



The Book

The O'Reilly Struts book (title not set yet) will be available sometime in the 3rd quarter of 2002. Published by O'Reilly, this book covers both Struts 1.0 and 1.1. The beta or draft chapters are being made available for download to facilitate an early public review process for the material.

The Author

Chuck Cavaness is a Senior Technologist at the S1 Corporation. His expertise spans server-side Java, distributed object computing, and application servers. Chuck is the most recent moderator for the "Java in the Enterprise" discussion forum hosted by JavaWorld. He spent several years writing Smalltalk and CORBA applications, and he has taught courses in object-oriented programming at Georgia Tech. He's written articles for JavaWorld and InformIt.com. He has also been the technical editor for many J2EE books, including *Using JavaServer Pages and Servlets* (Que 2000) and *Special Edition Using Java 2 Enterprise Edition* (Que 2001). Chuck earned his degree in computer science from Georgia Tech. His current interests focus on building presentation-tier frameworks based on the Apache Struts project.

Chuck is the co-author of *Special Edition Using Enterprise JavaBeans 2.0* (Que, 2001) and *Special Edition Using Java 2 Standard Edition* (Que, 2000). His next book, *Struts* (O'Reilly, 2002), will be available sometime in the 3rd quarter.

Spread to you by Asmodeous <asmodeous77@hotmail.com>

Table of Contents

Chapter 1

Introduction.....	9
Brief History of the Web	9
What are Java Servlets?.....	11
JavaServer Pages Technology	14
JSP Model 1 and Model 2 Architectures	16
Why is Model-View-Controller So Important?.....	18
What is a Framework?.....	20
Creation of the Struts Framework	21
Alternatives to Struts	22

Chapter 2

Inside the Web Tier	29
An Architecture Overview	29
The HTTP Request/Response Phase	34
Struts and Scope	40
Using URL Parameters.....	42
Forward versus Redirect.....	42

Chapter 3

Overview of the Struts Framework51

A Banking Account Example	51
Looking at the Big Picture.....	57
Struts Controller Components	58
Struts Model Components	66
The Struts View Components.....	70
Multiple Application Support.....	81
Summary	81

Chapter 4

Configuring the Struts Application83

Introduction to the Storefront Application	83
What is a Web Application?.....	85
The Web Application Directory Structure.....	86
Web Application Deployment Descriptor	88
Configuring the <i>web.xml</i> file for Struts	91
The Struts Configuration File.....	101
The <i>org.apache.struts.config</i> Package.....	102
Struts Console Tool	121
Reloading the Configuration Files.....	122

Chapter 5

Struts Controller Components123

The Controller Mechanism.....	124
The Utilities Classes	151

Chapter 6

Struts Model Components155

The “M” in MVC	155
What is a Business Object?	160
Persistence	162
What does Struts Offer For the Model?.....	163
Building the Storefront Model.....	164

Chapter 7

Struts View Components.....190

What exactly is a View?	190
What are ActionForms?.....	197
Using ActionErrors.....	207
Performing Presentation Validation	212
Using Dynamic ActionForms	213
Looking Ahead to JavaServer Faces	215

Chapter 9

Extending the Struts Framework.....218

What are Extension Points?.....	218
General Extension Points	219
Controller Extension Points.....	222

Extending View Components	227
Extending the Model Components	228
Downsides to Extending the Framework.....	231

Chapter 10

Exception Handling232

Java Exception Handling.....	232
System versus Application Exceptions.....	237
Using Chained Exceptions	238
Exception Handling provided by Struts.....	241
Tying Up the Loose Ends	253
Exceptions in Custom Tags	254
Internationalized Exception Handling	254
Conclusion.....	255

Chapter 11

Using the Struts Validator256

The Need for a Validation Framework.....	256
Installing and Configuring the Validator.....	257
Overview of Regular Expressions	265
Using an <code>ActionForm</code> with the Validator.....	265
Using the Validator Framework	269
Creating your own Validation Rules	270
The Validator and JSP Custom Tags.....	272
Internationalizing the Validation	275

Using the Validator Outside of Struts.....	276
--	-----

Chapter 12

Internationalization and Struts279

What is Internationalization?.....	280
Support for I18N in Java	281
Internationalizing your Struts Applications.....	288
Exception Handling and Internationalization	292

13293

Struts and Enterprise JavaBeans (EJB).....293

Implementing the Storefront Service Using EJB.....	295
Interfacing Struts to EJB	308

Chapter 14

Using Tiles320

Understanding Templates.....	320
Installing and Configuring Tiles.....	326
Overview of Tiles.....	329
The Tiles Tag Library.....	332
Using Definitions	340
Internationalization Support with Tiles	344

Chapter 16

Addressing Performance.....346

What is Good Performance?.....	346
--------------------------------	-----

Performance versus Load Testing	348
Performance and Stress Testing Tools	350
Testing the Storefront Application	351
Performance and Scalability Gotcha's	359

Chapter 18

Logging in a Struts Application361

Logging in a Web Application	361
Using the Servlet Container for Logging.....	363
Jakarta Commons Logging.....	374
Using the log4j Package	377
Using Commons Logging in JSP Pages	385
Creating an Email Appender	387
The Performance Impact of log4j.....	396
Third-Party log4j Extensions.....	397
Java 1.4 Logging API.....	397

Chapter 20

Packaging Your Struts Application399

To Package or Not to Package.....	399
Deciding on How to Package Your Application	402
Packaging the Struts Application as a WAR	407
Building your Struts Applications with Ant	409
Creating an Automated Build Environment	415
FTPing and Restarting your Server	417

1

Introduction

The Struts open source framework was created to make it easier for developers to build web applications based on Java Servlet and JavaServer Pages (JSP) technologies. Just like a building must have a solid foundation from which the rest of the structure can grow, web applications should be built with the same principle in mind. The Struts framework provides developers a unified framework from which Internet applications can be based upon. By using Struts as the foundation, developers are able to concentrate on building the business logic for the application.

The Struts framework was originally created by Craig R. McClanahan and added to the *Apache Software Foundation* (ASF) in 2000. The project now has several committers from around the world and there are many developers contributing to the overall good of the framework. The Struts framework is one of many well-known and successful Apache Jakarta projects. Others include Ant, Log4J, Tomcat, and many more. The overall mission of the Jakarta project is to provide commercial-quality server solutions based on the Java platform and in an open and cooperative fashion.

Brief History of the Web

No book on web technology would be complete without a brief look at how the *World Wide Web* (WWW) has become as popular as it has today. The web has come a long way since the days when the first hypertext documents were sent to others over the Internet. In 1989, when the physicists at CERN laboratory proposed the idea of sharing research information between researchers using hypertext documents, they had no idea of how big and essential the web would become to the daily life for much of the industrialized world. It has now become an accepted part of our vernacular.

It took a couple of years before the benefits of using the web became clear to others outside of the CERN laboratories, but as we all know, it eventually erupted into what we see and use today. From its beginnings, the web was designed for viewing hypertext documents, not editing them. However, it was a natural evolution to want the ability to also edit a document and submit the changes. In short time, the *Common Gateway Interface* (CGI) was created. CGI is a standard that allows web servers to interact or interface with external applications in such a way that hypertext pages no longer have to be static. A CGI program can retrieve results from a database and insert those results as a table in a hypertext document. Likewise, data entered into a hypertext page can also be inserted into the database. This opened up infinite possibilities and in fact, started the boon that was the Internet craze of the mid-nineties and lasts even today.

Although CGI applications are very good at what they do, there are some serious limitations with its approach. For one thing, CGI applications are very resource-intensive. For every request that comes from a browser, a new Operating System heavyweight process is created to handle the request. Once the CGI script is finished executing, the process has to be reclaimed by the OS. This constant starting and stopping of heavyweight processes is terribly inefficient. You can imagine how bad the response time might be if there are hundreds of concurrent users making requests to the same web application. Another major limitation of CGI is that it can't link to other stages of web server request processing easily once it begins executing. This is because it's running in a separate process from the web server and becomes difficult to handle things such as authorization, workflow, and logging. Other limitations of CGI have to do with the scripting languages available for CGI applications. Although CGI applications can be built in many different languages, the Perl programming language has been widely used to create web applications and in fact is still used by many Internet sites today. However, many web developers wanted more out of a web programming language.

There have been alternatives to standard CGI applications put forward. One is called FastCGI. *FastCGI* is a language independent extension to CGI that doesn't have the same process model that standard CGI uses. It's able to create a single heavyweight process for each FastCGI program, allowing multiple requests to run within the same process space. However, when clients interact with the same FastCGI program concurrently, it needs to create a pool of processes to handle each concurrent request. This is not much better than standard CGI. Another problem with FastCGI applications is that it's only as portable as the language in which they are written. Other solutions were provided by mod_perl for Apache, NSAPI for Netscape, and ISAPI for Microsoft's IIS web server. While these solutions might offer better performance and scalability over standard CGI programs, they are very proprietary.

Around 1997, while the JavaTM language was experiencing tremendous growth and use by application developers, the Java Servlet technology was created. This new web technology opened up an entirely new avenue for web developers to explore.

What are Java Servlets?

Java Servlets have become the mainstay for extending and enhancing web applications using the Java platform. They provide a component-based, platform-independent method for building web applications. Servlets don't suffer from the same performance limitations that standard CGI applications incur. Servlets differ from the standard CGI threading model in that they create a single heavyweight process and allow each user request to utilize a much lighter-weight thread, which is maintained by the JVM, to fulfill the request. Many user requests can be threaded through the same instance of a servlet. A servlet is mapped to one or more *Uniform Resource Locators* (URLs) and when the server receives a request to one of the servlet URLs, the service method in the servlet is invoked and responds. Because each user request is associated with a separate thread, multiple threads or users, can invoke the service method at the same time. This multi-threaded nature of servlets is one of the main reasons that they are much more scalable than standard CGI applications. Since servlets are written in Java, they are also not proprietary to a platform or OS.

Another significant advantage of being written in the Java language is that servlets are able to exploit the entire suite of Java API's, including JDBC and EJB. This was one of the factors in servlets becoming part of the mainstream so quickly. There was already a rich Java library in place for them to leverage. Characteristics such as garbage collection, multi-threading, and the Java Collections libraries, and the other benefits of the Java platform have propelled servlets far ahead of its peers for web development.

Servlets are not executed directly by a web server. They require a servlet container, sometimes referred to as a servlet engine, to host the servlet. This servlet container is loosely coupled to a particular instance of a web server and together they cooperate to service requests. Figure 1-1 illustrates how a web server and servlet container cooperate to service a request from a web browser.

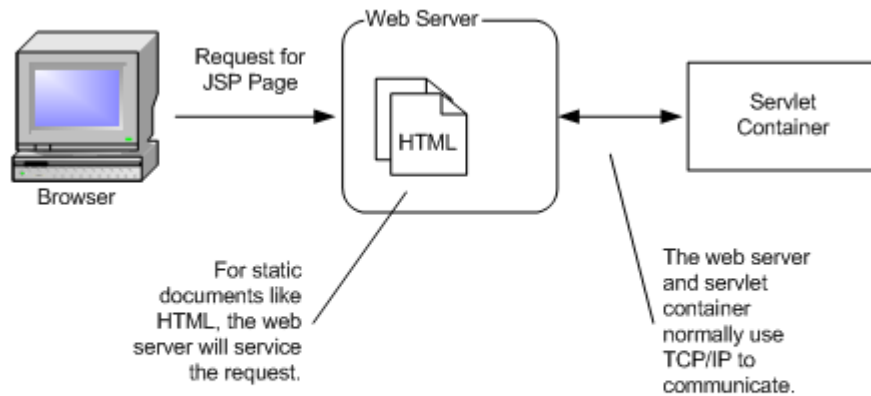


Figure 1-1. Serving a client request

Developers are free to choose from one of many servlet containers available to host their servlet in. They are not locked into a particular vendor or platform. Servlets can be ported to any one of these containers without recompiling the source code or making changes to the servlet. This leads to selecting a Best of Breed solution for web applications, which basically means that developers and organizations are free to choose specialized products or components from one or more companies for an application. The developers get the best of both worlds; the best product or component for a specialized need, while at the same time avoiding the high risk normally associated with a single solution.

There are several popular choices for servlet containers on the market. Some are standalone servlet containers that must be connected to an external web server to work and others provide both the web server and servlet container within the same product. There are even a few that are integrated into application servers and provide for much more functionality than just a servlet container. Table 1-1 lists some of the more popular servlet containers and a URL to get more information.

Table 1-1. Available Servlet Containers

Servlet Container	URL
Bluestone	http://www.bluestone.com
Borland Enterprise Server	http://www.inprise.com
iPlanet Application Server	http://www.iplanet.com
Orbix E2A (formally iPortal)	http://www.iona.com
Jetty	http://www.mortbay.com
JRun	http://www.allaire.com

Orion Application Server	http://www.orionserver.com
Resin	http://www.caucho.com
SilverStream	http://www.silverstream.com
Apache Tomat	http://jakarta.apache.org/tomcat
Weblogic Application Server	http://www.bea.com
WebSphere	http://www-4.ibm.com/sfotware/webservers/appserv

For a more complete listing of available servlet containers, visit Sun's servlet industry momentum web site <http://java.sun.com/products/servlet/industry.html>.

Although servlets are great at what they do, it became apparent very quickly that hardcoding HyperText Markup Language (HTML) output in a servlet as a response to a request had some serious limitations. First and foremost, it was very hard to make changes to the HTML because for every change, a recompilation of the servlet had to take place.

Secondly, supporting different languages is difficult to do because the HTML is hardcoded. Determining the user's language, region, and optional variant and then displaying the output is not easily accomplished. Many web applications built with servlets avoid the entire issue of *Internationalization** (I18N) by having different servlets, one for each supported Locale.

Finally, because HTML was embedded within the servlet, this caused a problem with responsibilities. Web designers build HTML pages. They are not usually experienced Java programmers, let alone skilled at object oriented design and programming. On the other hand, Java programmers should do what they do best and this typically isn't HTML. Although many developers have been crossed-trained in both skill sets, these are two fields that are better left separated. By mixing HTML within the servlet itself, it becomes very hard to separate the duties. Regardless of the skill set of the developers, it becomes very difficult to separate the lines of development with this approach. Even when a developer has the necessary skills to perform both functions, modifications to the page layout meant recompilation, which adds to development time.

Obviously, servlet programming is such a broad topic; it can't be covered in great detail here. If you feel that you need more information on Java Servlet technology, a great source of material is Jason Hunter's *Java Servlet Programming*, 2nd edition book published by O'Reilly. You can also find more information on the following web site: <http://java.sun.com/products/servlet/index.html>.

The Servlet specification can be downloaded from the Sun servlet site at <http://java.sun.com/products/servlet/download.html>.

* Internationalization is commonly referred to as I18N because the word begins with the letter I, ends with the letter N and contains 18 characters in between.

JavaServer Pages become the next step in the linear progression of developing web technologies based on the Java platform. The introduction of JSP pages as they are commonly referred to, help to alleviate the servlet limitations mentioned earlier and opened up many new doors for web developers. Unfortunately, it produced just as many uncertainties as well.

JavaServer Pages Technology

The first thing to understand about JavaServer Pages technology is that it's a natural extension to the servlet technology. In fact, after some pre-processing by a translator, JSP pages end up being nothing more than a Java servlet. This is a point that many developers have a hard time understanding in the beginning. JSP pages are text documents that have a *.jsp* extension and contain a combination of static HTML and Extensible Markup Language (XML) like tags and scriptlets. The tags and scriptlets encapsulate the logic that generates the content for the page. The *.jsp* text files are pre-processed and are turned into *.java* files. At this point, a Java compiler compiles the source and creates regular servlet byte code that can be loaded and ran as a servlet.

The translator that turns the *.jsp* file into a *.java* file takes care of the tedious work in creating a Java servlet from the JSP page. Figure 1-2 illustrates how a JSP page is translated and compiled into a servlet.

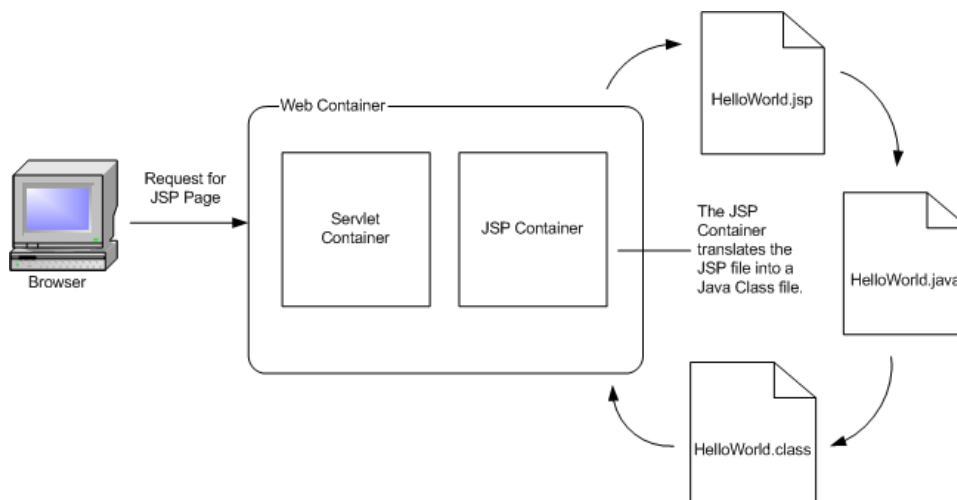


Figure 1-2. A JSP page is translated and compiled into a Java Servlet

JSP technology has become an extremely popular solution for building web applications using the Java platform.

JSP offers several advantages over its competitors:

- JSP is a specification, not a product. Vendors are allowed to build competing JSP containers, which helps in choosing a “Best of Breed” approach.
- JSP pages are compiled, not interpreted. This leads to better performance and more efficient processing.
- JSP pages support both scripting and access to the full Java language. JSP pages can also be extended through the use of custom tags.
- JSP is an integral part of the J2EE suite of APIs and is compatible and complimentary with all included technologies.
- JSP pages share the "Write Once, Run Anywhere™" characteristics of Java technology.

One of the limitations of hardcoding HTML inside of servlets mentioned in the previous section is the problem of separating page design and application logic programming responsibilities. This separation is easier to accomplish with JSP pages because HTML designers are free to create web pages with whatever tools they are accustomed to. When they are comfortable with the page layout, JSP developers are then able to insert JSP scriptlets and custom tags and save the file with a *.jsp* extension. That's pretty much all there is too it. When it comes time to change either the page layout or page logic, the developer would modify the portion of the JSP page necessary and allow the JSP page to automatically be recompiled. For HTML developers, many of today's popular tools are capable of working with JSP and custom tags.

JSP Scriptlets or Tag Libraries?

There are many developers who believe custom tags should be used in JSP pages, rather than scriptlets or expressions. The rationale is:

- Scriptlets mix logic with presentation
- Scriptlets break the separation of roles
- Scriptlets make JSP pages difficult to read and maintain

Custom tags on the other hand centralize code in one place and help maintain the separation of responsibilities. They also support the concept of reuse because the same tag can be inserted into multiple pages, while having the implementation reside in a single location. There's no redundancy or copy-and-paste programming like there is with JSP scriptlets.

Together, JSP pages and servlets combine for an attractive alternative to other types of dynamic web programming. Because they are both based on the Java language, they offer platform independence, extensibility into the enterprise, and most importantly, ease of development.

You can find more information about the JavaServer Pages technology at Sun JSP site: <http://java.sun.com/products/jsp>. The JavaServer Pages specification can be downloaded from <http://java.sun.com/products/jsp/download.html>.

JSP Model 1 and Model 2 Architectures

The early JSP specifications presented two approaches for building web applications using JSP technology. These two approaches were described in the specification as JSP *Model 1* and *Model 2* architectures. Although the terms are no longer used in the JSP specification, their usage throughout the web tier development community is still widely used and referenced.

The two JSP architectures differed in several key areas. The major difference was how and by which component the processing of a request was handled. With the Model 1 architecture, the JSP page handles all of the processing of the request and is also responsible for displaying the output to the client. This is better seen in Figure 1-3.

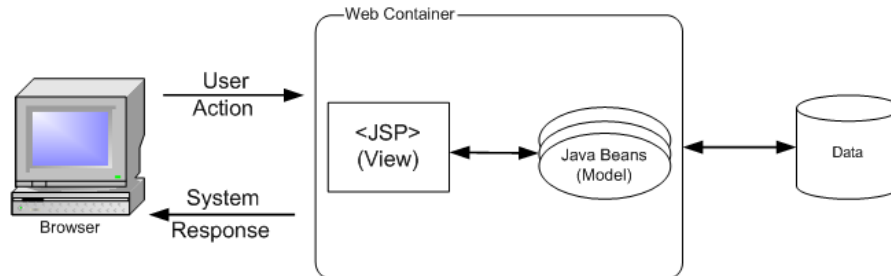


Figure 1-3. JSP Model 1 Architecture

Notice that in Figure 1-3 there is no servlet involved in the process. The client request is sent directly to a JSP page, which may communicate with JavaBeans or other services, but ultimately the JSP page selects the next page for the client. The next view is either determined based on the JSP selected or parameters within the client's request.

In direct comparison to the Model 1 approach, in the Model 2 architecture, the client request is first intercepted by a servlet, most often referred to as a *Controller servlet*. The servlet handles the initial processing of the request and also determines which JSP page to display next. This approach is illustrated in Figure 1-4.

