
            </tr>
            <tr><td align="center" colspan="2">
                <a href="index.jsp">B a c k</a>
            </td></tr>
        </table>
    </body>
</html>

```

The page displays all the action errors, if any, using `<s:actionerror/>` tag. All field errors are explicitly displayed earlier in the respective input field. We can also use `<s:fielderror/>` to print all field errors.

Similarly, update `login_success.jsp` for Struts 2 tags. Finally, update property files for all messages, which include Struts 1 and replace it with Struts 2.

Here's the code, given in Listing F.23, for `login_success.jsp` page for Struts 2 application using Struts 2 tags (you can find this `login_success.jsp` file in `Code\Appendix F\struts2\` folder in CD):

Listing F.23: login_success.jsp for Struts 2 application

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
    <head><title><s:text name="app.title"/></title>
    <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body>
        <div align="center">
            <h2><s:text name="app.success_login"/></h2>
            <a href="index.jsp">L o g O u t</a>
        </div>
    </body>
</html:html>

```

Running Struts 2—Login Application

Create a WAR file, say struts2.war, for the application and deploy it on your Web server. Now, let's run our Struts 2 application which has migrated from Struts 1 application. Running both these types of applications are same and we can access this Struts 2 application by giving `http://localhost:8080/struts2/` in the address bar of our browser. Figure F.7 shows the index.jsp page of Struts 2 application—Login application.



Figure F.7: The index.jsp page of Struts 2 application showing welcome message.

Now click on 'Login' hyperlink, which brings the login.jsp page, quite similar to login.jsp of Struts 1 application. The field errors are explicitly displayed over the input fields. This is shown in Figure F.8 where a message is displayed intimating text for the blank 'Password' field.

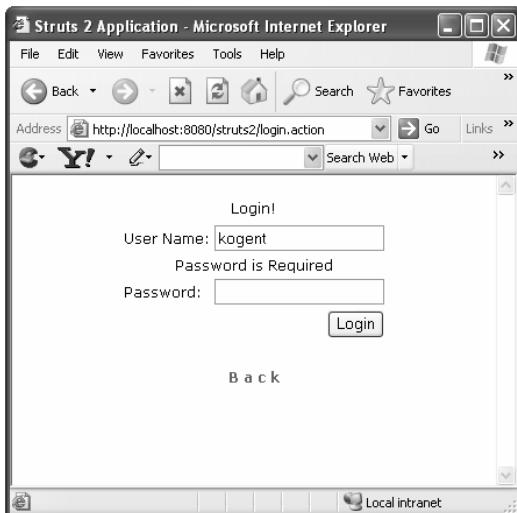


Figure F.8: The login.jsp of Struts 2 application showing error message.

The appearance of JSP pages here are similar in appearance to the JSPs designed for Struts 1 application, earlier in this chapter, with the only difference that Struts 2 is being displayed in place of Struts 1 (see title bars of various windows shown in figures for Struts 1 and Struts 2 applications). This is because of the updating of messages in resource bundles.

Similarly, you can run Struts 2 application for different locales by setting the language preference to other languages, say ‘French’. After setting ‘French’ as the first language in the list of preferred languages, open a new browser and run your application. Now your messages will be displayed in French language as set in `ApplicationResources_fr.properties`. Look at Figures F.9 and F.10 for pages localized for French language.

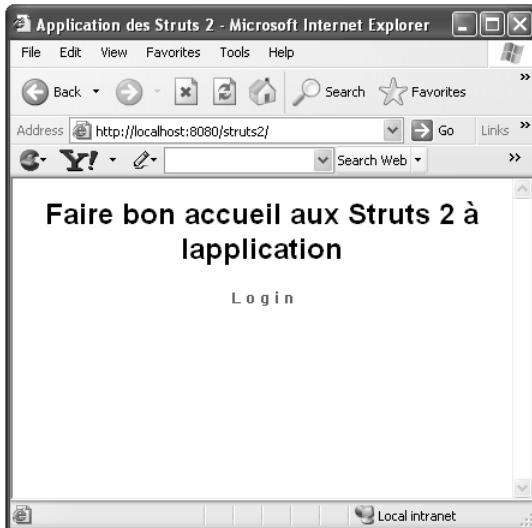


Figure F.9: The index.jsp page of Struts 2 application showing welcome message in French.