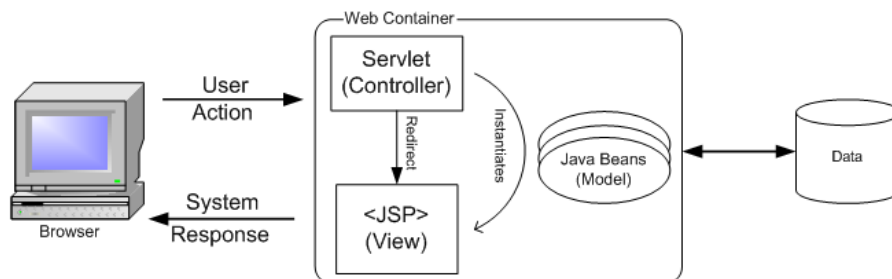


Figure 1-4. JSP Model 2 Architecture

As you can see from Figure 1-4, in the Model 2 architecture, a client never sends a request directly to a JSP page. The controller servlet acts as sort of a traffic cop. This allows the servlet to perform front-end processing like authentication and authorization, centralized logging, and possibly help with Internationalization. Once processing of the request has finished, the servlet directs the request to the appropriate JSP page. How exactly the next page is determined can vary widely across different applications. For example, in simpler applications, the next JSP page to display may be hardcoded in the servlet based on the request, parameters, and current application state. In other more sophisticated web applications, a workflow/rules engine may be used.

As you can see, the main difference between the two approaches is that the Model 2 architecture introduces a controller servlet that provides a single point of entry and also encourages more reuse and extensibility than Model 1. With the Model 2 architecture, there is also a clear separation of the business logic, presentation output, and request processing. This separation is often referred to as a *Model-View-Controller* (MVC) pattern. While the Model 2 architecture might seem overly complicated, it can actually simplify an application greatly. Web applications built using the Model 2 approach are generally easier to maintain and can be more extensible than comparable applications built around the Model 1 architecture.

All of this doesn't mean that applications built using the Model 1 approach are incorrectly designed. The Model 1 architecture might be the best decision for smaller applications that have simple page navigation, no need for centralized features, and are fairly static. However, for more larger enterprise-size web applications, it would be more advantageous to utilize the Model 2 approach.

Why is Model-View-Controller So Important?

Model-View-Controller is an architectural pattern that by it self has nothing to do with web applications directly. As we saw from the previous section, the JSP Model 2 approach is clearly about separating responsibilities in a web application built using Servlet and JSP technologies. Allowing a JSP page to handle the responsibilities of receiving the request, executing some business logic, and then determining the next view to display can really make for an unattractive JSP page, not to mention the problems this entanglement causes for maintenance and extensibility. By having components within a web application that have very clear and distinct responsibilities, the development and maintenance on an application can be made more efficient. This is also true for software development as a whole.

The MVC pattern is categorized as a design pattern in many software design books. Although there is usually much disagreement on the precise definition of the pattern, there are some fundamental ideas.

The MVC pattern has three key components:

The Model Component

Responsible for the business domain state knowledge

The View Component

Responsible for a presentation view of the business domain

The Controller Component

Responsible for controlling flow and state of the user input

Normally with the MVC pattern, there's a form of event notification that takes place to notify the view when some portion of the model changes. However, since a browser in a typical web application has a stateless* connection, the notification from the model to the view can't easily occur. Of course, an application could perform some type of push action to push data changes all the way to a client; but this doesn't and probably shouldn't happen in most web applications. A user can close a browser anytime and there isn't warning or notification sent to the server. There's a great deal of overhead necessary to management remote clients from the server side. This type of behavior is overkill for typical B2C and B2B web applications.

With standard web applications, a client must perform another request back to the server to learn about any changes to the model. For example, if a user is viewing the pricing information for an item and at the same time, the administrator changes the price for that item, the user isn't going to know this until they refresh the page or place the item into their shopping cart.

The MVC Model

Depending on the type of architecture of your application, the model portion of the MVC pattern can take many different forms. In a two-tier application, where the web tier interacts directly with a data store like a database, the model classes may be a set of regular Java objects. These objects may be populated manually from a result set returned by a database query or they can even be instantiated and populated automatically by an *Object-to-Relational Mapping* (ORM) framework like *TopLink* or *CocoBase*.

In a more complex enterprise application where the web tier communicates with an EJB server for example, the model portion of the MVC pattern might be Enterprise JavaBeans. Although the EJB 2.0 Specification made some improvements in performance through the use of local interfaces, there can still be a significant performance impact if the web tier attempted to use entity beans directly as the model portion of the application. In many cases, JavaBeans are returned from Session beans and used within the web tier. These JavaBeans are commonly referred to as value objects and are used within the views to build the dynamic content.

The MVC View

The views within the web tier MVC pattern typically consist of HTML and JSP pages. HTML pages are used to serve static content, while JSP pages can be used to serve both static and dynamic content. Most dynamic content is generated in the web tier. However,

* Web applications are considered stateless because the browser doesn't typically maintain an open socket to the web server. However, a web application may still maintain session data for a user or even store data within the browser on behalf of the user.

some applications may require the need for client-side JavaScript. This does not interface or infringe upon the MVC concept.

[\[Editors: This section is too short and I will need to discuss this a little more. Chapter 7 is dedicated to discussing what Struts offers in the way of View Components. This section should just discuss it from a generic web application point of view. Any feedback is appreciated for this section\]](#)

The MVC Controller

The controller portion of the web tier MVC design is generally a Java servlet. The controller in a web tier application performs the following duties:

1. Intercepts HTTP requests from a client.
2. Translates the request into a specific business operation to perform.
3. Either invokes the business operation itself or delegates to a handler.
4. Helps to select the next view to display to the client.
5. Returns the view to the client.

The *Front Controller* pattern, which is part of the J2EE Design Patterns*, describes how a web tier controller should be implemented. Since all client requests and responses go through the controller, there is a centralized point of control for the web application. This aides in maintenance and when adding new functionality. Code that would normally need to be put in every JSP page can be put in the controller servlet, since it processes all requests. The controller also helps to decouple the presentation components (views) from the business operations, which also aids development.

What is a Framework?

I have been throwing the word *framework* around in this chapter without having really defined what exactly it is or how it adds value in software development. In its simplest form, a framework is a set of classes and interfaces that cooperate to solve a specific type of software problem. A framework has the following characteristics:

- A framework is made up of multiple classes or components, each of which may provide an abstraction of some particular concept
- The framework defines how these abstractions work together to solve a problem

* The J2EE Design Patterns can be found at <http://java.sun.com/blueprints/patterns/index.html>

- The framework components are reusable

A good framework should provide generic behavior that can be utilized across many different types of applications.

There are many interpretations of what constitutes a framework. Some might consider the classes and interfaces provided by the Java language a framework, but it's really a library. There's a subtle, but very distinct difference between a software library and a framework. With a software library, your application is the main code that executes and it invokes routines on the library. With a framework, it contains the executing routines and invokes operations onto your extensions through inheritance and other means. The places where the framework can be extended are known as *extension points*. A framework is commonly referred to an "upside-down" library because of the alternate manner in which it operates. Figure 1-5 illustrates the subtle differences.

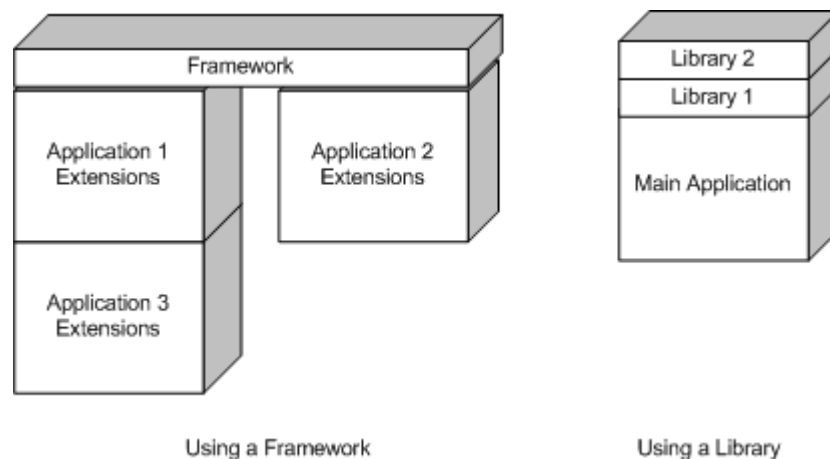


Figure 1-5. A framework has subtle differences from a library

Creation of the Struts Framework

By now you should have a foundation for JSP and Servlet technology and you should also understand the benefits that the Web MVC design and JSP Model 2 architecture adds to a web application. This section provides a little background and history on the Struts framework, which is an implementation of all of these ideas. This section will not leap into any technical details of the framework; that's saved for the later in the book. Instead, this section will describe the conditions under which the framework itself was created and how it has evolved over the past couple of years.

As was already mentioned, the Struts framework was created by Craig R. McClanahan and donated to the ASF in 2000. Craig is deeply involved in the expert groups for the

Servlet and JSP specifications and has written a large portion of the Tomcat 4.0 implementation. He also speaks at various conferences, such as *JavaOne* and *ApacheCon*.

Since Struts was first created, there have been several committers* to join the Struts bandwagon, and even more developers that have volunteered their time and effort to increase the value of the framework and keep it moving forward. The framework has already gone through several beta releases and a couple of *General Availability* (GA) releases and although there have been many new features added, the framework hasn't strayed far from the core idea.

The Struts group always welcomes new participants to the project. Any developer wishing to contribute to the project only need to volunteer their time and knowledge and that's all there is to it. To become a contributor, it's recommended that you first join the Struts User mailing list. If you like what you see there, then take a look at the Struts Developers mailing list. This is the best way to get started and become familiar with the direction of the framework. You should read the mailing list guidelines first before joining. You can find the guidelines at <http://jakarta.apache.org/site/mail.html>.

After reading the guidelines, you can join one or more of the Apache project mailing lists, including Struts, from the URL: <http://jakarta.apache.org/site/mail2.html>.

The main project web site for Struts is located at <http://jakarta.apache.org/struts>. For more information on downloading and installing Struts, see Appendix B.

Alternatives to Struts

Before we get too far into the Struts framework discussion in the coming chapters, we should talk about what alternatives are available to using Struts. Since Struts is an implementation of the Model 2 approach, you can safely assume that there are other implementations available. The problem in trying to compare any two software products in general is that feature sets are rarely ever the same. It's sometimes like trying to compare "apples" with "apples that have wheels". The goal of the two may be the same, but one may have features that are missing from the other.

Comparison between web application frameworks is made worse because many are open source and have a tendency to vanish as quickly as they appear. That's in no way saying that open source projects are any more or less volatile than their well-funded counterparts. It's true however, that you must be careful when choosing an open source framework on which to base your entire project. Make sure that you have a copy of the

* A committer is a developer who provides expert direction and advice to steer the Struts framework on the correct course. A committer has the ability to modify the source code repository and can cast votes that affect the future of the framework.

source code, in case you ever have to take ownership in the complete maintenance of the framework. It's always a valuable experience to compare feature sets of software products, open source or not, when choosing any framework or component for an application.

Because there are so many approaches to outputting a presentation view to a client, it's very difficult to categorize the different approaches. The goal of this section is to briefly introduce some other close and maybe not so close alternatives to Struts. Because versions and features may change with each new release, you should do your own research for the solution(s) that you are interested in. This list of alternatives is by no means exhaustive. However, it will provide a launching pad for you to perform your own research about alternatives.

The other point to make before we list the alternatives is that only solutions based on or around the Java platform are listed. It's assumed that you know Microsoft offers a competing technology based on *Active Server Pages* (ASP) technology, for example. Although the goal of ASP is similar to JSP, ASP or ASP+ is not compared here. That's better left for a book on JSP and Servlets. Even more so, the Struts framework goes way beyond what is offered by JSP alone and comparing ASP or other similar technologies would be the wrong comparison.

[\[Editors: I have done as much research on each of these technologies as time and documentation has allowed. If you have used one or more of these, and have significant pros and/or cons, please feel free to include them and I will update the section. I've tried to be fair, but was unable to really give any one a full test-drive.\]](#)

[I'm expecting this section to generate a great deal of positive and negative feedback as everyone tries to get in their favorite framework. I understand that and ready for it. I promise to update these sections during author review with the latest information that I can find and sample.](#)

[If you feel that I should just list each one in a row of a table and skip the commentary, let me know and if the majority agrees, then I'll change it.](#)

[chuck](#)

[1](#)

Building your own framework

At first, it might seem strange to see "Building your own framework" as an alternative to Struts. Why in the world would you want to build something like this from scratch, when it already exists in many different forms? The answer is the same reason the other open-source or commercial products are started. The available selection of products might just not be close enough to the desired framework and the decision is to build it in-house. Other times, the decision is completely out of the developer's hands and there's not much of a choice. It's happened to all of us, at one time or another.

There's a concept in software development called "The Not Invented Here Syndrome". It's where a software development team refuses to use a particular technology or software component that they didn't create themselves. In many cases, locating and eliminating dependencies is the correct course of action. If your application depended on a third-party framework and that framework stopped being supported, that's a problem. Every software package that an organization buys or gets from a third-party introduces more risk into a project. Often, this additional risk is necessary. However, this is not always true.

The best advice that one can give regarding building your own framework is to be honest and ask yourself several questions.

1. Have I taken the time to inspect what's available and build a prototype using the framework?
2. What does my application need that doesn't exist in one of the available frameworks?
3. For anything that I need that isn't available in one of the frameworks, can I extend the framework or find it from another source and add it?
4. Do I know as much about building this type of framework as the number of developers that have been working on the available ones?

Depending on honest answers to these questions, you might find that building your own framework isn't the best decision. A good guideline that many in the software development industry agree with is that if it pertains to your core business, then build it in house. If the software component is not directly related to your business, but is a generic type of framework, then go get it from somewhere else. Play to your developer's strengths and minimize their weaknesses.

Barracuda

The *Barracuda* presentation framework is a type of Model 2 architecture similar to Struts, but seems to go a step further and provides a model event notification mechanism. Unlike with a strictly JSP approach, the Barracuda framework touts to have created a template engine component, which is supposed to allow for more flexibility and extensibility. The framework leverages code-content separation provided by the XMLC approach of creating user interfaces. XMLC is a Java-based compiler that uses either an HTML or XML document and creates Java classes that can recreate the document when executed. The generated Java classes can be used to insert dynamic content into the document at runtime by manipulating Document Object Model (DOM) interfaces. The separation of markup and application logic allows for web designer to focus on markup and programmers to focus on coding.

Arguably, the one downside to using a framework similar to this one is that it might be a little steeper learning curve for developers, although one can argue that Struts is no walk

in the park. The involvement of XMLC and the fact that Java classes are created from HTML or XML documents might confuse less experienced developers.

The Barracuda framework has recently been released 1.0. You can find more information on the Barracuda presentation framework at the URL:

<http://barracuda.enhydra.org>

Cocoon

Stefano Mazzocchi founded the *Cocoon* project in January 1999 as an open source project under the ASF. The goal of Cocoon is to help the separation of content style, logic, and management functions for an XML based web site. Cocoon leverages XML, XSLT, and SAX technologies to help create, deploy, and maintain XML server applications. Cocoon is currently at release 2.0. Most types of data sources, including RDBMS, LDAP, File Systems, are supported. More information on Cocoon can be found at the URL:

<http://xml.apache.org/cocoon>

Expresso

The *Expresso* framework from Jcorporate is an application development framework that provides a component framework for developing database-driven web applications. The Expresso framework can be integrated into Struts and adds capabilities for security, *Object-to-Relational Mapping* (ORM), background job handling and scheduling, and many other features. Expresso can be classified as a companion product to Struts, rather than a competitor. It is currently as release 4.0

More information on the Expresso framework can be found at the following URL:

<http://www.jcorporate.com>

Freemarker, Velocity, and WebMacro

These three products are grouped together because they all represent similar types of template engines.

Freemarker is an open source HTML template engine for Java servlets. You store HTML in templates, which eventually get compiled into template objects. These template objects then generate HTML dynamically, using data provided by servlets. It uses its own template language and claims speeds approaching static HTML pages. The software is free and licensed under the GNU Library Public License. It is currently at release 2.0.

Velocity is a Java-based template engine that is similar in many ways to Freemarker. Velocity is another Jakarta project similar to Struts.

Velocity's is capable of performing more functionality than just dynamic content for web sites. It can generate SQL, PostScript, and XML from templates, for example. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems. Velocity also provides template services for the Turbine web application framework. Many other frameworks either support the Velocity scripting syntax or actually depend on it.

WebMacro is an open source Java servlet framework that claims to be used by several large Internet web sites like *AltaVista.com*. The WebMacro framework uses a lightweight scripting language that allows separation of how a page looks from the logic. WebMacro can be ran in standalone mode or hooked in with a servlet container. It's currently at release 1.0.

You can find more information on each of these three products from the web sites listed in Table 1-2.

Table 1.2. Template Engine URL's

Product	URL
Freemarker	http://freemarker.sourceforge.net
Velocity	http://jakarta.apache.org/velocity
WebMacro	http://www.webmacro.org

Maverick MVC Framework

The Maverick MVC framework offers the ability to render views using JSP, the Velocity scripting language, or *Extensible Stylesheet Language Transformations* (XSLT) directly. Maverick is a MVC type architecture, but actually provides a view template mechanism. It relies on one of the three scripting languages mentioned. One neat feature of Maverick is that it can use reflection on your JavaBeans in the presentation layer to create a DOM interface so that no XML generation or parsing is required. This allows for a little less clutter and probably better performance when using XSLT to generate the views.

You can find more information on the Maverick framework at the URL:

<http://mav.sourceforge.net>

Sitemesh

SiteMesh is basically a web page layout and integration system that aids is creating web sites that need a consistent look and feel. What SiteMesh does is to intercept requests to

any web page, whether it's static or dynamically generated and parse the content and generates a final page. This process is based on the well-known *Decorator** pattern.

SiteMesh is built with Servlet, JSP, and XML technologies, which makes it appropriate for J2EE applications. However it claims to also be easy to integrate with other web technologies such as CGI. More information on SiteMesh can be found at the URL:

<http://www.opensymphony.com/sitemesh>

Jakarta Turbine

Turbine is a servlet-based framework that is also an open source Jakarta project. Currently, there isn't a great deal of documentation, however it does seem similar to Struts with a few major differences. For one thing, it doesn't seem to be coupled to JSP. The focus seems to be to provide a collection of reusable components. There is a very large set of components included with the framework, but they are quite disparate. It seems to present more of a component library, but with the lacking documentation, it's hard to get a good feel on the complete architecture.

More information on Turbine can be found at the URL:

<http://jakarta.apache.org/turbine>

WebWork

WebWork is a small web application framework that utilizes something called Pull *Hierarchical Model View Controller* (HMVC). With a standard MVC design, changes made to the model are sort of pushed to the view. In the case of WebWork, the views sort of pull the data when they need it. Another interesting point is that WebWork doesn't seem to be tied to a servlet; therefore it can support other types of clients like Swing.

More information on the WebWork framework can be found at the URL:

<http://sourceforge.net/projects/webwork>

JavaServer Faces

At the time of this writing, there is a *Java Specification Request* (JSR) to create a new Java technology called *JavaServer Faces*. The specification defines the architecture and a

* The Decorator pattern is a structural design pattern mentioned in the book "Design Patterns" written by Gamma, Helm, Johnson, and Vlissides, affectionately known as the *GangOfFour*.

set of APIs for the creation and maintenance of Java server web applications. The idea is to create a standard set of JSP tags and Java classes that help developers create complex HTML forms and other *Graphical User Interface* (GUI) components based on Servlet and JSP technologies. I18N and input validation seems to be a big part of the intended support.

JavaServer Faces will be a specification and not an actual implementation. It will define a set of standard APIs and vendors will be able to create their own implementations and therefore, developers will have more than a single implementation to choose from.

The JSR indicates that they are aware that other projects like Struts, have already addressed many of the problems that this specification attempts to solve and that the JSR is aimed at creating a standard that will help unify the fragmented area. You'll have to keep your eye on this specification as it may have a huge impact on Struts and the entire web application area as a whole.

More information on the specification can be found at the URL:

<http://www.jcp.org/jsr/detail/127.jsp>

2

Inside the Web Tier

This chapter describes the physical and logical aspects of designing and utilizing a web tier for your applications. It discusses the close relationship that exists between the architectural tiers and the responsibilities that each play in the overall application. Special focus will be given to the web tier, which allows an application to communicate and interoperate with clients over the web.

The Struts framework is based on the Java Servlet Technology and to a lesser extent, JavaServer Pages, and therefore is dependent on a web container. For Struts developers, understanding how the web container processes client requests is fundamental to having a deeper understanding of the framework itself. This chapter illustrates the various components that are part of the web container and what each component's responsibilities are.

An Architecture Overview

This section presents a high-level architecture view for a Struts application. Although this section shows an architecture for an enterprise application, not all applications written using Struts will be of this size and makeup. However, this type of application does allow us to present many facets of how Struts applications may be configured.

Many applications, and J2EE application especially, can be described in terms of their *tiers*. The application's functionality is separated across these tiers, or functional layers, to provide separation of responsibility, reuse, improved scalability, and many other benefits. The separation of tiers may be a physical separation, where each one is located on a separate hardware resource, or the separation may be purely logical. In this case, one or more tiers are collocated on the same hardware resource, and the separation exists

in terms of software. Figure 2-1 illustrates the tiers that may be used by a typical Struts application.

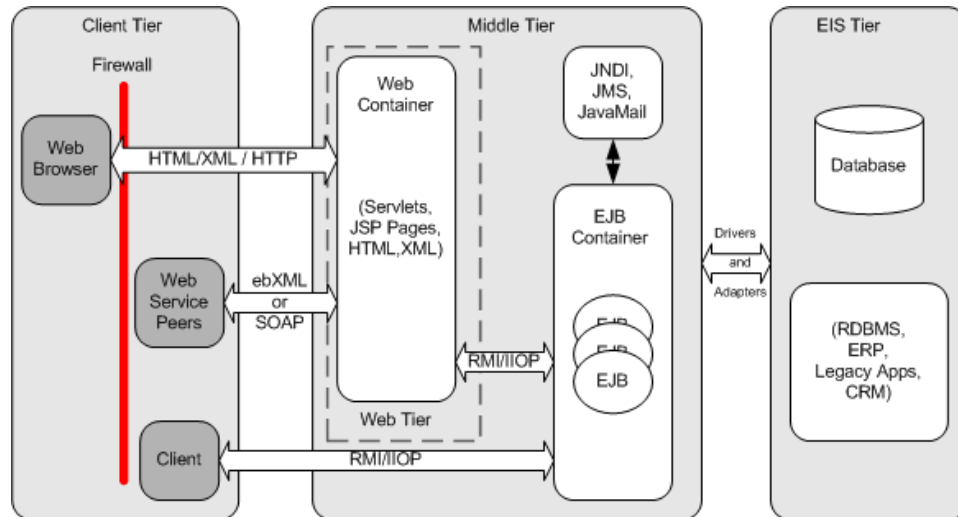


Figure 2-1. Functional Application Tiers

Not every Struts application will contain all of the tiers illustrated in Figure 2-1. For many smaller applications, the Middle tier may consist primarily of a web container that interacts directly with a database in the Enterprise Information Systems (EIS) tier.

What is a Container?

There are many different types of containers. There are EJB Containers, Web Containers, servlet containers, and so on. In general, containers provide a hosting environment for software components to run in. Containers provide general services that can be used by the components within the environment, without the need or worry of being required by the component developer. A Web Container allows servlets, JSP components, and other Java classes to be deployed and executed within the container. Services like JNDI, connection pooling, and transaction services, can be configured at the container level, and the component developers don't have to worry about the management of these resources; similar to the way in which EJB containers manages security, transactions, and bean pooling.

When using the services provided by a container, component developers may have to give up some control of the environment to the container, in trade for important services that the developers don't have to worry about building. Third-party vendors, who must follow certain guidelines that are explicitly laid out in public specifications, build these containers. Although each vendor is allowed to implement certain portions of the container in a proprietary manner, they must follow the specification to ensure that portability can be achieved by the developer.

The Client Tier

The Client tier provides a means for a user to interact with the application. This interaction may be through a web browser, or it could also be programmatic through a web services interface. Regardless of the type of client, the interaction includes submitting a request and receiving some type of response from the Middle tier.

In the case of the Struts framework, the most common type of client is a web browser. However, it is also possible to have clients like wireless devices and Java Applets.

The Web Tier

Figure 2-1 shows the Middle tier as an aggregate of the Web tier plus some type of application server component. In the case of Figure 2-1, an EJB container is shown. These two tiers are often combined and many application servers include Web tier functionality.

The Web tier provides the ability for the client tier to communicate and interact with application logic that resides in other tiers. In more traditional web applications, it's not uncommon for some or all of the application logic to reside in this tier. In larger, enterprise-scale applications, the Web tier acts as a translator and maps HTTP requests into service invocations on the Middle tier.

The Web tier is also responsible for managing screen flow based on application and user state. The Web tier communicates with either a database, or in the case of an enterprise application, an application server. The Web Tier is the glue that binds client applications to the core backend business systems.

The components that reside in the Web tier allow developers to extend the basic functionality of a web service. In the case of Struts, it does this by utilizing framework components that run in a servlet container.

The Middle Tier

The Middle tier is often referred as the “application tier” or “server”. This is due in part to the fact that there is often an application server within this tier. Not all Struts applications have an application tier. This is especially true for small web applications. Many small projects choose to forgo using a large application server and communicate directly with a database or some other data store. When an application server is present, the Web tier communicates to it using some variation of RMI. In the case where an EJB server is present in the application tier, the communication protocol is RMI over IIOP.

RMI Over IIOP

Remote Method Invocation (RMI) allows methods to be invoked on remote objects. Java's implementation of RMI is known as Java Method Remote Protocol (JRMP) has been around for quite some time and is specifically designed for Java-to-Java remote communications.

One of the issues with Java's version of RMI is that a JVM must be running on both the client and server for it to be used. With the number of so-called legacy applications that are written in languages such as C++, Java needed a way to communicate with these systems. This is where RMI over IIOP helps out.

Internet Interoperability Protocol (IIOP) was designed to allow distributed components to communicate with one another using TCP/IP. IIOP is language and platform independent.

So, by using RMI on top of IIOP, Java can communicate with applications written in many other different languages and on various platforms. RMI/IIOP as it is often written, is required for all EJB servers to support and exists in the EJB and J2EE specifications.

When included, the application tier might provide a more scalable, fault tolerant, and highly available architecture. This of course, depends on many different factors. One of the main purposes of using an application tier is to separate the responsibilities of presentation from that of the model and the business rules for the application. Today, many web applications are using EJB servers for their application tier. They may not be utilizing all aspects of the J2EE architecture, like EJBs, but there are other benefits that can be leveraged from a J2EE server.

The Enterprise Information System Tier (EIS)

The Enterprise Information System (EIS) tier contains data and services that are used throughout the enterprise. It provides access to enterprise resource such as databases, mainframes, Customer Relationship Management (CRM) applications, and resource planning systems.

The middle tier communicates with components in the EIS tier using protocols that are specific to that resource. For example, to communicate with a relational database, the Middle tier will normally use a JDBC driver. For Enterprise Resource Planning (ERP)

systems, a proprietary adapter is used, although some ERP systems and other enterprise resources are starting to support a more web service-like access approach.

Where does Struts Fit In?

After briefly examining these different tiers, it's important to understand which tier the Struts framework exists in. As illustrated in Figure 2-2, the Struts framework resides in the Web Tier.

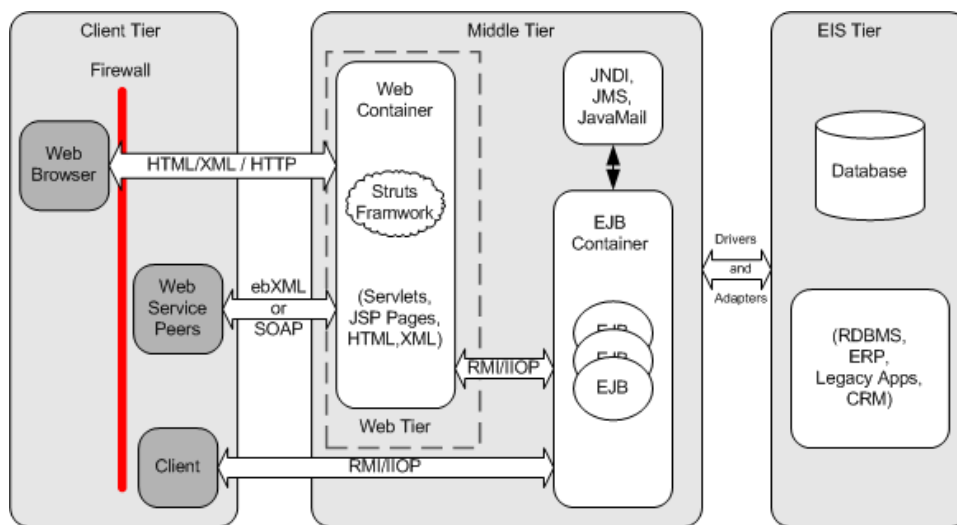


Figure 2-2. The Struts framework is used within the Web Tier

Struts applications are hosted by a web container and can make use of services provided by the container, like handling requests via the HTTP and HTTPS protocol. This frees developers up to focus on building applications that solve a business problem.

The HTTP Request/Response Phase

To better understand how the web server and servlet container work together to service clients, this section discusses the protocol for a HTTP request and response, from the time a client request is received until the server returns a response. This discussion is necessary because Struts makes heavy use of the request and response objects throughout the framework and a thorough understanding of the round-trip process will help to make things more clear for discussions later in the book.

Although the browser is only one possible type of client that can be used with Struts, it is certainly the most common. More and more developers are starting to use Struts for wireless applications and even

some interaction with web services, however the web browser remains the prominent client.

There are two types of HTTP messages, the request and the response. HTTP is based on a request/response model. The browser opens a connection to a server and makes a request. The server processes the client's request and returns a response. Figure 2-3 illustrates this process.

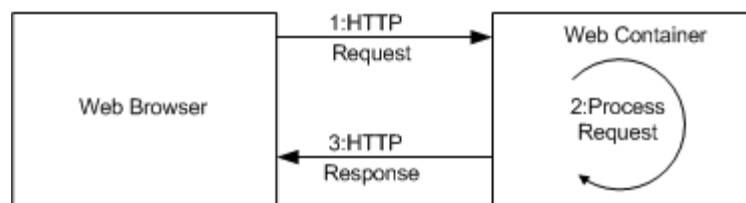


Figure 2-3. The HTTP request/response model

Both types of messages consist of a start line, zero or more header fields, and an empty line, which indicates the end of the message headers. Both message types may also contain an optional message body.

The format and makeup of the request and response messages are very similar, but there are a few differences. We'll discuss each one separately.

The HTTP Request

The start line of an HTTP request is known as the **Request-Line**. It's always the first line of the request message and it contains three separate fields:

- An HTTP method
- A Universal Resource Identifier (URI)
- An HTTP Protocol Version

Although there are several HTTP methods for retrieving data from a server, the two used most often are GET and POST. The GET method requests a specified resource from the server, indicated by the request URI. If the URI is a data producing resource, like a servlet, the data will be returned within the response message. Although the GET message can pass information in the query string, the POST method is used to explicitly pass data to the server that can be used for processing by the request URI.

The Universal Resource Identifier (URI) identifies the resource that should process the request. For the purposes of this discussion, it can either be an absolute path or a relative one. A request with an invalid URI will return an error code, typically a 404.

The HTTP request protocol version is simply an identifier to the server of which version of the HTTP specification the request conforms to. The following example illustrates the request line for a sample GET request.

```
| GET /index.html HTTP/1.0
```

You can actually execute this example by opening up a telnet session to a server running a web server. You must specify the host name and port number of the web server, such as:

```
| telnet localhost 80
```

You can then type the GET command. You will need to press enter twice after issuing the command; once for the end of the request line and then the other return is to let the server know you are finished with the request. Assuming there's a file called *index.html* in the root directory, you will see the HTML response returned. Actually, you will always see a response. It just may not be the one that you expected. We'll talk more about using the telnet application to interact with a web server when we discuss redirects and forwards later in this chapter.

As mentioned, the HTTP request may also contain zero or more header fields. Request header fields allow the client to pass additional information about the request and also the client itself, to the server. The format of a header field, for request and responses, is the name of the header field, following by a colon ":" and the value. If multiple values are specified for a single header field, they must be comma-separated. Table 2-1 shows some of the more commonly used request headers.

Table 2-1. Common HTTP Request header fields

Name	Purpose
Accept	Used to indicate the media types, which are acceptable for the response. If no Accept header field is present, the server can safely assume that the client accepts all media types. An example of an Accept header value is "image/gif, image/jpeg".
Accept-Charset	Used to indicate what character sets are acceptable for the response. If the Accept-Charset header is not present in the request, the server can assume that any character set is acceptable. The ISO-8859-1 character set can be assumed to be acceptable by all user agents.
Accept-Encoding	This header is very similar to the Accept header field, except that it further restricts the content-coding values, which are acceptable by the client. An example of an Accept-Encoding header value is "compress, gzip".
Accept-Language	Used to indicate which languages the client would prefer to have the response in. An example of an Accept-Language header value is "en-us, de-li, es-us".

Content-Encoding	It helps indicate what encoding mechanism has been applied to the body of the message and therefore what decoding must be used to get the information. An example of a Content-Encoding header value is "gzip".
Content-Type	Used to indicate the media type of the body sent to the recipient. An example of a Content-Type header is "text/html; charset=ISO-8859-1".
Host	Used to indicate the host and port number of the resource being requested, as obtained from the original URL. An example of the Host request header is "www.somehost.com".
Referer	The Referer request header field allows the client to specify the address (URI) of the resource from which the request URI was obtained. It is mainly used for maintenance and tracking purposes.
User-Agent	The User-Agent request header field contains information about the client that originated the request. It's mainly used for statistical purposes and tracing of protocol violations. An example of a User-Agent is "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)".

The message body for a request is used to carry data that is associated with the request to the server. The data that is included within the body is different from the values used by the header fields both in terms of format and content. The header fields can be viewed as meta-data about the message body.

The HTTP Response

Once the server has received and processed the request, it must return an HTTP response message back to the client. The response message consists of a status line, zero or more header fields, followed by an empty line. It may also have an optional message body, similar to the request message.

The first line of the HTTP response message is known as the status line. It consists of the HTTP protocol version that the response conforms to, followed by a numeric status code and its textual explanation. Each field is separated by a space. An example response status line is shown here:

```
| HTTP/1.1 200 OK |
```

The status code is a 3 digit numeric value that corresponds to the result code of the server's attempt to satisfy the request. The status code is for programmatic applications, while the reason text is intended for human readers. The first digit of the status code defines the category of the resulting code. Table 2-2 provides the allowed first digits and the corresponding category.

Table 2-2. Status Code Categories

Numeric Value	Meaning
100-199	Informational – Request has been received, continuing to process it.
200-299	Success – The action was successfully received, understood, and accepted.
300-399	Redirection – Further action must be taken in order to complete the request.
400-499	Client Error – The request contains bad syntax or cannot be fulfilled.
500-599	Server Error – The server failed to fulfill an apparently valid request.

There are quite a few status codes that have been defined. They are also extensible, which allows applications to extend the behavior of the server. If a client application doesn't recognize a status code that has been returned by the server, it can assume the type of response status by using the first digit of the returned status code. Table 2-3 lists some of the most common response status codes.

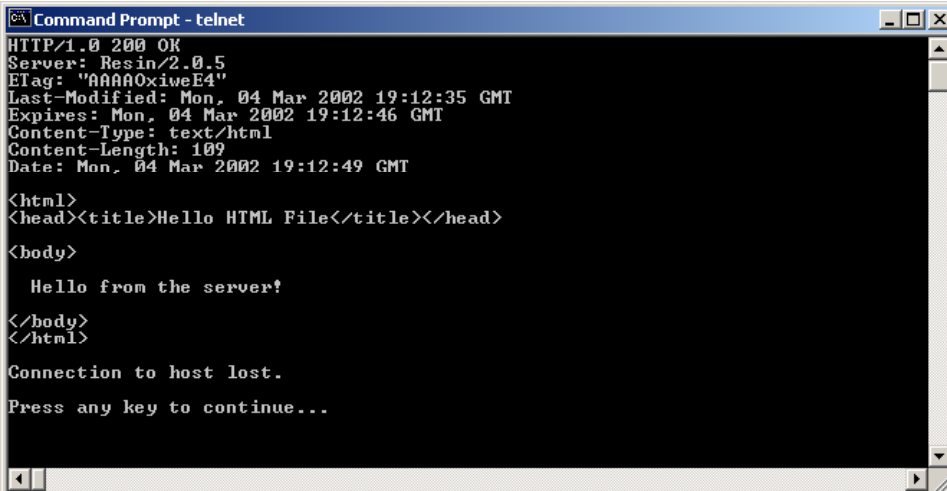
Table 2-3. Common HTTP Response Status Codes

Code	Meaning
200	OK—The request has succeeded.
302	Moved Temporarily —The request resides temporarily under a different URI. If the new URI is a location, the location header field in the response will give the new URL. This is typically used when the client is being redirected.
400	Bad Request—The server couldn't understand the request due to malformed syntax.
401	Unauthorized—The request requires authentication and/or authorization.
403	Forbidden—The server understood the request, but for some reason is refusing to fulfill it. The server may or may not reveal why it has refused the request.
404	Not Found—The server has not found anything matching the request URI.
500	Internal Server Error—The server encountered an unexpected condition which prevented it from fulfilling the request.

The header fields in the response are similar in format to those found in the request message. They allow the server to pass additional information to the client, which cannot be placed in the status line. These fields give information about the server and about further access to the URI contained within the request. After the last response header,

which is followed by an empty line, the server can insert the response message body. In many cases, the response message body is HTML output. Figure 2-4 illustrates an example response:

```
| GET /hello.html HTTP/1.0
```



```
HTTP/1.0 200 OK
Server: Resin/2.0.5
ETag: "AAAA0xiweE4"
Last-Modified: Mon, 04 Mar 2002 19:12:35 GMT
Expires: Mon, 04 Mar 2002 19:12:46 GMT
Content-Type: text/html
Content-Length: 109
Date: Mon, 04 Mar 2002 19:12:49 GMT

<html>
<head><title>Hello HTML File</title></head>
<body>
  Hello from the server!
</body>
</html>

Connection to host lost.
Press any key to continue...
```

Figure 2-4. An example HTTP response message

HTTP versus HTTPS

You've obviously noticed that the request and response message text shown in the previous examples have all been standard readable text. This is fine for some cases, however it also means that computer hackers can as well. When you need to ensure the integrity and privacy of information that is sent over a network, especially an open one like the Internet, one of the options is to use the HTTPS protocol, rather than standard HTTP.

HTTPS is normal HTTP wrapped by a Secure Sockets Layer (SSL). SSL is a communication system that ensures privacy when communicating with other SSL-enabled applications. It's really just a protocol that runs on top of the TCP/IP layer. It encrypts the data through the use of symmetric encryption and digital certificates. An SSL connection can only be established between a client and server when both systems are running in SSL mode and are able to authenticate each other.

The fact that the SSL layer encrypts the transmitted data has no impact on the underlying request and response message. The encryption and subsequent decryption on the other side occurs after the message is constructed and is decoupled from the HTTP portion of the message.

Chapter 14 deals with security in a Struts application. HTTPS and SSL will be covered more thoroughly there.

Struts and Scope

The Struts framework utilizes various shared resource areas to store objects. The shared resource areas all have a lifetime and visibility rule that defines the *scope* of the resource. This section discusses these resources, their scopes, and how the framework utilizes them.

Request Scope

Each time a client issues an HTTP request, the server creates an object that implements the `javax.servlet.http.HttpServletRequest` interface. Among other things, this object contains a collection of key/value attribute pairs that can be used to store objects for the lifetime of the request. The key of each pair is a `String` and the value can be any type of `Object`. The methods to store and retrieve objects into and out of the Request scope are:

```
public void setAttribute( String name, Object obj );  
public Object getAttribute( String name );
```

Request-scope attributes can be removed using the `removeAttribute()` method, however, because the scope of the attribute is only for the lifetime of the request, it is not as important to remove them as it is for other request-scope attributes. Once the server fulfills a request and a response is returned to the client, the request, and therefore the attributes are no longer available to the client and may be garbage collected by the JVM.

The Struts framework provides the ability to store JavaBeans into the request, so that they can be used by presentation components like JSP pages. This makes it much easier to get access to JavaBeans data, without having to do manual cleanup of removing the objects later. The web container will take care of it for you. There's seldom a need to remove objects from request scope. As you can see, the visibility of objects stored at the request level is only for resources that have access to that request. Once the response has been returned to the client, the visibility is gone. Objects that are stored in one request are not visible to any other client request.

Session Scope

The next higher level of visibility is session scope. The web container will create an object that implements the `javax.servlet.http.HttpSession` interface to identify a user across multiple page requests. The user's session will persist for a period of time that is based on how frequent the user makes a request. This allowed inactivity time is configurable through the application deployment descriptor. It may also be prematurely destroyed by calling the `invalidate()` method on the session object.

The session also allows for a collection of objects to be stored based on a key/value pair schema, similar to the request object. The only difference between this one and the one

provided by the request is the duration of the objects. Since sessions exist across multiple requests, objects stored in the session scope live longer than those at the request level. The Struts framework uses session attributes quite extensively. An example of an object that may be stored as a session attribute is the `Locale` object for the user. This allows the entire framework access to the user's locale to perform localized behavior. Objects stored in one user's session are not visible to users with a different session.

There is no synchronization provided by container for the session object. If multiple threads attempt to access an object stored in the session and modify the value, it's up to the developer to provide synchronization. There is no automatic synchronization provided by the session. Although the need to synchronize access to the session is quite low, you must realize that the developer has the responsibility to protect the resources. Scenarios where multiple threads might access the session at the same are if your application uses frames, or if you have a process takes a long time to complete. Both of these situations could cause more than one thread to attempt to access the same session attribute.

Application Scope

An even higher level of visibility and duration comes with objects stored at the application scope level. These are objects that are visible to all client and threads of the current web application. They live until they are programmatically removed or until the application is terminated.

The server creates an object that implements the `javax.servlet.ServletContext` interface for each and every web application that is installed within the container. The `ServletContext` object allows objects to be stored and retrieved exactly like they are done for the request and session scopes, with the one difference being that the objects are visible to the entire application and persist for the lifetime of the application. The Struts framework uses application scope to store JavaBeans that need to be visible to all users. Normally, objects are stored in this scope during application startup and remain there until the application exits.

Page Scope

The last scope to discuss has to do exclusively with JSP pages and is referred to as page scope. Objects with page scope are only accessible within the JSP page that they are created. Once the response is sent to the client or the page forwards to another resource, the objects are no longer available. The objects are stored in the `javax.servlet.jsp.PageContext` object for each page.

Every JSP page has an implicit object reference named `pageContext` that can be used to store and retrieve page-level objects. It includes the same `getAttribute()` and `setAttribute()` methods that the other scopes offer and function in the same manner.

Using URL Parameters

Request* parameters are strings that are sent with the client request to the server. The parameters are inserted into the `HttpServletRequest` object from the URI query string and data that is sent in a POST method. The parameters are formatted as key/value pairs.

URL parameters are only available via the `getParameter()` method when a POST method is used. If the HTTP method is a GET, the only way the server can gain access to the parameters included in the query string is by parsing the String returned from the `getPathInfo()` or `getRequestURI()` methods.