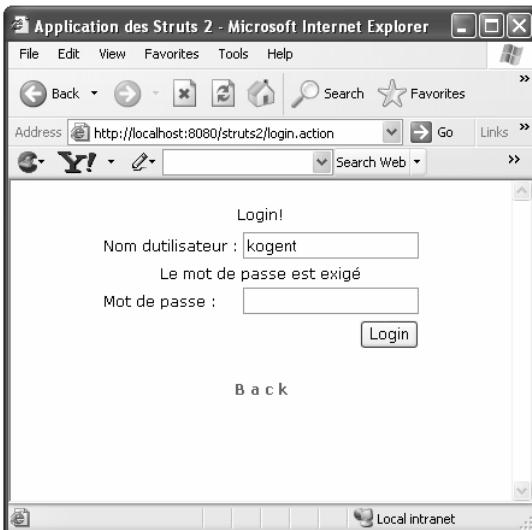


Figure F.10: The login.jsp of Struts 2 application showing error messages in French.

Appendix: F

In this way, we can migrate a simple Struts 1 application to Struts 2 application by inserting Struts 2 Framework APIs and updating all components to support the new framework and all the new concepts introduced. Most of the architecture implemented is same in both Struts 1 and Struts 2 Framework. All the new concepts and their implementations are added from WebWork2 framework. The introduction of Interceptors, Results, and POJO Action classes along with the new framework packages changed the way of designing, configuring and integrating various components of a Struts based Web application. This makes the development of Struts 2 project more flexible, elegant, precise and simple. The implementation of different forms of Dependency Injection further reduced the dependency of framework on other APIs, like Servlet API.

The main highlight of this chapter is the development of a Struts 1 application and its step by step migration towards a Struts 2 application. The discussion in this chapter describes various changes at the basic level of component designing for the application based on Struts Framework, and details how the support for POJO objects as Action class and simplification of execute() method signature are totally different in Struts 2 Framework. In addition, the chapter details the implementation of FormBean functionality in Action class, the configuration style, and introduces <include>, <package> and <result> elements and Struts 2 tags.



Index of Terms

A

AbstractUITag class, 234, 237, 257, 258, 261
Action annotation, 591
Action Chaining, 286
Action class, 12, 14, 16, 20, 22, 43, 44, 93, 189, 279, 334, 501, 583, 586, 588, 590, 591, 592, 593, 597, 598, 600, 604, 605, 608, 609, 610, 611, 612, 614, 620, 624, 625, 626, 627, 628, 629, 631, 632, 634, 636, 637, 639, 643, 644, 645, 646, 647, 649, 653
Action configuration, 583, 603
Action interface, 22, 44, 45, 46, 58, 68, 89, 280, 282, 610, 644, 646
ActionCleanupFilter class, 576
ActionContext class, 53, 54, 55
ActionContextCleanUp filter, 20, 424, 425
ActionForm class, 23, 72, 93, 430, 445
ActionFormResetInterceptor, 430, 431
ActionFormValidatorInterceptor, 430
ActionForward class, 634
ActionInvocation interface, 129, 393
ActionMapper interface, 50
ActionMapping class, 50, 70
Actions, 16, 22, 23, 32, 41, 43, 44, 59, 60, 68, 70, 151, 153, 288, 373, 397, 399, 409, 414,

416, 427, 428, 430, 434, 443, 576, 605, 624, 626, 644, 645, 647

ActionServlet class, 625
ActionSupport class, 22, 31, 44, 46, 48, 58, 74, 89, 104, 185, 229, 350, 355, 410, 531, 626, 645, 646
Alias Interceptor, 101
AnnotationValidationInterceptor class, 126
ApplicationAware interface, 63, 74, 79
applicationContext.xml, 428, 429
application-managed security, 452, 454, 464, 471, 479, 482, 484
Assert class, 488
Authentication, 453, 454, 458, 459, 460, 461, 462, 463, 464, 465, 467, 470
Authorization, 453, 459, 460, 461, 462, 463
AuthorizationFilter class, 483
Autowiring, 427

B

BasicTilesContainer, 549, 550
Bean Configuration, 583

C

Cactus, 488, 494, 495, 496, 498, 499, 504
Cactus task, 498

Cactus task definitions, 498
Cactus Testing Framework, 494
Cactus tests, 498, 499
CactusTestCase, 495
Chain Result, 286, 287, 288, 289, 308
Chaining Interceptor, 102
Checkbox Interceptor, 104
Config Browser Plugin, 411, 438, 448, 449
Configuration files, 9
ConfigurationManager, 17, 19
Constant Configuration, 579, 583
Container-managed security, 453
container-managed security, 465
Control tags, 195, 196, 214, 227, 230
Controller Servlet, 15, 625, 638

D

Data tags, 195, 202, 214, 227, 230
Debugging Interceptor, 105, 106
Dependency Injection and Inversion of Control, 62, 63, 628
DispatchAction class, 627
Dispatcher Result, 284, 286, 288, 289, 290, 291, 292, 293, 296
DispatcherListener, 17
DynaBeans, 23

E

Exception configuration, 583, 590
exception handlers, 33, 51, 587, 605
Exception Interceptor, 107, 108, 147
Execute and Wait Interceptor, 106
execute() method, 22, 33, 38, 43, 44, 45, 49, 52, 54, 58, 60, 68, 70, 75, 76, 79, 82, 85, 90, 102, 110, 136, 137, 145, 146, 157, 185, 229, 280, 282, 303, 307, 308, 309, 310, 311, 349, 355, 399, 500, 501, 532, 588, 593, 600, 608,

627, 628, 629, 632, 634, 637, 639, 644, 645, 647, 653
expressions, 106, 115, 166, 168, 169, 170, 171, 172, 173, 175, 177, 179, 180, 181, 182, 183, 186, 185, 192, 197, 211, 289, 294, 295, 298, 300, 343, 414, 415, 525, 526

F

Field Validator, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 335, 346, 353, 354, 359
file tag, 237
File Upload Interceptor, 109
FilterDispatcher, 14, 15, 19, 20, 34, 37, 189, 190, 346, 402, 419, 424, 425, 426, 478, 501, 533, 561, 576, 577, 584, 592, 601, 610, 625, 626, 646
form tags, 16, 233, 256, 257
forward element, 629
FreeMarker, 11, 13, 16, 20, 32, 107, 287, 293, 294, 421, 422, 425, 432, 521, 522, 523, 525, 527, 528, 536, 542, 543
FreeMarker Result, 287, 293, 294
FreeMarkerPageFilter, 425, 426

G

General attribute, 235, 261
Generic Tags, 195, 217
getAction(), 129, 280, 313
getInvocationContext(), 129, 130
getProxy(), 129
getResult(), 50, 130
getStack(), 129
getText(), 74, 387, 389, 390, 397, 398, 646
Groovy, 168, 255, 272, 407, 413, 414, 433, 434, 435

H

hashCode(), 169
HttpHeader Result, 287, 294

I

I18N, 111, 392
I18nInterceptor class, 393
Include configuration, 583, 586, 604
intercept() method, 105, 108, 117, 129, 130, 393, 394
Interceptor configuration, 50, 94, 145, 583, 603, 605
Interceptor Stacks, 23, 95, 98
Interceptors, 3, 10, 13, 14, 15, 17, 19, 20, 21, 24, 26, 32, 33, 38, 39, 43, 44, 60, 62, 70, 72, 89, 90, 92, 93, 94, 95, 96, 97, 98, 100, 101, 108, 109, 111, 122, 123, 126, 129, 131, 132, 133, 134, 137, 138, 141, 143, 144, 145, 147, 152, 153, 157, 162, 164, 279, 283, 391, 393, 409, 417, 434, 575, 577, 578, 581, 583, 586, 587, 603, 604, 606, 614, 623, 624, 630, 648, 653
Internationalization, 379, 380, 382, 380, 381, 386, 387, 390, 391, 394, 395, 397, 399, 401, 405, 414, 416, 542, 631

J

JasperManager, 413
JasperPrint, 413
JasperReports Plugin, 413
JFreeChart Plugin, 415, 438, 446, 448
JSF Plugin, 416, 417
JSP Standard Tag Library, 168
JspTestCase, 495
JSTL, 13, 23, 168
JSTL EL, 23

JUnit, 484, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 498, 499, 500, 504

L

Lambda Expressions, 179
Locale, 46, 55, 56, 93, 148, 380, 381, 383, 385, 393, 646
LocaleProvider interface, 46
localization, 317, 379, 414
Logger Interceptor, 111, 112

M

Message Store Interceptor, 121
Model, 9, 11, 12, 13, 14, 15, 28, 112, 113, 114, 119, 141, 145, 149, 152, 153, 279, 326, 343, 503, 521, 522, 530, 542, 623
Model-Driven Interceptor, 112, 114, 119, 145, 153
MultipartWrapperRequest, 419
MutableTilesContainer, 549, 550