URL parameters play an important role in all web applications and the Struts framework is no exception.

Forward versus Redirect

It's often necessary to share control of a request by more than a single component. One servlet may be responsible for authenticating and authorizing a client, while it's the job of a different servlet to retrieve some data for the user. This sharing of control of a request is accomplished in several different ways. Two of which, we want to discuss in this section.

There are several important differences between how a web container processes a forward request versus a redirect. Since the Struts *front controller* servlet, which was discussed in Chapter 1, will always perform one or the other for a typical request, it's important that you understand these differences and the impact that each mechanism will have on your application.

How a Redirect Works

When a `sendRedirect()` method is invoked, it causes the web container to return a response back to the browser with information that a new URL should be requested. Because the browser issues a completely new request, any objects that are stored as

* URL parameters are often referred to as query or request parameters.

request attributes before the redirect occurs, will be lost. This is one of the biggest differences between a forward and redirect. Figure 2-5 illustrates why this occurs.

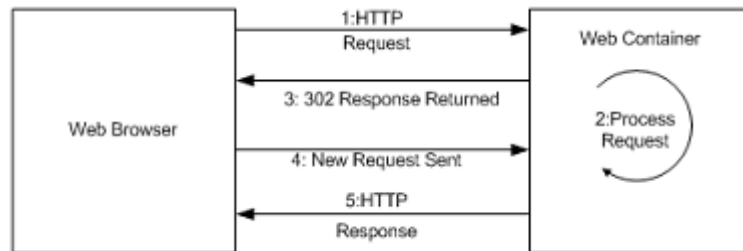


Figure 2-5. A redirect causes the browser to issue a new request

Because of the extra round-trip that occurs, a redirect is slower than using a forward. Example 2-1 provides an example servlet that performs a redirect for a JSP page called *result.jsp*, when a request is issued for the servlet.

Example 2-1. A Java servlet that performs a redirect when it receives a request

```
package com.oreilly.struts.chapter2examples;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;

/**
 * Perform a redirect for the page "redirect.jsp"
 */
public class RedirectServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        redirect( request, response );
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        redirect( request, response );
    }

    /**
     * Set a few URL parameters and objects for the request to see what happens
     * to them during a redirect.
     */
    protected void redirect(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException{
        log( "A request arrived for " + req.getServletPath() );

        // Put some objects into request scope
```

```
req.setAttribute( "firstName", "John" );
req.setAttribute( "lastName", "Doe" );

String contextPath = req.getContextPath();
String redirectStr = contextPath + "/result.jsp?username=foo&password=bar";

log( "redirecting to " + redirectStr );

// Always call the encodeRedirectURL method when performing a redirect
resp.sendRedirect( resp.encodeRedirectURL(redirectStr) );
}
```

When the servlet in example 2-1 receives either a GET or POST request, it calls the `redirect()` method and passes the `HttpServletRequest` and `HttpServletResponse` objects to it. The method sets two `String` objects into the request so that it can be demonstrated that they will not be available after the redirect. It creates a `String` that will become the URL that the client is told to make a new request for. After the `encodeRedirectURL()` method is called and passed the redirect string, the `sendRedirect()` method is invoked on the response object.

All URLs passed to the `sendRedirect()` method should be ran through the `encodeRedirectURL()` method so that the Session ID can be included if the browser doesn't support cookies and session tracking needs to occur. The Struts framework performs this step automatically in the `RequestProcessor` during normally action processing.

The JSP page that the servlet will redirect to is shown in example 2-2.

Example 2-2. A simple JSP page that a client is redirected to when calling the `RedirectServlet` from example 2-1

```
<html>
<head>
  <title>Struts Redirect/Forward Example</title>
</head>

<body>

<br>
<%
String firstName = (String)request.getAttribute( "firstName" );
if ( firstName == null ){
  firstName = "Not found in request";
}

String lastName = (String)request.getAttribute( "lastName" );
if ( lastName == null ){
  lastName = "Not found in request";
}
```

```
%>

<b>First Name:</b> <%=firstName%><br>
<b>Last Name:</b> <%=lastName%><br>

</body>
</html>
```

When you enter the URL for this servlet in a browser:

<http://localhost:8080/servlet/com.oreilly.struts.chapter2examples.RedirectServlet>

The browser output will look like the one in Figure 2-6.

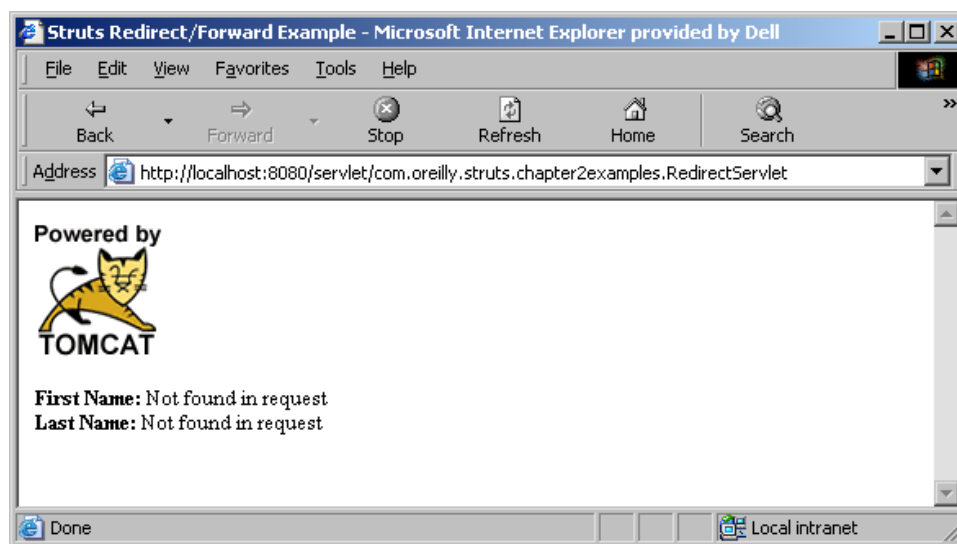


Figure 2-6. The output page when the RedirectServlet is called

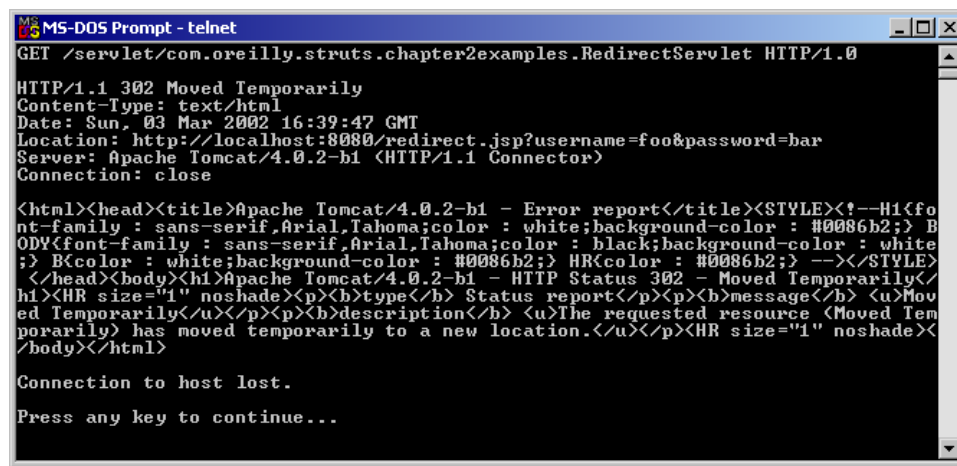
You'll notice that the first name and last name arguments that were set in the servlet were not found in the request. This is because a second request was actually issued for this page. Fortunately, we can peek behind the scenes and observe what's taking place.

Let's look at the HTTP response that comes back to the client when the **RedirectServlet** is requested. We can see this using a standard telnet session. Since an HTTP connection uses a simple network socket to communicate with the server, we can partially simulate the interaction between a browser and a server using the telnet application. You can establish a connection to a web server using telnet by connecting to the port that the server is listening on, usually port 80 or in the case of Tomcat 8080.

```
| telnet localhost 8080
```

You can do this from the command line, whether you're using a DOS shell or Unix. The telnet session will alert you if it's unable to connect to a server with the hostname and port.

If a connection is established, the telnet session will sit and wait for you to enter the HTTP request information. The only information that you are required to enter is the first line of the request, which tells the server what resource the client wants. Each line in the request is a text line, separated by a newline character. The request header ends with a blank line, so you actually have to press “enter” twice to let telnet know you are done with the request. Figure 2-7 shows the HTTP response message returned from Tomcat when a request is made for the `RedirectServlet` resource.



```
MS-DOS Prompt - telnet
GET /servlet/com.oreilly.struts.chapter2examples.RedirectServlet HTTP/1.0

HTTP/1.1 302 Moved Temporarily
Content-Type: text/html
Date: Sun, 03 Mar 2002 16:39:47 GMT
Location: http://localhost:8080/redirect.jsp?username=foo&password=bar
Server: Apache Tomcat/4.0.2-b1 <HTTP/1.1 Connector>
Connection: close

<html><head><title>Apache Tomcat/4.0.2-b1 - Error report</title><STYLE><!--H1{font-family : sans-serif,Arial,Tahoma;color : white;background-color : #0086b2;} BODY{font-family : sans-serif,Arial,Tahoma;color : black;background-color : white;} B{color : white;background-color : #0086b2;} HR{color : #0086b2;} --></STYLE></head><body><h1>Apache Tomcat/4.0.2-b1 - HTTP Status 302 - Moved Temporarily</h1><HR size="1" noshade><p><b>type</b> Status report</p><p><b>message</b> <u>Moved Temporarily</u></p><p><b>description</b> <u>The requested resource <u>Moved Temporarily</u> has moved temporarily to a new location.</u></p><HR size="1" noshade></body></html>

Connection to host lost.
Press any key to continue...
```

Figure 2-7. You can use a telnet session to inspect the HTTP response headers

In Figure 2-7, the first line of the request issues a GET and a path for a resource. The path is the portion of the HTTP request that comes after the host name and includes the preceding “/” character. The “HTTP/1.0” string at the end of the GET request is the HTTP version protocol.

HTTP version 1.1 added a number of optimizations over 1.0. However, there are additional request headers that must be included, which would make this example more complicated than necessary. Therefore, 1.0 was used to issue the request.

Everything after the first line is a response from the server. The entire HTML output at the bottom comes from Tomcat. It informs anyone that is interested that the original request has performed a redirect and the client should request a new URL.

If you look at the Location response header, which is five lines down in Figure 2-7, you can see that the server has informed the client what the URL should be for the new request. Any URL parameters that were attached to the original request will still be present in the new request. You can see this by looking at the value of the `Location` response header in Figure 2.7. When the browser issues the new request, these parameters will be sent.

The technique of using telnet provides a simple way of interacting with a web server and viewing what responses it sends back to the client.

Hopefully, you now have a better understanding of how a redirect works. It should also be clearer why any objects placed into the original request are not available to the redirected resource. This will become very important later when we discuss the Struts framework.

One interesting thing may not catch your attention here. Notice back in example 2-1, the redirect string didn't explicitly contain the hostname and port that the client needed to use for the new request. It's the job of the servlet container to translate the relative URL to a fully qualified URL for transmission back to the client. Later in the Chapter 20, you'll see a way of overriding this behavior. This may be necessary if you're using a proxy server and masquerading the server on another host or port.

Using a Forward

A forward differs from a redirect in several distinct ways. For one, when you invoke a `forward` for a request, the request is sent to another resource on the server, without informing the client that a different resource is going to process the request. This process occurs completely within the web container and the client is never the wiser. Unlike a `redirect`, objects can be stored into the request and passed along for the next resource to make use of. Figure 2-8 illustrates the steps that take place when a request is forwarded.

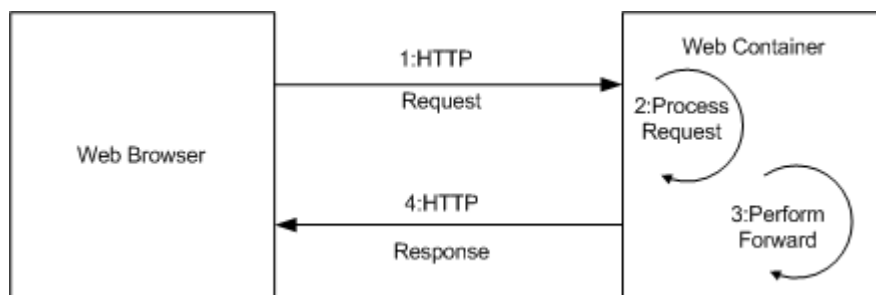


Figure 2-8. During a forward, the request is sent to a secondary resource on the server without notifying the client

Since a forward takes place completely on the server and there is no communication with the client, the performance is better than a redirect. However, there are some differences in how a forward deals with relative URLs. Example 2-3 should make this clearer.

```
Example 2-3. A Java servlet that performs a forward when it receives a request
package com.oreilly.struts.chapter2examples;
```

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;
/**
 * Perform a forward to the page "redirect.jsp"
 */
public class ForwardServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        forward( request, response );
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        forward( request, response );
    }
    /**
     * Set a few URL parameters and objects for the request to see what happens
     * to them during a redirect.
     */
    protected void forward(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException{
        log( "A request arrived for " + req.getServletPath() );

        // Put some objects into request scope
        req.setAttribute( "firstName", "John" );
        req.setAttribute( "lastName", "Doe" );

        String redirectStr = "/result.jsp?username=foo&password=bar";

        this.log( "forwarding to " + redirectStr );
        RequestDispatcher dispatcher = req.getRequestDispatcher( redirectStr );
        dispatcher.forward( req, resp );
    }
}
```

When the servlet in example 2-3 receives either a GET or POST request, it calls the `forward()` method and passes the `HttpServletRequest` and `HttpServletResponse` objects to it. As with the redirect example, two `String` objects are set into the request. However, in contrast with the redirect example, the objects will be available after the forward. It next creates a redirect path that the request will be passed to. A `RequestDispatcher` is created and the `forward()` method is invoked on it. We'll use the same JSP page from example 2-2. In the browser, we enter a URL like:

<http://localhost:8080/servlet/com.oreilly.struts.chapter2examples.ForwardServlet>

and the browser output should look similar to Figure 2-9.

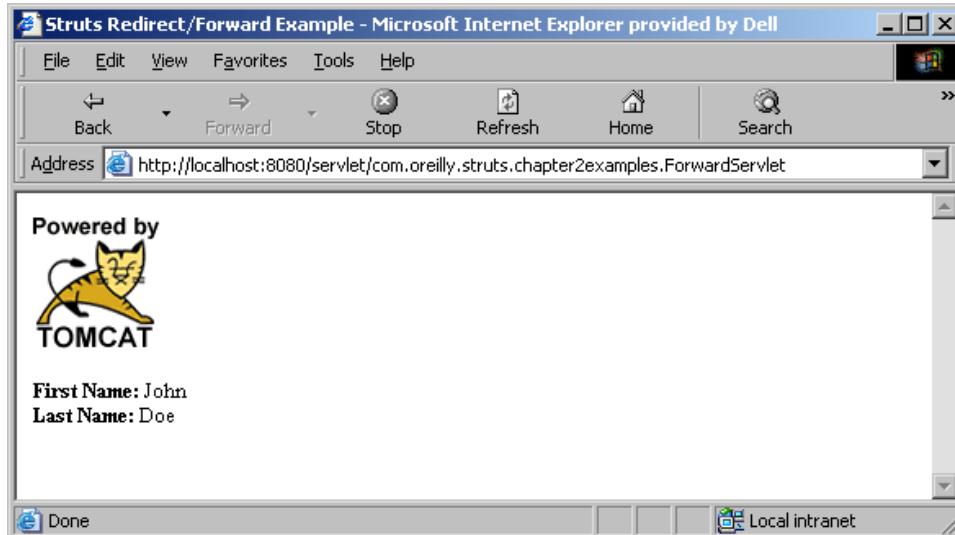


Figure 2-9. The output page when the ForwardServlet is called

There are two interesting items that can be seen from Figure 2-9. The first is that the first and last name fields have values. This is because the objects were placed into the request before the forward occurred and the *result.jsp* was able to retrieve the values and use them in the page.

The second interesting item is the URL in the address bar. Notice that the URL back in Figure 2-6 showed the new URL that the client requested, whereas it didn't change in Figure 2-9. This shows further that the client is not aware that a forward occurred. It didn't get a notice that the browser URL should change.

The forward method of the `RequestDispatcher` class can only be called when output hasn't already been committed to the client. Writing something to the response object and then calling forward will result in an `IllegalStateException` being thrown by the container.

Which one for Struts?

It's difficult to decree whether you should use a redirect or a forward for Struts applications, since we haven't discussed the framework yet. However, there are some key points about using each approach. Both mechanisms have their pros and cons within Struts and web applications in general. For many situations, a forward is recommended over a redirect. This is chiefly due to the fact that when a forward is used, objects that are stored in the request scope, are easily available to the presentation components. In fact, using a forward is the default case for the Struts framework. Another advantage of a forward is that it's much more efficient, since the client is not required to issue a new request.

There are however, some situations where a redirect is necessary or preferred over using a forward. The problem revolves around the fact that when forwards are used, the URL's are not always keep consistent with the state of the client application. If the browser's refresh button is pressed, unexpected results can occur. This may not make too much sense right now, but it will be explored further in Chapter 5.

[\[Editors: I wanted to mention something in here about HttpSession management, but I wasn't sure if it belonged in here or say in the design chapter. Any feedback on this question would be helpful.\]](#)

3

Overview of the Struts Framework

[\[Note to all editors: After my initial reading of this chapter, I realized that I need to emphasize the new 1.1 features a little more. I will make those additions during author review\]](#)

It's finally time to introduce the Struts framework. Familiarity with the material from the previous two chapters will allow you to absorb the information here much faster. This chapter will provide an overview of the Struts framework and not attempt to uncover all of the features or in any significant depth. It will emphasize instead, how all of the pieces fit into the MVC and Model 2 architecture presented in Chapter 1.

The rest of the book will be spent pulling back the layers and uncovering the details of the framework, expanding on the more basic concepts and terminology introduced here. It is important that you have a firm grasp of the fundamentals presented in this chapter. Even if you are familiar with the basic concepts of the Struts framework, it's recommended that you take the time and immerse yourself in this chapter first.

If you are one of the many developers trying to convince your manager to switch to the Struts framework, this should be an excellent chapter for them to read. Since the discussion will be kept at an overview level, this chapter will not be filled with too many boring technical details. Hopefully, it will provide a valuable overview for both them and you.

A Banking Account Example

This section introduces an online banking application that we'll be using to familiarize you to the world of Struts. It represents the proverbial "Hello World" application that you

find in many first examples. A banking will be used to make the discussion more interesting and a little more applicable to the types of applications that Struts can be used for. The example presented here will not be complete and will be kept purposely trivial, but will nonetheless help provide a high-level overview of the major components that are present in all Struts applications and how those components fit together. A more comprehensive and thorough example based on a shopping cart paradigm will be used throughout the rest of the book, but the banking example will be a good one to start with. Almost everyone should be familiar with the concept of online banking and we can avoid spending too much time on explaining the business requirements.

Our online banking application will allow an end-consumer to login to the financial institution's web site and view account information and transfer funds from one account to another, assuming the user has more than one account of course. The user must present a valid set of credentials to enter the site, which for this example will consist of both an *Access Number* and a *Pin Number*.

If the user leaves one or both fields blank, the application will display a formatted message informing the user that both fields are required. If the user enters values for both fields, but the authentication fails, the login screen will be re-displayed along with a formatted error message informing the user that the login has failed. Figure 3-1 shows the online banking login screen after an invalid login attempt has been detected.

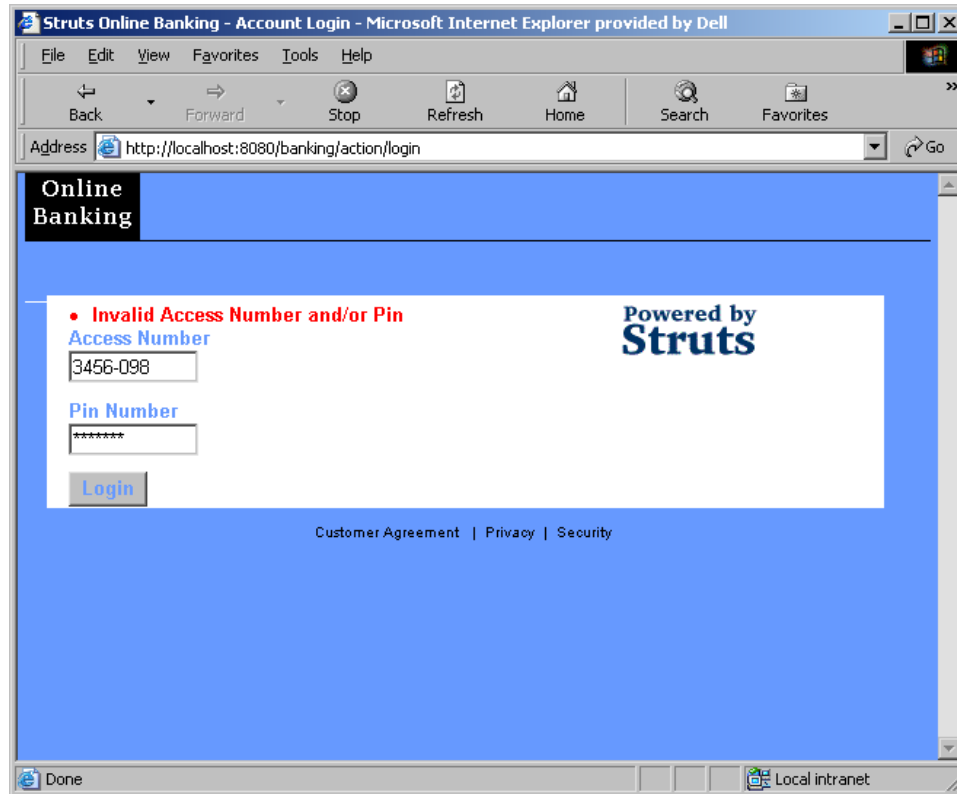


Figure 3-1. The Login screen for the online banking application

If the proper credentials are entered for an account, the user is taken to the account information screen. This screen will show all of the accounts that the user has with the financial institution, as well as the current balance for each account.

For this example, we are not going to provide a robust, full-fledge security service and security realm. In fact, no time at all will be spent on developing a real security service. This means that authentication and authorization will be hard-coded for simplicity. Handling security in a web application can get very complicated and there's no reason to muddy the waters with it in this chapter. Chapter 14 will cover security in depth and it's best to wait until then to discuss security. For the purposes of this chapter, a simple Java interface will be used that contains a single `login()` method. This interface, or rather an object that implements this interface, will be used to authenticate users. The authentication interface is shown in Example 3-1.

Example 3-1. The `IAuthentication` Interface used by the banking application