N

Namespace configuration, 583
Non-Field Validator, 331, 346, 361, 362, 365

O

ObjectFactory, 18, 62, 119, 427, 429, 434, 579, 581
OGNL, 3, 13, 20, 21, 23, 24, 26, 55, 56, 106, 115, 164, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 182, 183, 184, 186, 185, 187, 188, 189, 191, 289, 294, 295, 298, 300, 323, 324, 331, 338, 364, 435
OgnlContext, 173, 182

P

Package Configuration, 33, 584, 603
package tag, 604
PageContext, 57, 201, 290, 291
ParameterAware interface, 65, 120
Parameters Interceptor, 114
patterns, 3, 34, 62, 180, 296, 410, 411, 459, 464, 466
plug-ins, 17, 500
POJO Actions, 628
Prepare Interceptor, 115, 116, 147
PreResultListener, 118
Projection, 178
PropertyResourceBundle class, 639

R

Redirect Action Result, 287, 296, 297
Redirect Result, 287, 295, 296, 299
RequestDispatcher class, 289, 290
RequestProcessor class, 93, 94, 624
Result configuration, 583, 603
Result interface, 279, 280, 285, 287, 311
Result Types, 282, 285
ResultConfig class, 280
ResultTypeConfig class, 281
Roles Interceptor, 117

S

saveLocale(), 393
Scoped-Model-Driven Interceptor, 119, 149
Security, 20, 453, 454, 455, 464, 467, 471, 475, 476, 478, 482, 484
ServletActionContext class, 56, 57, 58
Servlet-Config Interceptor, 120, 151, 154
ServletContext, 55, 57, 58, 120, 495, 550, 556

ServletRequestAware interface, 66, 79, 85, 644
ServletResponseAware interface, 66
Servlets, 4, 6, 7, 8, 13, 17, 289, 296, 456, 464, 495, 507, 537, 541, 551, 583
ServletTestCase, 495, 497
Session Interceptor, 105
SessionAware interface, 64, 85, 151, 185, 531
SiteGraph Plugin, 421
SiteMesh Plugin, 423, 424
Spring Plugin, 426, 427
SpringObjectFactory, 428, 580
Static Parameters Interceptor, 121
Stream Result, 287, 297
Struts 1 Plugin, 430, 438, 443, 446
Struts configuration file, 30, 69, 94, 96, 279, 294, 298, 502, 647
struts.properties, 18, 67, 105, 132, 134, 145, 215, 217, 265, 386, 387, 395, 399, 401, 419, 420, 427, 434, 437, 560, 576, 579, 584, 598, 599, 626, 631, 643, 647, 648
struts.xml, 18, 20, 26, 27, 28, 29, 30, 32, 37, 38, 51, 52, 53, 69, 76, 80, 84, 87, 89, 94, 95, 96, 97, 100, 102, 103, 105, 107, 108, 109, 112, 114, 116, 117, 118, 119, 122, 123, 124, 125, 126, 128, 129, 130, 132, 134, 137, 138, 139, 144, 153, 157, 189, 190, 205, 206, 215, 219, 222, 228, 265, 267, 273, 279, 280, 282, 283, 284, 285, 286, 288, 296, 302, 305, 306, 307, 313, 314, 315, 347, 350, 356, 357, 363, 364, 367, 369, 373, 374, 401, 402, 408, 410, 423, 427, 428, 430, 434, 437, 439, 441, 442, 445, 446, 447, 478, 500, 501, 502, 503, 533, 534, 557, 558, 560, 568, 569, 570, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 598, 601, 602, 603, 604, 605, 606, 607, 608, 610, 611, 614, 622, 625, 629, 630, 643, 647, 648
struts-config.xml file, 629, 639, 643, 647

struts-default package, 49, 96, 97, 100, 283, 284, 286, 305, 351, 582, 587, 589, 604, 606, 607
struts-default.xml, 33, 50, 51, 53, 69, 76, 94, 95, 96, 97, 98, 100, 104, 107, 114, 116, 120, 137, 144, 219, 222, 228, 283, 284, 286, 305, 306, 314, 351, 401, 408, 428, 437, 442, 576, 577, 578, 579, 581, 582, 587, 589, 604, 606, 614, 622, 629, 630, 647, 648
struts-plugin.xml, 408, 411, 416, 418, 421, 426, 429, 430, 431, 437, 442, 445, 448, 555, 557, 558, 568, 570, 576, 581
StrutsTilesListener class, 555, 556
StrutsTilesRequestContext class, 555