```
package com.oreilly.struts.banking.service;  
  
import com.oreilly.struts.banking.view.UserView;
```

```
/**
 * Provides methods that the banking security service should implement.
 */
public interface IAuthentication {
    /**
     * The login method is called when a user wishes to login to
     * the online banking application.
     * @param accessNumber The account access number.
     * @param pinNumber The account private id number.
     * @returns A ValueHolder object representing the user's personal data.
     * @throws InvalidLoginException if the credentials are invalid.
     */
    public UserView login( String accessNumber, String pinNumber )
        throws InvalidLoginException;
}
```

Since Java interfaces can't be instantiated, we need to have a class that implements this interface.

The `com.oreilly.struts.banking.service.SecurityService` class is shown in Example 3-2. It implements the `IAuthentication` interface and provides the application the ability to authenticate users. Of course, since we are not really going to authenticate against a security realm, the `SecurityService` class allows anyone to login to the application.

Example 3-2. The SecurityService used by the banking application for authentication

```
package com.oreilly.struts.banking.service;

import com.oreilly.struts.banking.view.UserView;
/**
 * Used by the example banking application to simulate a security service.
 */
public class SecurityService implements IAuthentication {

    /**
     * The login method is called when a user wishes to login to
     * the online banking application.
     * @param accessNumber The account access number.
     * @param pinNumber The account private id number.
     * @returns A ValueHolder object representing the user's personal data.
     * @throws InvalidLoginException if the credentials are invalid.
     */
    public UserView login(String accessNbr, String pinNbr)
        throws InvalidLoginException {

        // A real security service would check the login against a security realm
        // This example is hard coded to only let in 123/456
        if(
            (accessNbr != null && accessNbr.equalsIgnoreCase("123")) &&
            (pinNbr != null && pinNbr.equalsIgnoreCase("456")) ){
```

```
        /* Dummy a UserView for this example.
         * This data/object would typically come from the business layer
         * after proper authentication/authorization had been done.
         */
        UserView newView = new UserView( "John", "Doe" );

        // Give the user a primary key, which will be used later. Of course
        // this would come from the data store in a real application.
        newView.setId( "39017" );

        return newView;
    }
    else {
        // If the user enters anything other than 123/456, throw this exception
        throw new InvalidLoginException();
    }
}
```

Remember, for this example application, we are only going to authorize the user if the *accessNumber* entered is “123” and a *pinNumber* of “456”. If the `login()` method is successful, a `com.oreilly.struts.banking.view.UserView` object is created and returned. If an invalid set of credentials is passed to this method, a `com.oreilly.struts.banking.service.InvalidLoginException` is thrown. The `LoginAction` is comparing the values entered by the user against the hard-coded values. Obviously, in a real application, you would check against some type of a security realm, such as a relational database or an LDAP server.

Once the user has logged in successfully, they may perform one of three different actions:

- View an account detail
- Transfer funds from one account to another (if the user has two or more accounts)
- Logout

Obviously, in a real online banking application, there would be much more functionality included. However, for the purposes of the overview, this limited feature set will suffice. Figure 3-2 shows the account information screen, which the user is taken to after a successful login.

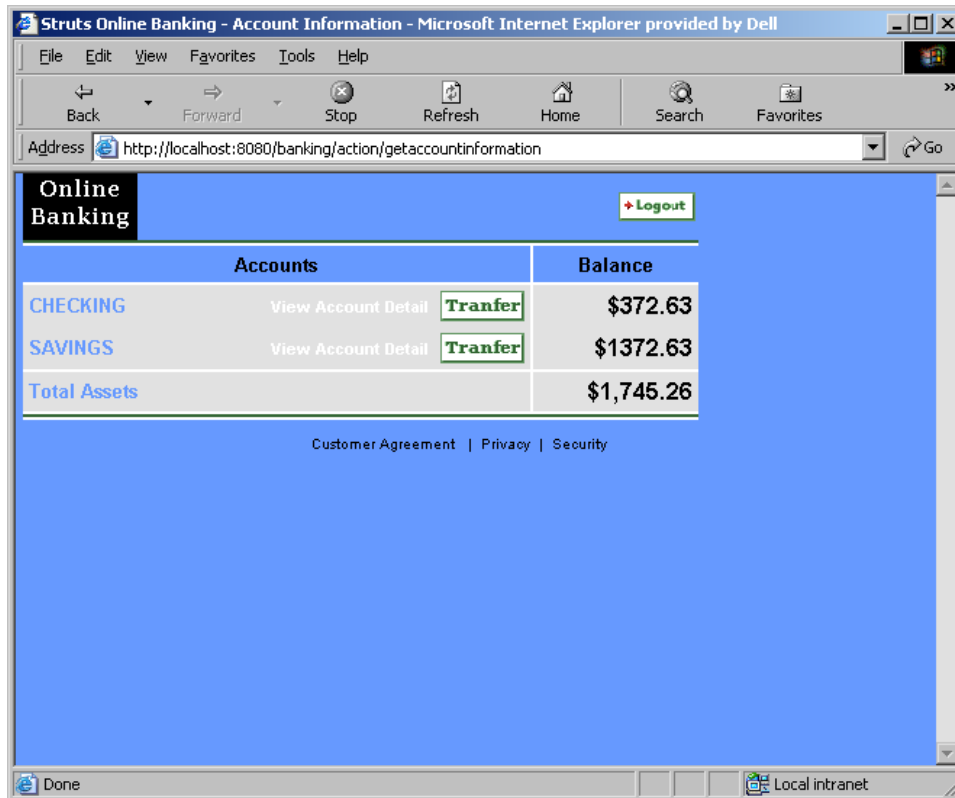


Figure 3-2. After the user successfully enters the banking application, they are taken to the Account Information screen

The user can view detailed information about an account by clicking on the *View Account Detail* link. Figure 3-3 shows the Account Detail Screen for one of the accounts listed in Figure 3-2.

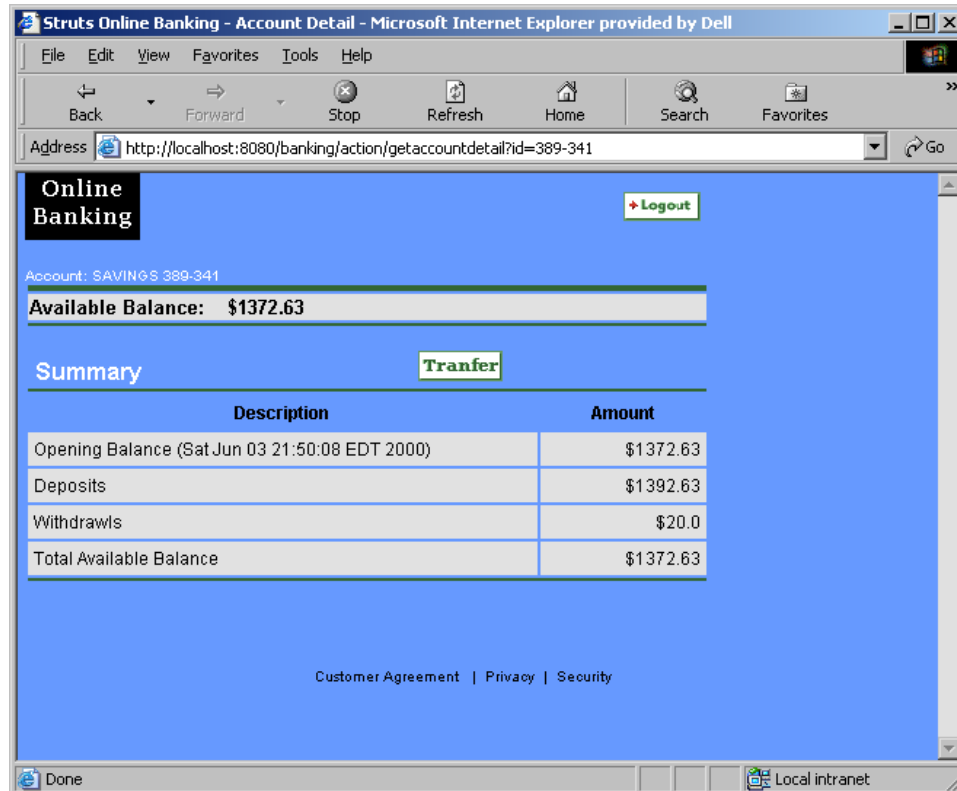


Figure 3-3. The Account Detail screen for the online banking application

They may also transfer funds from one account to another by clicking on the *Transfer Funds* link next to the account they wish to transfer funds from. Figure 3-3 shows the *Account Detail* screen. Because the purpose of this chapter is to familiarize you with the components of the Struts framework and not teach you the correct functionality of a web banking application, the funds transfer functionality will not actually be implemented. This will be a nice feature for you to implement later on as a practical exercise if you desire.

Finally, the user may simply logout of the application altogether by clicking on the logout function. The user will be logged out of the application and returned to the login screen.

Looking at the Big Picture

Now that we've shown an example that we can use as the basis of this Chapter's discussion, it's time to start looking at how we can solve it using the Struts framework. Although Chapter 1 discussed the MVC pattern in the order of Model-View-Controller, it doesn't necessarily make sense to follow that same order as we explore the Struts

components. In fact, it's more logical to cover the components in the same order that the Struts framework uses them to process a request. For that reason, the components that make up the controller portion of the framework will be discussed first.

The Struts Component Packages

The Struts framework is made up of approximately 200 Java classes, divided into 15 Java packages. Approximately is an appropriate term because the framework is continuously growing and being shaped. Appendix A provides a complete listing of the Struts API for your convenience. For this chapter, we'll only focus on the top-level packages. Figure 3-4 shows the top-level packages for the Struts framework and their dependencies.

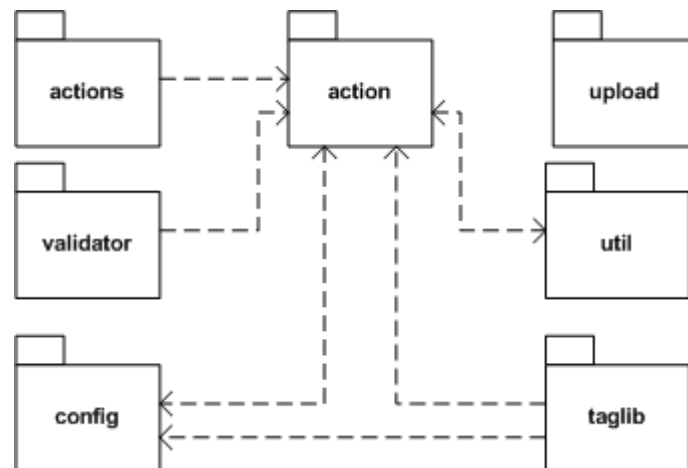


Figure 3-4. The Struts framework consists of 7 top-level packages

The framework components are not arranged by what role they play in the MVC pattern. They are actually arranged a little haphazardly. This has to do more with how fast the framework has evolved, rather than poor decisions made by the designers. The `action` package for example, contains classes for the controller, some that are used by the view domain, and even a few that probably would have been better off in the `util` package. With that being said, it's not that hard to get use to where everything is and after awhile, it even gets comfortable. In many cases, you normally only have to import the `action` package anyway. A Few of these top-level packages also contain sub-packages. These sub-packages will be presented as we move through the book.

Struts Controller Components

As you saw in Chapter 1, the controller component in a MVC application has several responsibilities. Those responsibilities include receiving input from a client, invoking a business operation, and coordinating the view to return back to the client. Of course,

there are many other functions that the controller may perform, but these are a few of the primary ones.

You also learned that, with the JSP Model 2 architecture, on which Struts was fashioned the controller was implemented by a Java servlet. This servlet becomes the centralized point of control for the web application. The controller servlet maps user actions into business operations and then helps to select the view to return to the client based on the request and other state information. Figure 3-5 shows the figure from Chapter 1 as a reminder.

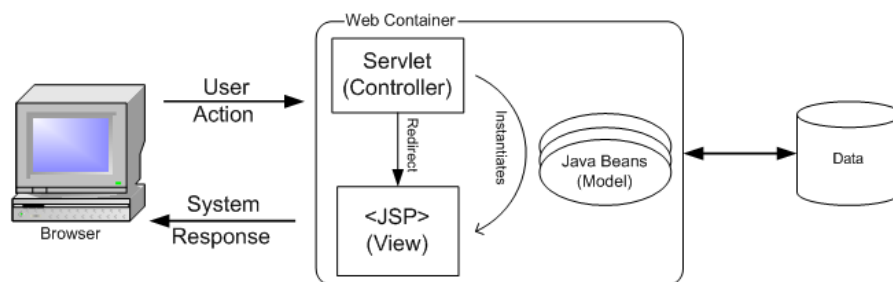


Figure 3-5. The Struts framework uses a Java servlet to handle the controller responsibilities

In the Struts framework however, the controller responsibilities are implemented by several different components, one of which is an instance of the `org.apache.struts.action.ActionServlet` class.

The Struts ActionServlet

The `ActionServlet` extends the `javax.servlet.http.HttpServlet` class and is responsible for packaging and routing HTTP traffic to the appropriate handler in the framework. The `ActionServlet` class is not abstract and therefore can be used as a concrete controller by your applications. Prior to version 1.1 of the Struts framework, the `ActionServlet` was solely responsible for receiving the request and processing it by calling the appropriate handler. In version 1.1, a new class called `org.apache.struts.action.RequestProcessor` has been introduced to process the request for the controller. The main reason for decoupling the request processing from the `ActionServlet` is to provide you with the flexibility to subclass the `RequestProcessor` with your own version and modify how the request is processed. For the banking application example, we are going to keep it simple and utilize the default `ActionServlet` and `RequestProcessor` classes provided by the framework. Chapter 5 will describe in detail how these classes can be extended to modify the default controller behavior. For brevity in this chapter, we will refer to these

two components simply as the “controller”. In Chapter 5, the role and responsibilities of each component will be explained.

Like any other Java servlet, the Struts `ActionServlet` must be configured in the deployment descriptor for the web application. We won't go into detail about the deployment descriptor here, that's saved for Chapter 4 “Configuring the Struts Application”.

Once the controller receives a client request, it delegates the handling of the request to a helper class. This helper knows how to execute the business operation that is associated with the requested action. In the Struts framework, this helper class is a descendant of the

Is the Action part of the Controller or Model?

Throughout the various articles, tutorials, and other resources that are available on the Struts framework, developers have a differing opinion on whether the `Action` class is part of the controller or the model. The argument for it being part of the controller is that it isn't part of the “real” business logic. If Struts were replaced with an alternative framework, chances are the `Action` class would be replaced with something else. Therefore, it really isn't part of the model domain, but rather tightly coupled to the Struts controller. It doesn't make sense to put business logic into the `Action`, since other types of clients can't easily reuse it.

Another reason to consider the Struts `Action` class part of the controller is that it has access to the `ActionServlet` and therefore all of the controller resources, which the domain model shouldn't know about. Hypothetically, the `Action` class behavior could have been left in the servlet and the servlet would just call the appropriate method on itself. If this were the case, there would be no doubt about whether this was controller or model functionality.

With all of this said, the `Action` class in many cases, does invoke an operation on the business model and many developers end up trying to insert too much of their business logic into the action classes. Eventually, the line becomes very blurry. Perhaps this is why some developers consider it part of the model. This book will take the approach that the `Action` class is part of the controller.

`org.apache.struts.action.Action` class.

Struts Action Classes

An `org.apache.struts.action.Action` class in the Struts framework is an extension of the controller component. It acts as an *Adaptor** between a user action and a business operation. The `Action` class decouples the client request from the business model. This decoupling allows for more than a one-to-one mapping between the user request and an `Action` class. The `Action` class can perform other functions, such as authorization, logging, and session validation, before invoking the business operation.

The Struts `Action` class contains several methods, but the most important is the `execute()` method. Here is the method signature:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws IOException, ServletException;
```

The `execute()` method is called on an instance of an `Action` class by the controller when a request is received from a client. The controller will create an instance of the `Action` class if one doesn't already exist. The Struts framework will only create a single instance of each `Action` class in your application. Since there is only one instance for all users, you must ensure that all of your `Action` classes operate properly in a multi-threaded environment. Figure 3-6 illustrates how the `execute()` method is invoked by the primary controller components.

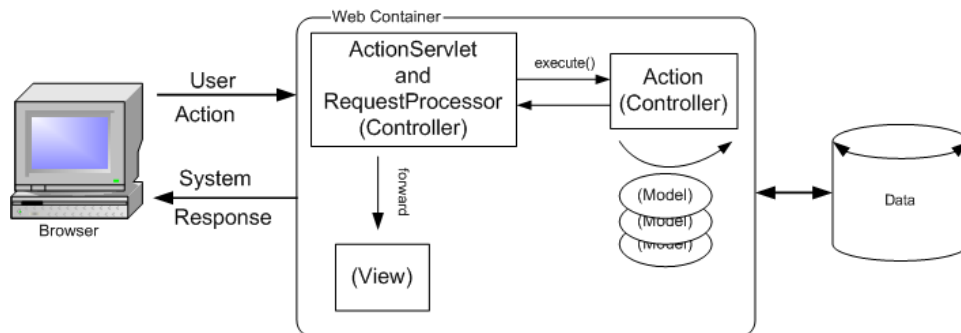


Figure 3-6. The `execute()` method of the `Action` class is called by the primary controller

Although the `execute()` method is not abstract, the default implementation just returns null so you will need to create your own `Action` class implementations and override this method.

* The Adaptor pattern is a structural design pattern mentioned in the book “Design Patterns” written by Gamma, Helm, Johnson, and Vlissides, affectionately known as the *GangOfFour (GoF)*.

There is some debate over how best to implement `Action` classes using Struts. Whether you create a different `Action` class for each and every operation or whether you put several business operations in the same `Action` class is subjective and has pros and cons on both sides.

In chapter 5, we'll discuss an action provided by the Struts framework called `org.apache.struts.actions.DispatchAction`. This `Action` gives you the ability to create a single `Action` class and implement several similar operations like *Create, Read, Update, and Delete* (CRUD) within it. This has the effect of creating a smaller number of `Action` classes, but might make maintenance a little harder. Chapter 21 will discuss some practices for designing your `Action` classes.

As was mentioned previously, there are various ways to organize your `Action` classes. For the banking application, we will create a unique `Action` class for each action that the user can perform. Therefore, we will need to create the following actions:

- `Login`
- `Logout`
- `GetAccountInformation`
- `GetAccountDetail`

Each one of the action classes that we'll create for the banking application will extend the Struts `Action` class and override the `execute` method to carry out the specific operation. In chapter 5, you'll learn that it's best to create an abstract base `Action` class for your application that all of your other action classes extend.

This application specific base action would extend the Struts `Action` class and provide you with added flexibility and extensibility that's hard to see the need for at the beginning stages of development, but later you'll be glad that you did. For the banking application however, things will be kept simple and the actions will direct descendants of the Struts `Action` class.

The `com.oreilly.struts.banking.action.LoginAction` class is shown in Example 3-3. It extends the Struts `Action` class and is invoked by the controller when a user attempts to login to the banking application.