T

tags, 11, 13, 16, 26, 27, 29, 30, 31, 37, 64, 67, 70, 76, 77, 81, 87, 132, 134, 135, 139, 140, 141, 150, 151, 156, 159, 160, 166, 168, 177, 187, 195, 196, 197, 202, 203, 206, 207, 208, 209, 210, 212, 213, 214, 215, 216, 217, 220, 221, 222, 223, 225, 226, 227, 228, 229, 230, 233, 234, 235, 236, 237, 245, 246, 253, 256, 257, 258, 261, 262, 264, 265, 266, 267, 269, 271, 273, 274, 276, 285, 291, 292, 304, 346, 348, 353, 354, 361, 362, 367, 368, 387, 388, 389, 395, 396, 397, 405, 410, 415, 417, 418, 422, 424, 425, 433, 435, 438, 439, 440, 443, 456, 474, 475, 476, 500, 508, 509, 510, 511, 516, 520, 523, 537, 539, 540, 546, 547, 553, 554, 555, 559, 562, 563, 564, 565, 567, 568, 571, 575, 578, 579, 580, 586, 598, 599, 603, 604, 606, 611, 613, 616, 617, 618, 621, 624, 630, 631, 635, 636, 637, 648, 649, 650
Template, 20, 235, 261, 263, 264, 523, 537, 541, 546, 547, 562, 564
Test interface, 488
TestCase class, 488, 495
TextProvider interface, 47
Third Party Plugins, 433

Tiles, 18, 409, 432, 433, 504, 544, 545, 546, 547, 548, 549, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 567, 568, 570, 576
Tiles Plugin, 432, 544, 553, 555, 557, 559, 560, 561, 567
tiles.xml, 547, 548, 552, 558, 560, 562, 563, 564, 567, 568, 570, 575
TilesContainerFactory class, 550, 555, 556
TilesFilter class, 551
TilesListener, 432, 549, 550, 551, 552, 553, 555, 556, 557, 561
TilesResult class, 555, 556, 558
TilesServlet class, 551
Timer Interceptor, 122
Token Interceptor, 123, 124
Token Session Interceptor, 124
Tomcat, 20, 27, 34, 35, 36, 67, 272, 402, 457, 458, 459, 461, 467, 469, 539, 630
Transport level security, 453
TypeConverter, 167, 618, 621

U

UI Tags, 233, 235, 256, 261, 265, 266, 269, 271, 274
UIBean class, 235, 261, 262, 263, 264
Unicode, 414
Unit test, 487, 498
Unit testing, 487

V

Validateable interface, 46, 127, 145, 645
validation, 3, 11, 13, 16, 17, 21, 23, 49, 50, 52, 82, 92, 97, 99, 101, 118, 125, 126, 127, 128, 133, 158, 161, 162, 163, 236, 315, 317, 320, 321, 323, 324, 325, 326, 328, 329, 334, 335, 336, 338, 341, 343, 344, 345, 346, 348, 351, 352, 353, 355, 357, 361, 362, 364, 366, 367,

- 369, 374, 375, 377, 378, 381, 416, 430, 431, 432, 446, 460, 468, 505, 508, 519, 587, 594, 595, 610, 611, 612, 613, 614
- Validation Annotations, 336
- Validation Interceptor, 125, 126, 128, 158, 162, 163
- validation rules, 52, 125, 128, 159, 161, 163, 334, 335, 336, 345, 351, 353, 357, 594, 610, 614
- validation.xml file, 159, 161, 162, 163, 326, 334, 335, 351, 352, 357, 364, 369
- ValidationAware interface, 46, 104, 109, 125, 127, 145, 350, 351
- Validator interface, 318
- ValidatorFactory, 318, 331, 359
- validators.xml, 318, 332, 334, 359
- ValueStack, 20, 21, 22, 24, 26, 48, 55, 56, 57, 60, 114, 129, 145, 164, 182, 294, 616
- Velocity, 11, 13, 15, 16, 18, 20, 168, 258, 279, 287, 293, 298, 299, 302, 421, 422, 425, 432, 504, 521, 536, 537, 538, 539, 540, 541, 542, 543
- Velocity Result, 287, 293, 298, 299
- VelocityPageFilter, 425, 426
- view components, 29, 395, 500, 528, 545, 564, 565, 567, 632, 635

W

- WEB-INF directory, 539, 602, 629
- WebRequest, 494, 495, 497
- WebWork 2, 11, 18
- WebWork2, 10, 11, 12, 13, 16, 17, 18, 24, 182, 623, 624, 625, 653
- Workflow Interceptor, 126, 127, 128, 162

X

- XSLT Result, 299, 300
- XWork Validation Framework, 317, 318

Z

- Zero Configuration, 591