Example 3-3. The `LoginAction` used by the online banking application

```
package com.oreilly.struts.banking.action;

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

```
import org.apache.struts.action.*;
import com.oreilly.struts.banking.IConstants;
import com.oreilly.struts.banking.service.IAuthentication;
import com.oreilly.struts.banking.service.SecurityService;
import com.oreilly.struts.banking.service.InvalidLoginException;
import com.oreilly.struts.banking.view.UserView;
import com.oreilly.struts.banking.form.LoginForm;
/**
 * This Action is called by the RequestProcessor when a login attempt
 * is made by the user. The ActionForm should be an instance of
 * a LoginForm and contain the credentials needed by the SecurityService.
 */
public class LoginAction extends Action {
    public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {

        UserView userView = null;

        // Get the credentials from the LoginForm
        String accessNbr = ((LoginForm)form).getAccessNumber();
        String pinNbr = ((LoginForm)form).getPinNumber();

        /*
         * In a real application, you would typically get a reference
         * to a security service through something like JNDI or a factory.
         */
        IAuthentication service = new SecurityService();

        // Attempt to login
        userView = service.login( accessNbr, pinNbr );

        // Since an exception wasn't thrown, login was successful

        // Invalidate existing session if it exists
        HttpSession session = request.getSession(false);
        if(session != null) {
            session.invalidate();
        }

        // Create a new session for this user
        session = request.getSession(true);

        // Store the UserView into the session and return
        session.setAttribute( IConstants.USER_VIEW_KEY, userView );
        return mapping.findForward( IConstants.SUCCESS_KEY );
    }
}
```

The `LoginAction` in Example 3-3 gets the credentials from the `ActionForm` that was passed in as an argument in the `execute()` method.

The `ActionForm` class will be discussed in the “The Struts View Components” later in the Chapter.

A `SecurityService` is then created and the `login()` method and passed the security credentials. If the login succeeds, a new `HttpSession` is created for the user and the `UIView` that is returned from the `login()` method is put into the session. If authentication fails, an `InvalidLoginException` will be thrown. You should notice that there's no try/catch block for the `InvalidLoginException` in the `execute()` method. This is because one of the new features of 1.1 is the built-in exception handling capabilities of the framework. Even better is the fact that it's declarative. This is a great new feature that removes much of the burden of exception handling from the developer. With the declarative exception handling in Struts, you specify what exceptions can be thrown from the actions and what you want the framework to do with them. You specify this information in the configuration file like this:

```
<global-exceptions>
  <exception
    key="global.error.invalidlogin"
    path="/login.jsp"
    scope="request"
    type="com.oreilly.struts.banking.service.InvalidLoginException"/>
</global-exceptions>
```

This fragment from the banking configuration file tells the framework that if an `InvalidLoginException` is thrown by any action, it should forward to the `login.jsp` resource and build an error message using the key “`global.error.invalidlogin`” from the resource bundle. You also have the ability to override the default exception handling behavior with whatever functionality that you need it to perform. Exception handling will be covered at length in Chapter 10.

Mapping the Actions

At this point, you might be asking yourself, “How does the `controller` know which `Action` instance to invoke when it receives a request?” The answer is by inspecting the request information and utilizing a set of action mappings.

Action mappings are part of the Struts configuration information that is configured in a special XML file. This configuration information is loaded into memory at startup and made available to the framework at runtime. Each `<action>` element is represented in memory by an instance of the `org.apache.struts.action.ActionMapping` class. The `ActionMapping` object contains a path attribute that is matched against a portion of the URI of the incoming request. We'll talk more about action mappings and the Struts configuration file in Chapter 4.

The following XML fragment illustrates the `login` action mapping from the configuration file used by the banking application.


```
<action
  path="/login"
  type="com.oreilly.struts.banking.action.LoginAction"
  scope="request"
  name="loginForm"
  validate="true"
  input="/login.jsp">
  <forward name="Success" path="/action/getaccountinformation" redirect="true"/>
  <forward name="Failure" path="/login.jsp" redirect="true"/>
</action>
```

The *login* action mapping shown here maps the path “/login” to the *Action* class *com.oreilly.struts.banking.LoginAction*. Whenever the controller receives a request where the path in the URI contains the string “/login”, the *execute()* method of the *LoginAction* instance will be invoked.

The Struts framework also uses the mappings to identify the resource to forward the user to once the action has completed. We’ll talk more about configuring action mappings in chapter 4.

Determining the Next View

We’ve talked about how the controller receives the request and how the action mappings and request information are used to determine the correct action instance to invoke and pass the request to. What hasn’t been discussed is how or what determines the view to return back to the client.

If you looked closely at the *execute()* method signature in the *Action* class from the previous section, you might have noticed that the return type for the method is an *org.apache.struts.action.ActionForward* class. The *ActionForward* class represents a destination to which the controller may send control once an Action has completed. Instead of specifying an actual JSP page in the code, you can declaratively associate an action forward mapping with the JSP and then use that *ActionForward* throughout your application.

The action forwards are specified in the configuration file, similar to action mappings. They can be defined for a specific action as this forward is for the *logout* action mapping.

```
<action
  path="/logout"
  type="com.oreilly.struts.banking.action.LogoutAction"
  scope="request">
  <forward name="Success" path="/login.jsp" redirect="true"/>
</action>
```

The *logout* action declares a *<forward>* element that is named “Success”, which forwards to a resource of “/login.jsp”. Notice in this case, a *redirect* attribute is set to “true”. Instead of performing a forward using a *RequestDispatcher*, the request that invokes the logout action mapping will be redirected instead.

The action forward mappings can also be specified in a global section independent of any specific action mapping. In the previous case, only the *logout* action mapping could reference the action forward named “Success”. In the case of forwards declared in the global forwards section, all action mappings can reference them. Here is an example global forwards section from the banking configuration file:

```
<global-forwards>
  <forward name="SystemFailure" path="/systemerror.jsp" />
  <forward name="SessionTimeout" path="/sessiontimeout.jsp" />
</global-forwards>
```

The forwards defined in the global section are more general and don't apply to a specific action. Notice that every forward must have a name and path, but the redirect flag is optional. If you don't specify a redirect attribute, its default value is false and thus performs a forward. The forward behavior can be configured and we'll discuss it more in the next chapter. The path attribute in an *ActionForward* can also specify another Struts Action. You'll see an example of how to do this in Chapter 5.

Now that you understand from a high-level how the Struts controller components operate, it's time to look at next piece of the MVC puzzle, the Model.

Struts Model Components

There are several different ways to look at what constitutes a model for Struts. The lines between business and presentation objects can get quite blurry when dealing with web applications. One application's business objects are another's value objects.

It's important to keep the business objects separate from the presentation, so that the application is not tightly coupled to one type of presentation. It's very likely that the look and feel of a web site will change over time. Studies show that the freshness of a web site's appearance helps to attract new customers and also keep existing customers coming back. This may not be as true in the *Business-to-Business* (B2B) world, but it's definitely true for *Business-to-Consumer* (B2C) applications, which make up the majority of the web applications used today.

Using Value Objects

Value objects are often used to provide a *coarse-grained* view of remote, find-grained data. For example, if your application were using entity beans, instead of making several remote calls to get individual state about the object, you would make a single call that returned a local value object that contained all of the data that you need. There is sometimes a summary and a detailed value object for the remote object to help soften how much data is returned.

Although the value object represents the remote business object, it doesn't necessarily contain the same business logic. In fact, it might not contain any business logic at all. It really represents a "snapshot" of the remote object at a particular instance in time, namely when the client requested the data. Value objects can also be used to update the business object as well. However, it gets a little more complicated because of issues like optimistic locking and synchronization, which we will discuss later.

Value objects are also referred to as "View" objects because they are primarily used by the presentation views as presentation JavaBeans. Because of performance reasons, using value objects in a distributed application is almost a necessity. You might be better off by using the same technique for smaller applications as well, so that the database or business objects are not the ones used within the presentation views. This helps to decouple the business objects from the presentation, making maintenance and future enhancements easier.

The type of model components that you use might also depend on whether you're building a traditional two-tier application or a multi-tiered distributed application. Typically with a two-tiered application, the business objects are colocated with the web application. *Collocation* means that objects are deployed within the same Java Virtual Machine. This makes it easier to utilize these business objects to retrieve data for the views. However, just because it's easier doesn't naturally make this a smart thing to do. The business objects may be made up of very deep object graphs and contain references to many other non-presentational resources. If you're not careful, the business objects can quickly become coupled to a specific presentation, which could mean unintended side effects each time the look and feel of the web site changes.

Another benefit of separating your business objects from the presentation objects is that you can build coarse-grained objects that your JSP pages and custom tags can have an easier time dealing with. All of your business logic should remain separate from the presentation and the presentation views should simply retrieve data from the value objects and display it.

The `LoginAction` class that was shown in Example 3-3 didn't contain the actual authentication logic. Since the action class is part of the controller functionality, it delegates the handling of the business logic to another service. In the case of the `LoginAction`, it relies on a `SecurityService` component. This component could have been a remote reference to a session EJB or maybe just a wrapper around some JDBC code that performed the authentication. In either case, the `LoginAction` doesn't know or care how the service is implemented. This is very helpful because the implementation could change drastically and as long as the `IAuthentication` interface remained unchanged and was implemented by the service, little code would have to change. This approach also helps with reuse. Imagine if you have another type of client, like a Swing GUI, that needed to be authenticated. Since the logic is encapsulated into a separate component and not in the `Action` class, you are free to reuse this security service.

You should strive to keep business logic out of the `Action` classes to protect against change. In the case of the `LoginAction`, the `login()` method returned an object of class `com.oreilly.struts.banking.view.UserView`. This is a good example of how to use value objects effectively. Example 3-4 shows the `UserView` used in the example application.

Example 3-4. The `UserView` value object, used by the presentation tier

```
package com.oreilly.struts.banking.view;

import java.util.Set;
import java.util.HashSet;
/**
 * A value object for that wraps all of the user's security information
 */
public class UserView implements java.io.Serializable {
    private String id;
    private String lastName;
    private String firstName;

    // A unique collection of permission String objects
    private Set permissions = new HashSet();

    /**
     * Constructors
     */
    public UserView(String first, String last) {
        this(first, last, new HashSet());
    }

    public UserView(String first, String last, Set userPermissions) {
        super();
        firstName = first;
        lastName = last;
        permissions = userPermissions;
    }
}
```

```
/**
 * Returns true if the argument is among the collection of
 * permissions allowed for the user. Otherwise returns
 * false.
 */
public boolean containsPermission(String permissionName) {
    return permissions.contains(permissionName);
}
/**
 * Retrieve the last name of the user
 */
public String getLastName() {
    return lastName;
}
/**
 * Set the last name of the user
 */
public void setLastName(String name) {
    lastName = name;
}
/**
 * Retrieve the first name of the user
 */
public String getFirstName() {
    return firstName;
}
/**
 * Set the first name of the user
 */
public void setFirstName(String name) {
    firstName = name;
}
/**
 * Retrieve the id of the user
 */
public String getId() {
    return id;
}
/**
 * Set the id of the user
 */
public void setId(String id) {
    id = id;
}
}
```

The `UserView` provides a coarse-grained view of a remote object. There might be five security tables all joined by foreign keys that contain the data, but when the web tier gets a `UserView`, it has already been consolidated and made easy to access. In fact, one implementation of this application could get the data from a relational database and another from a *Lightweight Directory Access Protocol* (LDAP) instance. The nice thing about encapsulating the authentication behind the security service is that the presentation tier would not have to change when the security realm was switched. The `Action` is free

to set the `UIView` object into the request and then forward to a JSP, where the data can be extracted and presented to the user.

The framework doesn't have a great deal of support in the way of the model components. This is better left for EJB, JDO, or some other type of persistence framework. You can also access a database directly from the framework, but you should still attempt to separate that layer from all other parts of the framework. You can do with by making use of the appropriate design patterns to encapsulate the behavior.

The Struts View Components

The last of the MVC components to discuss are the Struts View components. Arguably, it's the easiest to understand. The view components that are typically employed in a Struts application are:

- `JavaServer Pages`
- `Custom Tags`
- `HTML`
- `Java Resource Bundles`
- `Struts Action Forms`
- `Value Objects`

Using the Struts ActionForm

Struts `ActionForm` objects are used in the framework to pass client input data back and forth between the user and the business layer. The framework will automatically collect the input from the request and pass this data onto an `Action` using a form bean, which then can be passed along to the business layer. To keep the presentation layer decoupled from the business layer, you should not pass the action form itself to the business layer, but rather create the appropriate value objects using the data from the form and pass these objects as argument to the business layer. The following steps illustrate how the framework processes an `ActionForm` for every request.

1. Check the mapping for the action and see if a form bean has been configured for it.
2. If so, use the `name` attribute to lookup the form bean configuration information.
3. Depending on the scope configured for the form bean for the action, check to see if there's already an instance of the form bean at the appropriate scope.
4. If an `ActionForm` instance is present in the appropriate scope and it's the same type as needed for this new request, reuse it.

5. Otherwise, create a new instance of the required form bean and store it into the appropriate scope that is set by the *scope* attribute for the action mapping.
6. The `reset()` method is called on the `ActionForm` instance.
7. Iterate through the request parameters and for every parameter name that has a corresponding set method name on the `ActionForm`, populate it with the value for that parameter.
8. Finally, if the `validate` attribute is set to true, then invoke the `validate()` method on the `ActionForm` instance and return any errors.

For every HTML page where form data is posted, you should use an `ActionForm`. The action forms can be used for multiple pages if necessary to collect data over several pages.

Example 3-5 shows the `com.oreilly.struts.banking.form.LoginForm` that is used with the banking application.

Example 3-5. The LoginForm used with the online banking application

```
package com.oreilly.struts.banking.form;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;
import org.apache.struts.util.MessageResources;
/**
 * This ActionForm is used by the online banking appliation to validate
 * that the user has entered an accessNumber and a pinNumber. If one or
 * both of the fields are empty when validate is called by the
 * ActionServlet, error messages are created.
 */
public class LoginForm extends ActionForm {
    // The user's private id number
    private String pinNumber;
    // The user's access number
    private String accessNumber;

    /**
     * Default Constructor
     */
    public LoginForm() {
        super();
        resetFields();
    }

    /**
     * Called by the framework to validate the user has entered the
     * accessNumber and pinNumber fields.
     */
    public ActionErrors validate(ActionMapping mapping, HttpServletRequest req ){
        ActionErrors errors = new ActionErrors();
```

```
// Get access to the message resources for this application
// There's not an easy way to access the resources from an ActionForm

// This assumes our messages are coming from the default MessageResources
MessageResources resources =
    (MessageResources)req.getAttribute( Action.MESSAGES_KEY );

// Check and see if the access number is missing
if(accessNumber == null || accessNumber.length() == 0) {
    String accessNumberLabel = resources.getMessage( "label.accessnumber" );

    // Create an error message for the missing accessNumber value
    ActionError newError =
        new ActionError("global.error.login.requiredfield", "");
    errors.add(ActionErrors.GLOBAL_ERROR, newError);
}

// Check and see if the pin number is missing
if(pinNumber == null || pinNumber.length() == 0) {
    String pinNumberLabel = resources.getMessage( "label.pinnumber" );

    // Create an error message for the missing pingNumber value
    ActionError newError =
        new ActionError("global.error.login.requiredfield", "Pin Number");
    errors.add(ActionErrors.GLOBAL_ERROR, newError);
}
// Return the ActionErrors, in any.
return errors;
}

/**
 * Called by the framework to reset the fields back to their default values.
 */
public void reset(ActionMapping mapping, HttpServletRequest request) {
    // Clear out the access number and pin number fields
    resetFields();
}

/**
 * Reset the fields back to their defaults.
 */
protected void resetFields() {
    this.accessNumber = "";
    this.pinNumber = "";
}
public void setAccessNumber(String nbr) {
    this.accessNumber = nbr;
}
public String getAccessNumber() {
    return this.accessNumber;
}

public String getPinNumber() {
    return this.pinNumber;
}
```



```
}  
public void setPinNumber(String nbr) {  
    this.pinNumber = nbr;  
}  
}
```

The `ActionForm` class provided by the Struts framework implements several methods, but by far the two most important are the `reset()` and `validate()` methods.

```
public void reset(ActionMapping mapping, HttpServletRequest request);  
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request);
```

The default implementation for both methods in the Struts `ActionForm` class doesn't perform any default logic. You'll need to override these two methods in your `ActionForm` classes as was done in the `LoginForm` class shown in Example 3-5.

The controller calls the `reset()` method right before it populates the `ActionForm` instance with values from the request. It gives the form bean a chance to reset their properties back to the default state. This is very important as the form bean instance may be shared across different requests and different users, so this can be a very important method to implement. However, if you are using an `ActionForm` instance across multiple pages, you might not want to implement the `reset()` method so that the values don't get reset until you're completely done with the instance. Another approach is to implement your own `resetFields()` method and call this method from the action class after a successful update to the business tier.

The `validate()` method is called by the controller after the values from the request have been inserted into the form bean. The form bean should perform any field validation that can be done and return any detected errors back to the controller. Business logic validation should be performed in the business objects, not in the `ActionForm`. The validation that occurs in the `ActionForm` is more about presentation input validation. Where to perform certain types of validation logic will be covered in detail in chapters 6 and 7.

The `validate()` method in the `LoginForm` in Example 3-5 checks to see if the access number and/or pin number is missing and creates error messages if they are. If no errors are generated, the controller then passes the `ActionForm` on with several other objects to the `execute()` method. The `Action` instance can then pull the information out of the form bean, as with any other Java object.

You might have noticed that the `execute()` method in the `Action` class contains an argument that is always of the type `ActionForm`. You will need to cast this argument to the appropriate subclass to retrieve the needed properties. If you need to see an example of this, look back at Example 3-3.

Once you've created your `ActionForm` classes, you need to inform Struts that they exist and which action mappings use which action forms. This is done in the

configuration file, as you might have guessed. The first step is to configure all of the form bean classes. This is done in the `<form-beans>` section of the configuration file. The following fragment from the banking configuration informs Struts of the three `ActionForm` beans used by the banking application:

```
<form-beans>
  <form-bean name="loginForm" type="com.oreilly.struts.banking.form.LoginForm"/>
  <form-bean
    name="accountInformationForm"
    dynamic="true"
    type="org.apache.struts.action.DynaActionForm">
    <form-property name="accounts" type="java.util.ArrayList"/>
  </form-bean>
  <form-bean
    name="accountDetailForm"
    dynamic="true"
    type="org.apache.struts.action.DynaActionForm">
    <form-property
      name="view"
      type="com.oreilly.struts.banking.view.AccountDetailView"/>
  </form-bean>
</form-beans>
```

The name attribute for each form bean must be unique and the type attribute must define a fully qualified Java class that extends the Struts `ActionForm` class.

The next step is to use one of the form bean names from the `<form-beans>` section for one or more action mappings. The following fragment shows the mapping for the `LoginAction`, which you've already seen earlier in this chapter.

```
<action
  path="/login"
  type="com.oreilly.struts.banking.action.LoginAction"
  scope="request"
  name="loginForm"
  validate="true"
  input="/login.jsp">
  <forward name="Success" path="/action/getaccountinformation" redirect="true"/>
  <forward name="Failure" path="/login.jsp" redirect="true"/>
</action>
```

Notice how the name attribute of the login mapping matches one of the names in the `<form-beans>` section. The Struts developers should have probably called this attribute `<form-beans>`, but hindsight is always 20/20. The names of the form beans are case sensitive.

One of the new features with 1.1 is shown in the previous `<form-beans>` fragment. With previous versions of the framework, you had to always extend the `ActionForm` class with your own subclass. Even is the action form performed very generic behavior. With Struts 1.1, a new type of action form has been added called `org.apache.struts.action.DynaActionForm`. This class can be configured for an action form for an action mapping and will automatically handle the data passed

from the HTML form to the action. It's able to be generic because internally it uses a Map to store the values. This allows it to store any object. Chapter 7 will cover the dynamic action form in more detail.

We'll wrap up this section and talk about something that can be a little confusing—what's the difference between a form bean that we've seen here and the value objects that we mentioned earlier? This is a good question and one that is a little confusing for developers new to Struts.

The view components can utilize both form beans and value objects to populate dynamic content. In cases where you don't have a form bean configured for a mapping, value objects can be used to build the views. In cases where a form bean is defined for the mapping, there are several different ways to handle extracting the data from the bean. One approach is to always wrap a form bean around the value object or objects returned from the business tier and force the Struts view components to access the value object data through the form bean methods. Likewise, when a client submits an HTML page, Struts will invoke the form bean setter methods, which can shove the data back into the view objects after the validation method has completed successfully. This provides a single cohesive interface for the views to retrieve and submit the HTML data to. We'll discuss the various pros and cons to this and other approaches in Chapter 7.

Using JavaServer Pages for Presentation

JavaServer Pages make up the majority of what has to be built for the Struts view components. Combined with custom tag libraries and HTML, it becomes easy to provide a set of views for an application. Although JavaServer Pages make up the majority of what organizations and developers are using to display the dynamic content, it's not the only technology.

There are other forms of presentation technologies that can be combined with the Struts framework. One very popular one is the XML/XSLT combination. This alternate model is being referred to as Model 2X, which is a combination of the controller servlet from the Struts framework and XSLT and beans serialized from the value objects to render the views. Many developers feel that JSP has the following problems:

- Developers are free to embed application logic into the JSP pages. This can lead to an application that is difficult to maintain.
- JSP syntax is not currently XML compliant, which may cause the XML or HTML that gets generated, not to be *“well formed”*.
- Developers must learn the JSP syntax and how to program custom tags.
- Developing a processing pipeline where each node in the pipeline may modify the data or layout is not possible with JSP pages. This makes it difficult to separate layout and style.

- A recompile of the JSP page is necessary for each change made to the page.

Some developers may see these issues as not any big deal. Many sites have been built using the JavaServer Pages technology. However for those that want alternatives, they are available. In Chapter 16 of the book, we'll look at alternate presentation technologies can used by the framework.

Custom Tag Libraries

The Struts framework provides five core tag libraries that can be used by your applications. Each one has a different purpose and can be used individually or along side the others. You may also create your own custom tags and can even extend the Struts tags if you need them to perform extra functionality. The custom tag libraries that are included with the framework are:

- HTML Tag Library
- Bean Tag Library
- Logic Tag Library
- Template Tag Library
- Nested Tag Library

Unless you are planning on using templates as part of your application, the Template Tag library might not be necessary, but the others are invaluable to making your application easy to develop and maintain.

To use the libraries in your application, you need to first register them with the web application. To do this, you should add the following lines to the deployment descriptor for each web application that you wish to use Struts for.

```
<web-app>

  <taglib>
    <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
```

```
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-nested.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>
</web-app>
```

More information on installing and configuring Struts for your application can be found in Appendix B.

The next step is to create your JSP pages and include the following lines.

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>
```

Once this is done and the Struts JAR file is in the web application's CLASSPATH, you are then able to use the custom tags in your JSP pages. Example 3-6 illustrates the usage of several of the Struts custom tags inside the *login.jsp* page for the banking application.

Example 3-6. The login.jsp used by the online banking application

```
<%@ page language="java" contentType="text/html" %>

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html>
<head>
  <html:base/>
  <title><bean:message key="title.login"/></title>
  <link rel="stylesheet" href="stylesheets/login_style_ie.css" type="text/css">
</head>

<body topmargin="0" leftmargin="5" marginheight="0" marginwidth="0"
bgcolor="#6699FF">

<html:form action="login" focus="accessNumber">

<table border="0" cellpadding="0" cellspacing="0" width="100%" bgcolor="#6699FF">
  <tr><td>
    <html:img srcKey="image.logo" width="79" height="46"
      altKey="image.logo.alt" border="0"/>
  </td></tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="100%">
  <tr><td bgcolor="#000000">
    <table border="0" cellpadding="0" cellspacing="0" width="1" height="2"></table>
  </td></tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="1" height="1">
```

```
<tr><td></td></tr>
</table>

<table>
  <tr><td></td></tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="590">
  <tr><td width="15" height="31"></td><td width="12"></td></tr>
  <tr>
    <td width="15"></td>
    <td width="575" bgcolor="#FFFFFF" colspan="2">
      <table cellpadding="0" cellspacing="0" border="0" width="575" height="3">
        <tr><td></td></tr>
      </table>
    </td>
  </tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="590" bgcolor="#ffffff">
  <tr>
    <td width="15" bgcolor="#6699FF"></td>
    <td width="15"></td><td width="379"></td>
    <td width="15"></td>
    <td width="15"></td>
    <td width="15"></td>
  </tr>
  <tr>
    <td bgcolor="#6699FF" width="15"></td>
    <td></td>
    <td valign="top">
      <table border="0" cellpadding="0" cellspacing="0">
        <tr class="fieldlabel">
          <td><bean:message key="label.accessnumber"/></td>
        </tr class="fieldlabel">
          <td><html:text property="accessNumber" size="9" maxlength="9"/></td>
          <td class="error"><html:errors/></td>
        </tr>
        <tr class="fieldlabel"><td height="10"></td></tr>
        <tr class="fieldlabel"><td><bean:message key="label.pinnumber"/></td></tr>
        <tr class="fieldlabel">
          <td><html:password property="pinNumber" size="4" maxlength="4"/></td>
        </tr>
        <tr><td height="10"></td></tr>
        <tr><td><html:submit styleClass="fieldlabel" value="Login"/></td></tr>
        <tr><td></td></tr>
      </table>
    </td>
  </tr>
  <td width="151" valign="top">
    <html:img srcKey="image.strutspower" altKey="image.strutspower.alt"/>
  </td>
</tr>
</table>
<%@include file="include/footer.jsp"%>
```

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

One of the first things that should strike you about the login page in Example 3-6 is just how little Java code there is. In fact, there's none. Instead you see mostly HTML formatting tags and several uses of Struts tag libraries. This is exactly the purpose of using custom tag libraries. Because there's no Java programming necessary, HTML designers can work freely with the page layout without being burdened by the programming aspects of the page. The other nice feature is that many JSP pages can use the same tag. For more information on these and other tag libraries, see Chapter 8, "Custom Tag Libraries".

As you can probably imagine by looking at the JSP page in Example 3-6, maintenance and customization becomes very easy to support. In fact, one of the hardest customizations that developers face is to quickly and effortlessly customize a web application for multiple languages. There are several built-in features of the Java language that help support Internationalization aspects, and Struts builds on top of that support to provide more.

Message Resource Bundles

The Java library includes a set of classes to support reading message resources from either a Java class or a properties file. The core class in this set is the `java.util.ResourceBundle`. The Struts framework provides a similar set of classes, based around the `org.apache.struts.util.MessageResources` class that provides similar functionality, but provides for a little more flexibility that the framework requires.

The standard Java support for Internationalization has grown with the past several releases and the Struts framework could probably use what's included with 1.2 and newer, but since Struts was created before 1.2, they had to build in their own support for several key pieces.

With a Struts application, you must provide a Java message bundle for each language that you wish to support. The name of class or properties file must adhere to the guidelines listed in the JavaDocs for the `java.util.ResourceBundle` class. Example 3-7 shows the properties file used by the example banking application.

Example 3-7. The message resources used by the online banking application

```
# Labels
label.accessnumber=Access Number
label.pinnumber=Pin Number
label.accounts=Accounts
label.balance=Balance
```

```
label.totalassets=Total Assets
label.account=Account
label.balance=Available Balance
label.description=Description
label.amount=Amount
label.deposits=Deposits
label.withdrawls=Withdrawls
label.openingbalance=Opening Balance

# Links
link.customeragreement=Customer Agreement
link.privacy=Privacy
link.security=Security
link.viewaccountdetail=View Account Detail

# Page Titles
title.login=Struts Online Banking - Account Login
title.accountinfo=Struts Online Banking - Account Information
title.accountdetail=Struts Online Banking - Account Detail

# Button Labels
label.button.login=Login

# Error messages
global.error.invalidlogin=<li>Invalid Access Number and/or Pin</li>

global.error.login.requiredfield=<li>The {0} is required for login</li>

# Images
image.logo=images/logo.gif
image.logo.alt=Struts Online Banking

image.logout=images/logout.gif
image.logout.alt=Logout

image.strutspower=images/struts-power.gif
image.strutspower.alt=Powered By Struts

image.transfer=images/transfer.gif
image.transfer.alt="Transfer Funds"

image.clear=images/clear.gif
```

If you look back at the login JSP page in Example 3-6, you can see how the messages from the bundle are used. For example, the following fragment from the login JSP page illustrates the key `title.login` from Example 3-6 being used and inserted between the HTML `<title>` tags in the page.

```
<title><bean:message key="title.login"/></title>
```

The Struts `org.apache.struts.taglib.bean.MessageTag` is one of several custom tags included in the framework that can take advantage of the resource bundle. JSP pages can retrieve values from the resource bundle using the `MessageTag` based on a key as shown in the login JSP page from Example 3-6. The key in the message tag must

correspond to a value on the left side of the equal sign in the bundle. Case is very important and the value must match exactly.

A message resource bundle is used for reasons other than just for localization. It can also save time during application maintenance. For example, if you use the same text messages or labels throughout various parts of your web site or application, when one or more of these values need to change, you only need to make the change in a single location. Even if you don't have requirements for Internationalization, you should still utilize resource bundles.

With Struts 1.1, you now have the ability to define multiple `MessageResources` for an application. This allows you isolate certain types of resources, into separate bundles. For example, you might want to store the image resources for an application into one bundle and the rest of the resources into another. How you organize your application's resources is up to you, but you now have the flexibility to separate them based on some criteria. Some applications choose to separate along components lines. For example, all resources relating to the catalog go into one bundle, order and shopping cart resources into another, and so on. Chapter 21 "Struts Design Strategies" will discuss these and other strategies for organizing you resource bundles.

Multiple Application Support

Prior to version 1.1, each Struts application was limited to having a single configuration file. The single instance of the file, which is normally called `struts-config.xml`, was specified in the web application deployment descriptor. It was the sole provider of the configuration information for the Struts application. The fact that there was only a single place to put configuration information made it very difficult for larger projects because it often became a bottleneck and caused contentions to use and modify this file.

With the creation of 1.1, this problem has been alleviated with the advent of multi-application support. You can now define multiple configuration files and allow developers to work better in parallel. Multiple applications or multiple sub-apps as some are calling this new feature will be discussed further in Chapters 4, 5, 6, and 7.

Summary

As you learned in this chapter, the Struts framework provides an implementation for the MVC structure, tailored for a web application. The Struts `ActionServlet`, `RequestProcessor`, and `Action` classes provide the Controller components; the `Action` communicates with your application's model components, and finally the

combination of the `ActionForm`, value objects, JSP pages, and tag libraries make up the View.

This chapter focused on Struts at a very high level and left out many of the details that make the framework even better. Struts, like other valuable software frameworks, allow you to focus on developing the business logic, instead of spending expensive development time on low-level infrastructure functionality like request dispatching and field-level validation. Hopefully this peripheral discussion has enticed you to read on and explore the framework details in the next several chapters.

4

Configuring the Struts Application

The Struts framework uses two separate, but somewhat related types of configuration files, which must be properly configured before an application will function properly. Due to the popularity and flexibility of the self-describing nature of XML, both of these configuration files are based on XML.

The web application deployment descriptor *web.xml* is described fully in the Java Servlet specification. This configuration file is necessary for all web applications, not just those built with the Struts framework. There is however, Struts specific deployment information that must be configured within it when building web applications using Struts.

Although the Struts framework supports the 2.2 Servlet Specification, many servlet containers already have support for version 2.3. This book will include coverage of the 2.3 Specification as well as 2.2.

The second configuration file that this chapter will examine is the Struts configuration file, commonly named *struts-config.xml*. As you'll see however, the name can be just about anything that you want it to. The Struts configuration file makes it possible for you to declaratively configure many of your application's settings. You can think of the Struts configuration file as the rules for the web application.

Introduction to the Storefront Application

Throughout the rest of the book, we are going to be using a shopping cart type application for all of the examples. The application will be built up along the way as we

go through the Chapters and at the end of the book, we should have a fairly complete application that uses every 1.1 feature. Figure 4-1 shows the main page of the storefront application.

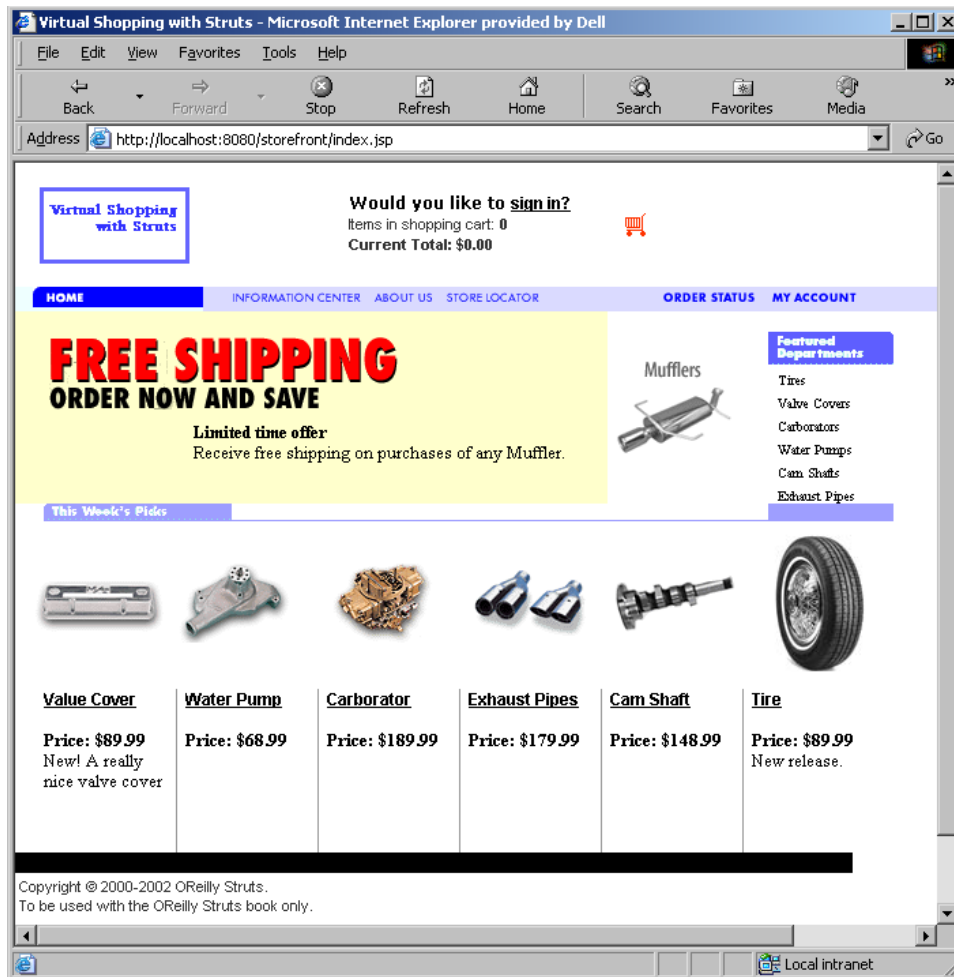


Figure 4-1. The main page of the example storefront application

This storefront application will demonstrate an e-commerce automotive parts supplier, but you can substitute any items that you want, as long as you have your own images and data to put into the application. In the end, you'll have the storefront application as a complete WAR file that can be deployed into any compliant web container and used as an example for many different purposes.

What is a Web Application?

Applications built using the Struts framework are at their core, web applications. A *web application* is a collection of individual components that once bound together, form a complete application that can be installed and executed by a web container. The components are tied together due to the fact that they reside in the same web context and in many cases, may refer to one another, directly or indirectly. For example, if you have an application that is rooted on a web server under a directory called *storefront*, then all of the files that are in this directory and below are considered part of the *storefront* web application. Any reference to a resource with the storefront prefix is routed to this web application. So, if a user typed in the following URL:

<http://www.somehost.com/storefront/index.jsp>

then the JSP page would be served from the root of the *storefront* web application.

A web application can be installed and executed in multiple web containers concurrently. For that matter, multiple instances of the same web application can be installed in the same web container. However, because of the manner in which URL's are matched to the correct web application, each web application must have a unique name within the web container. This means that you can't have two web applications running in the same web container using the same name.

Let's take a closer look at exactly what type of components can reside in a web application.

Elements of a Web Application

Obviously, not all web applications are created equal. They will not have the same functional and non-functional requirements across organizations, departments, or even the same vertical markets. Therefore, not all web applications will contain the same types of resources. In general however, web applications can consist of one or more of the following types of components:

- Servlets
- JSP Pages
- Standard JavaBeans and Utility Classes
- HTML Documents
- Multimedia Files (Images, Audio and Video Files, CAD Drawings, etc...)
- Client side Applets, Stylesheets, and JavaScript Files
- Text Documents

- Meta information that ties all of the above components together

The Web Application Directory Structure

A web application typically consists of a structured hierarchy of directories. Although the servlet specification does not require servlet containers to support a hierarchical structure, it is highly recommended and most, if not all do. The root directory of the hierarchy serves as the document root for the web application. As you saw earlier, requests that are made using the web application's root context path will be served out of the directory for that web application.

Within the web application directory hierarchy, a special directory exists named *WEB-INF*. This directory should contain meta-information relating to the web application. This directory is a private directory and therefore no resource within this directory should be able to be requested by a client. However the resources in the *WEB-INF* directory are visible to servlets and Java classes that reside within the web application.

Because the servlet specification requires that the *WEB-INF* directory not be visible to a web client, this is an ideal location to put files and other resources that you do not wish to expose directly to a client. Resources like XML configuration files and other private application resources should be placed within this directory for the web application. As you'll see later in this chapter, the Struts configuration file is also normally located in this directory.

The *WEB-INF* directory is where the deployment descriptor for the web application should be placed. The deployment descriptor is covered in detail in the next section.

There are two special directories underneath *WEB-INF* that get special treatment by the web container. The *WEB-INF/classes* directory is used to place servlets and utility classes that can be used by the web application. If the Java classes are scoped within a Java package, the *classes* directory must contain the proper sub-directories within it that match the package name.

For example, suppose you had a Java servlet named `com.oreilly.struts.framework.StorefrontController` for a web application named storefront. The `StorefrontController.class` file would have to be placed in the framework directory as shown in Figure 4-2.



Figure 4-2. Java classes that are in a package must be in the proper directories

The other special sub-directory underneath **WEB-INF** is the **lib** directory. The **WEB-INF/lib** directory is an area where JAR files can be deployed and will be picked up by the class loader for the web application.

Based on the 2.3 Servlet Specification, the web application class loader must load classes from the **WEB-INF/classes** directory first, and then from library JARs located in the **WEB-INF/lib** directory.

Other than these special requirements, the directory structure for a web application is left up to the developer. It should be based on the functional and non-functional needs of the application. With smaller web applications, files and resources may be combined into a few common directories. For larger web applications however, each component may need to have a separate directory underneath the web application root directory. This will allow for easier development and maintenance. Figure 4-3 shows a directory structure for an example storefront web application.

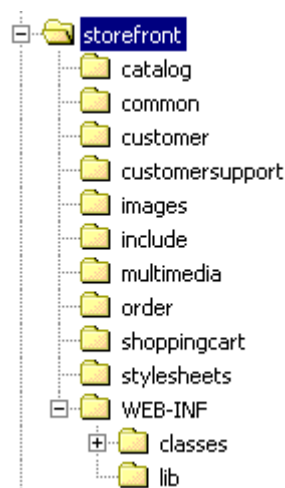


Figure 4-3. The directory structure for the storefront web application

Here are a few things to consider when choosing the directories for your Struts applications are:

- Keep any of the components that are optional separated from the required ones so that partial deployment will be easier to support
- Take into account the size of the development team and the necessity to prevent file and checkout contention
- Be careful to consider file and resource dependencies and to make sure to take advantage of reuse and include files

Other than these simple guidelines, the directory structure for your web application is entirely up to you.

Web Application Archive Files

Web applications can be packaged into a format called *Web ARchive* format (WAR) using the *Java ARchive* tool (JAR). The extension of the file must be *.war*. For example, if we archived the storefront application, it would be named *storefront.war*. When a web application is packaged as a *WAR* file, it must maintain its relative directory as illustrated in Figure 4.3.

Web containers are capable of loading the *WAR* file and un-packing the resources back into the necessary structure that can be loaded by the container. The WAR file format is most useful when you need to distribute an application. It can also be part of a much larger distributable file called *Enterprise ARchive* format (EAR). Chapter 20 will discuss the best practices of packaging your application using these different formats.

Web Application Deployment Descriptor

The web application deployment descriptor helps to convey configuration information between application developers, deployers, and assemblers. Web containers also utilize the descriptor to help configure and load the web applications when the container is started.

The following types of deployment information are required to be supported by all servlet containers that are compliant to the Servlet 2.3 Specification.

- Initialization Parameters
- Session Configuration
- Servlet Declaration
- Servlet Mappings

- Application Lifecycle Listener Classes
- Filter Definitions and Mappings
- MIME Type Mappings
- Welcome File List
- Error Pages

Security configuration information is not required unless the servlet container is part of a J2EE implementation. The following elements are not required unless the servlet container is using JSP pages or is part of a J2EE application server.

- Tag libraries Mappings
- JNDI References

Much of the functionality described in this list was added during the 2.3 version of the servlet specification. If you are using a 2.2 compliant container, it will not be available to you. However, the Struts framework currently supports 2.2 as well as 2.3.

Web Application Deployment Descriptor DTD

The format for both the web deployment descriptor and the Struts configuration file is based on a *Document Type Definition* (DTD), which defines the legal building blocks that may be used in the XML files. From the DTD point of view, all XML documents, including the web application deployment descriptor and the Struts configuration files, are made up of the following elements:

- Elements
- Tags
- Attributes
- Entities
- PCDATA
- CDATA

Using these components, DTDs help specify what is a valid and well-formed* XML document. The DTD for the 2.3 web application deployment descriptor can be downloaded from the following URL:

<http://java.sun.com/dtd/index.html>

The following DTD declaration shows the top-level elements that make up the deployment descriptor for a web application.

```
<!ELEMENT web-app (icon?, display-name?, description?,
    distributable?, context-param*, filter*, filter-mapping*,
    listener*, servlet*, servlet-mapping*, session-config?, mime-
    mapping*, welcome-file-list?, error-page*, taglib*, resource-
    env-ref*, resource-ref*, security-constraint*, login-config?,
    security-role*, env-entry*, ejb-ref*, ejb-local-ref*)
>
```

The *web-app* element is the root of the deployment descriptor for a web application. The other elements inside the parenthesis are child elements, which must be placed inside the root *web-app* element within the XML file. The symbols next to the child elements indicate the allowed multiplicity of the child elements within the XML file. Table 4-1 provides a brief explanation of the symbols.

Table 4-1. Multiplicity symbols of child elements within a DTD

Symbol	Meaning
No Symbol	When there is no symbol next to the child element, this indicates that the child must occur once and only once within the parent element.
+	The + sign declares that the child element can occur one or more times within the parent element.
*	The * sign declares that the child element can occur zero or more times within the parent element. This symbol is used quite often.

* A *well-formed* XML document is one that is properly formatted with all begin tags closing with end tags, attributes are quoted properly, the entities are declared, and so on. When an XML document is *well-formed*, it is easier for a computer program to parse it and possibly deliver it over a network. A *valid* XML document is one declares a DTD and adheres to the rules set forth in that DTD.

?

The ? sign declares that the child element occurs zero or one time within the parent element. In other words, the child element is optional. This symbol is used quite often.

Configuring the *web.xml* file for Struts

Although the *web.xml* file is used for configuring any generic web application, there are a few Struts specific configuration options that you must configure within this file when using the Struts framework. The next section describes the necessary steps that you'll need to perform to ensure that your Struts applications are properly configured.

Mapping the Struts *ActionServlet*

The first and maybe most important step that you need to perform is to configure the Struts *ActionServlet* that will receive all incoming requests for the application.

You only need to configure a single *ActionServlet*, regardless of the number of sub-applications that are being used. Some developers choose to setup multiple controller servlets to handle different functional areas of the application. Since servlets are multi-threaded, you don't gain any real performance or scalability value by using multiple *ActionServlet* mappings. Because Struts now supports multiple sub-applications, arguably there might not be any reason to have more than a single controller servlet for an application.

There are two steps in configuring the Struts controller servlet in the *web.xml* file. The first step is to use the *servlet* element to configure the servlet instance that can later be mapped in the *servlet-mapping* element. The child elements that are used in the *servlet* element are shown here.

```
<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
(servlet-class|jsp-file), init-param*, load-on-startup?, run-
as?, security-role-ref*)>
```

The child elements that we are most interested in for configuring Struts are *servlet-name*, *servlet-class*, and the *init-param*. The following partial *web.xml* file illustrates how these three elements are used to declare the Struts controller servlet instance.

```
<web-app>
  <servlet>
    <servlet-name>storefront</servlet-name>
    <servlet-class>com.oreilly.struts.framework.StorefrontController</servlet-class>
    <init-param>
      <param-name>config</param-name>
```

```
<param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
</servlet>
</web-app>
```

The *servlet-name* element specifies the canonical name that the servlet is referenced as throughout the rest of the deployment descriptor. Each servlet instance element must have a unique name within the web application. When configuring the *servlet-class* element for a Struts application, this element must specify a fully qualified Java class that extends the `org.apache.struts.action.ActionServlet` class.

Because the Struts `ActionServlet` class is not abstract, you are free to utilize that class and avoid having to create a subclass of the `ActionServlet` for your application. With earlier versions of the Struts, it was probably more important to extend the `ActionServlet` class with one of your own because most of the processing occurred there and sub classing allowed you to override that functionality with that of your own. With version 1.1 however, most of the processing functionality has been moved to the Struts `RequestProcessor` class, which you can configure declaratively as you'll see later in this chapter, and there is very little reason to create your own `ActionServlet` class, although you are still free to do so.

The other step that needs to be performed in order to configure the Struts controller servlet in the deployment descriptor is to configure the servlet mapping. This is done using the *servlet-mapping* element. The following partial deployment descriptor illustrates how combining the *servlet-mapping* element with the *servlet* element shown previously completes the `ActionServlet` configuration.

```
<web-app>
  <servlet>
    <servlet-name>storefront</servlet-name>
    <servlet-class>com.oreilly.struts.framework.StorefrontController</servlet-class>
  </servlet>

  etc...

  <servlet-mapping>
    <servlet-name>storefront</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Notice that the name given to the `StorefrontController` servlet within the *servlet* element is used in the *servlet-mapping* element. This tells the container that all requests that have an extension of `.do` should be serviced by the `StorefrontController` servlet instance.

Mapping Request to Servlets

This is probably a good time to digress for a moment and discuss exactly how URLs that a user types in are mapped to the correct web application and servlet. When a web application is installed into a container, the container is responsible for assigning a `ServletContext` to it. There is a single instance of a `ServletContext` object for each web application deployed in a container.

If the container is distributable and utilizes more than one JVM, the web application may have a separate `ServletContext` instance for each JVM.

The `ServletContext` provides an external view of the web application for the servlet. A servlet can use the `ServletContext` object to gain access to external resources, log events, and store attributes and objects that other servlet instances in the same context can access. It's essentially an application-scope shared resource.

Since a servlet is associated with a specific web application, all requests that begin with a specific request path, otherwise known as the *context path*, are routed to the web application and get associated with its `ServletContext` instance. Servlets that are associated with the default application have an empty string "" as the context path.

When a web container receives a client request, it must determine the correct web application to forward it to. It determines this by matching the request URL with the longest context path that matches an installed web application. For example, if there are two web applications installed in a container and one is rooted at <http://www.somehost.com/storefront> and the other at http://www.somehost.com/storfront_demo, if a client request arrived at the server with a request URL of http://www.somehost.com/storefront_demo/login.do, then the server would match it to the web application that had the closest match, which in this case would be the storefront demo application.

Once the container determines the correct web application; it must next determine which servlet from the web application should process the request. The web container uses the request URL, minus the context path to determine the path that will be used to map the request to the correct servlet. The web container uses the following guidelines. The first successful match is used with no further matches attempted.

1. The web container will attempt to locate an exact match of the request path to the path of a servlet.
2. The container will recursively try to match the longest path prefix. The servlet that contains longest match, if any, is selected.
3. If the URL path contains an extension, for example *.do*, the servlet container will try to match a servlet that handles requests for that extension. The extension is defined as the part of the last segment after the last '.' character.

4. If none of the previous rules produced a match, the container will attempt to use a default servlet if one is configured. Otherwise, the request will return an error response.

The web container must use case sensitive string comparisons when checking for a match.

The concept of extension mappings was mentioned in step 3 of the matching guidelines. There is another type of mapping that can be used, *path mapping*. A *servlet-mapping* that utilizes a path mapping allows a URL that doesn't contain an extension to match to the servlet. Using the storefront servlet mapping from before, the following partial *web.xml* illustrates how path mapping can be configured.

```
<web-app>
  <servlet-mapping>
    <servlet-name>storefront</servlet-name>
    <url-pattern>/action/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Using the path mapping here, all requests that get mapped to this web application and contain the string */action* in the request URL, will be serviced by the storefront servlet. This is regardless of what is in place of the "*" character.

Specifying Multiple Sub-Applications

As was briefly discussed in Chapter 3, the Struts 1.1 release has added the ability to define multiple Struts configuration files, one for each sub-application that is supported. In previous versions of the framework, you would configure a single Struts configuration file in the web application deployment descriptor by specifying a relative path to the configuration file using the *config* initialization parameter. With version 1.1 and the concept of multiple sub-applications, you are now able to create multiple Struts configuration files and specify them in the *web.xml* using multiple *config* initialization parameters and the sub-application prefix. The next section discusses the initialization parameters that can be configured for the servlet.

Declaring the Initialization Parameters

Initialization parameters are used to make configuration options available to a servlet. This allows the developer to declaratively affect the runtime environment of the servlet. Initialization parameters are configured within the servlet element using *<init-param>* elements as shown in the following *web.xml* fragment.

```
<web-app>
  <servlet>
    <servlet-name>storefront</servlet-name>
    <servlet-class>com.oreilly.struts.framework.StorefrontController</servlet-class>
    <init-param>
```

```
<param-name>config</param-name>
<param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<init-param>
  <param-name>host</param-name>
  <param-value>localhost</param-value>
</init-param>
<init-param>
  <param-name>port</param-name>
  <param-value>7001</param-value>
</init-param>
</servlet>
etc...

<servlet-mapping>
  <servlet-name>storefront</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

You can specify any parameters that you need within the *init-param* element as long as it's a name/value pair. For example, the previous web deployment descriptor included initialization parameters for a host and port. If you were using EJB, this might be a way to include the server connection information. Zero or more *init-param* elements are allowed.

There are specific initialization parameters that can be specified for the Struts servlet. In earlier versions of Struts, many of the configuration options that are now in the Struts configuration file, resided as *init-param* elements in the web deployment descriptor. Although applications that were originally built and tested with version 1.0 will still continue to work using the version 1.1, you may want to move some of the initialization parameters that you currently have specified in the web deployment descriptor, to the proper location in the Struts configuration file. Although the framework includes functionality that will allow the previous initialization parameters to work in the web.xml, we will be covering the 1.1 parameters here.

Table 4.1 identifies the initialization parameters that can be specified for Struts 1.1

Table 4-1. Initialization Parameters for web.xml using Struts 1.1

Name	Purpose/Default Value
config	A context-relative path to the default struts configuration file. The default value is <i>/WEB-INF/struts-config.xml</i> , which serves as the default application.

config/sub1	For additional sub-applications, you can specify them by using the value <i>config/</i> and the prefix of the sub-application. In this example, the <i>init-param</i> name would be <i>config/sub1</i> and the value might be <i>WEB-INF/struts-sub1-config.xml</i> . This tells the controller to load the sub-application sub1 from using the additional Struts configuration file. You can declare as many sub-applications as you need.
debug	The debugging detail level for this servlet, which controls how much information is logged. This parameter is optional and defaults to 0 if not specified, which is the lowest amount of logging information possible.
detail	The debugging detail level for the Digester, which logs information as it parses the configuration files. This parameter is optional and defaults to 0 if not specified, which is the lowest amount of logging information possible.
validating	A boolean value indicating whether to use a validating XML parser to process the configuration file, which is strongly recommended. This ensures that problems with the configuration files will be detected and reported immediately. This parameter is optional and defaults to true if not specified.

If you are supporting a 1.0 Struts application using the 1.1 release, the *web.xml* may still contain many of the configuration parameters that are now defined in the Struts configuration file. The parameters apply only to the default application and will eventually be removed in future releases.

Configuring the Tag Libraries

The Struts framework provides several JSP tag libraries that must be configured in the web application deployment descriptor if you choose to use them. You inform the container of these tag libraries by declaring one or more *taglib* elements within the web deployment descriptor. The following partial *web.xml* file illustrates how the tag libraries are configured.

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

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

  <taglib>
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
```



```
<taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
</taglib>
</web-app>
```

The *taglib* element has two sub-elements, the *taglib-uri* and *taglib-location*. The *taglib-uri* element specifies a URI identifying a tag library that is used by the web application. The value may either be a relative or an absolute URI. It must be a valid URI, but here it's used as a unique identifier for the tag library. The *taglib-location* element specifies the location (as a resource) of the tag library descriptor file.

The Struts tag libraries are not the only ones that may be declared in the web application deployment descriptor. If you create any of your own custom tag libraries, you should create *taglib* elements for them here as well. Although the JSP specification allows for both explicit and implicit mappings, it's best to list the tag libraries that are used within a web application in the *web.xml* file so that all parties to the development of the web application know exactly what the dependencies are.

Setting up the Welcome File List

The *welcome-file-list* element allows you to configure default resources that should be used when a valid, but partial URI is entered for a web application. You can specify multiple welcome files and they will be used in the order that they are configured.

To understand this better, suppose we configured the *welcome-file-list* element for the storefront application as in example 4-1.

Example 4-1. A partial web.xml file showing the welcome files for the Storefront application

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

The *welcome-file-list* shown in example 4-1 indicates that a request to the server for <http://www.somehost.com/storefront>, which is the root of the storefront application, should resolve to <http://www.somehost.com/storefront/index.jsp>. This is very beneficial because most containers would by default look for *index.html* or *index.htm* instead. You can specify multiple *welcome-file* elements within the *welcome-file-list*. This might be helpful if for example, you deployed your application on various types of containers and the first *welcome-file* resource was not found on the server. The container would continue to try to match the welcome files up to the request URI until it found one on the server and served that resource to the client. The order of the welcome file entries in the deployment descriptor is used for the matching process.

There should be no trailing or leading “/” characters in the *welcome-file* element. If there are no welcome files configured for the web application or the URI entered by a client, the web container may handle the request in a manner that is appropriate. This may mean returning a 404 response or a directory listing. It's a good idea to configure a welcome file for at least the root web application.

Using a Struts Action in the welcome-file-list

Because the web containers don't use the servlet mappings for resources in the welcome-file-list, you can't directly setup a `<welcome-file>` element to use a Struts action. However, there is an alternate way that will allow you to meet the same goal.

You need to create a global forward in the Struts configuration file for the action that you would like to invoke:

```
<global-forwards>
  <forward name="welcome" path="viewsignin.do"/>
</global-forwards>
```

Then create a JSP page called *welcome.jsp* (the name can actually be anything that you want) and use the Struts forward tag to forward to the global forward when the page is accessed.

```
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<logic:forward name="welcome"/>
```

Those are the only two lines that you need to put in the *welcome.jsp* file. You will then need to add a `<welcome-file>` element for the *welcome.jsp* page:

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

When the container uses the *welcome.jsp* resource, it will automatically forward to the global forward named “welcome”. This in turn invokes the *viewsignin.do* action and achieves the desired result.

Configuring Error Handling in *web.xml*

Although the Struts framework provides a suitable error handling mechanism, there are times that problems can slip through the exception-handling crack, and users are shown a servlet or JSP exception. To absolutely prevent this from happening, you should use the *error-page* element available to the web application deployment descriptor. Example 4-2 shows a partial *web.xml* file that utilizes the error-page element to prevent users from seeing a 404 or a 500 error.

Example 4-2. Using the error-page element will keep users from seeing unhelpful error pages

```
<web-app>
```

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

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

When an error status code is set in the response, the container will consult the list of error page declarations for the web application. If there is a match found, the container will return the resource indicated by the location element. The value of the *location* element must begin with a “/” character. It must also refer to a resource within the web application. If you need to refer to a resource outside of the web application, you can use the HTML Refresh META TAG. In the location element, refer to a static HTML document that only contains the following line:

```
<meta http-equiv="Refresh" content="0;URL=http://www.somehost.com/404.jsp">
```

When the error occurs, the Refresh META tag will reload immediately, but it will use the alternate URL provided. This strategy is also a good way to allow users to refer to resources with a static extension such as *.htm*, but then reload to a dynamic page such as a JSP page.

Servlet can also generate exceptions, which you are able to declare an error page for. Instead of specifying the *error-code* element, you can specify a fully qualified Java class using the *exception-type* element. Since servlets can throw the following exceptions during processing:

- *RuntimeException* or *Error*
- *ServletException* or subclasses
- *IOException* or subclasses

The Java exception class declared in the *exception-type* entry must be one of these types of throwable's.

Example 4-3 illustrates how you would substitute the *exception-type* element for the *error-code*.

Example 4-3. Using the exception-type element instead of the error-code

```
<web-app>
  <error-page>
    <exception-type>javax.servlet.ServletException</exception-type>
    <location>/common/system_error.jsp</location>
  </error-page>
</web-app>
```

For the majority of this chapter, you have been shown partial deployment descriptors. This was done mainly for space, but also so that we could ease our way into the various supported elements. It's finally time to include a complete example of a web deployment descriptor. Example 4-4 shows the web deployment descriptor for the storefront application.

Example 4-4. A complete web.xml configured for Struts 1.1

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>storefront</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>
    <init-param>
      <param-name>validating</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>storefront</servlet-name>
    <url-pattern>/action/*</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <error-page>
    <error-code>404</error-code>
    <location>/common/404.jsp</location>
  </error-page>
  <error-page>
    <error-code>500</error-code>
    <location>/common/500.jsp</location>
  </error-page>
</web-app>
```

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

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

<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
</taglib>
</web-app>
```

The Struts Configuration File

The Struts framework depends on one or more configuration files to be able to load and create the necessary application specific components at startup. The configuration files allow the behavior of the framework components to be specified declaratively, rather than have the information and behavior hard-coded. This gives developers the flexibility to provide their own extensions, which can be discovered dynamically by the framework.

The configuration file is based on the XML format and can be validated against the Struts DTD *struts-config_1_1.dtd*. Although there are some similarities between the 1.0 and 1.1 versions of the framework with respect to the configuration file, there are as many differences. Fortunately, the designers of the framework have made backwards compatibility a goal of the 1.1 release, therefore your 1.0.X applications should continue to function properly with the new version.

Configuring Multiple Sub-Applications still

Even though the overview that was given in Chapter 3 mentioned multiple sub-applications briefly, we haven't fully introduced the new feature that the Struts 1.1 release has added to the framework. With sub-applications, you have the ability to define multiple Struts configuration files, one for each sub-application that is supported. Each sub-application can provide its own configuration information, including message resources and be completely independent from other sub-applications.

Sub-applications or multi-applications as some developers are referring to them, allow a single Struts application to be split into separate projects, thus better supporting parallel development.

Although the functionality for multiple sub-applications exists in the framework, you are not required to implement more than one, the default application. We'll discuss sub-applications later in the chapter and much more in Chapters 5, 6, and 7. For now, we are going to concentrate on configuring the default application and then we'll show just how easy it is to add multiple sub-applications.

The *org.apache.struts.config* Package

The `org.apache.struts.config` package was added to version 1.1. The framework uses these JavaBeans to hold onto in memory, the configuration information it reads from the Struts configuration files. Figure 4-4 shows the main classes from the *config* package.

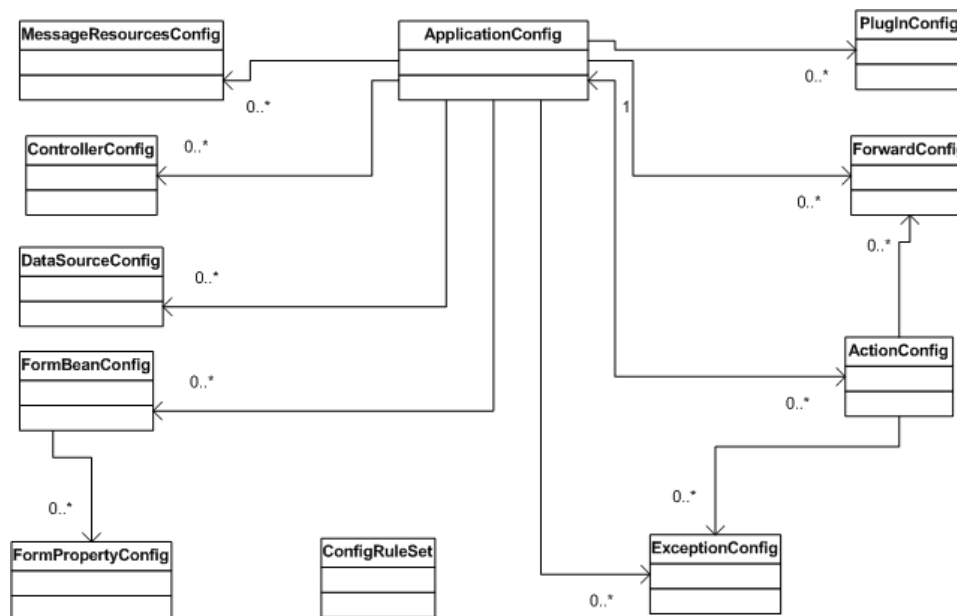


Figure 4-4. The class diagram of the *org.apache.struts.config* package

Each JavaBean class in the *config* package holds onto information from a specific section of the configuration file. After the configuration file has been validated and parsed, the Struts framework uses instances of these JavaBeans, to represent in-memory versions of the information that has been declared in configuration file. The JavaBean classes act as runtime holders of the configuration information and are used by the framework components when they need to determine what has been configured.

The `org.apache.struts.config.ConfigRuleSet` class shown in Figure 4-1 has a slightly different, but related job. It contains the set of Digester rules that are required to parse a Struts configuration file. Its job is to construct instances of the configuration JavaBeans when the application is started.

The `ApplicationConfig` class

A special introduction should be made for the `org.apache.struts.config.ApplicationConfig` class, as it plays a very important role in the framework. As Figure 4-4 indicates, it is central to the entire `config` package and holds onto the configuration information that describes an entire Struts application. If multiple sub-applications are being used, then there is one `ApplicationConfig` object for each sub-application. The `ApplicationConfiguration` class will surface throughout our discussion the framework.

The Struts Configuration DTD

As the web application's DTD is used to validate the `web.xml` file, the Struts DTD is used to validate the Struts configuration file. The following Struts DTD declaration indicates that the `struts-config` element is the root element for the XML file and that it has seven child elements.

```
<!ELEMENT struts-config (data-sources?, form-beans?, global-exceptions?, global-  
forwards?, action-mappings?, controller?, message-resources*, plug-in*)>
```

The `data-sources` Element

The `data-sources` element allows you to setup a rudimentary data source that you can utilize from within the Struts framework. A data source acts as a factory for database connections. It provides a single point of control for handing out connections and many data source implementations utilize a connection pooling mechanism to improve performance and scalability.

There are many vendors that provide their own implementation of a data source object. The Java language provides the `javax.sql.DataSource` interface that all implementations must implement. Most popular application servers provide built-in data source components and you will also find that all of the major database vendors provide a data source implementation as well.

The `data-sources` element can contain zero or more `data-source` elements as shown here.

```
<!ELEMENT data-sources (data-source*)>
```

The `data-source` element allows for multiple set-property elements to be specified, which allows you to configure properties that are specific to your data source implementation.

```
<!ELEMENT data-source (set-property*)>
```

Throughout the discussion of the Struts configuration elements in the rest of this chapter, you will notice a child element called *set-property* in many of the major elements of the configuration file. The *set-property* element specifies a name and value of an additional JavaBeans configuration property whose setter method will be called on the object that represents the surrounding element. Using this element is especially useful for passing additional property information to an extended implementation class. The *set-property* element is never required and you will only use it if you need to pass properties to a configuration class that does already an attribute predefined.

The set-property element defines 3 attributes, of which only two are required. The *property* attribute is the name of the JavaBeans property whose setter method will be called. The value attribute is a string representing the value that will be passed to the setter method after proper conversion. The next section on the *data-sources* element will provide an example of using the *set-property* element. The same format will be replicated wherever the set-property element is declared.

The attributes for the data-source element are illustrated in table 4-2.

Table 4-2. The attributes for the data-source element

Name	Description
id	Not currently used.
className	The implementation class of the configuration bean that will hold the data source information. If specified, it must be a descendant of <code>org.apache.struts.config.DataSourceConfig</code> , which is the default class when no value is specified. This attribute is optional.
key	The servlet context attribute under which this data source will be stored. The attribute is optional and the default value is <code>Action.DATA_SOURCE_KEY</code> .
type	A fully qualified Java class name of the data source implementation class. The class represented by this value must implement <code>javax.sql.DataSource</code> and be configurable from JavaBeans properties. This attribute is optional and the default value is <code>org.apache.struts.util.GenericDataSource</code> .

Example 4-5 illustrates how you can configure a data source within the Struts configuration file.

Example 4-5. Specifying a data source that can be used within the Struts framework

```
<data-sources>
  <data-source>
    <set-property property="autoCommit" value="true"/>
```



```
<set-property property="description" value="MySQL Data Source"/>
<set-property property="driverClass" value="com.caucho.jdbc.mysql.Driver"/>
<set-property property="maxCount" value="10"/>
<set-property property="minCount" value="2"/>
<set-property property="user" value="admin"/>
<set-property property="password" value="admin"/>
<set-property property="url"
    value="jdbc:mysql-caucho://localhost:3306/storefront"/>
</data-source>
</data-sources>
```

Example 4-5 illustrates a *data-source* element configured to connect to a *MySQL* database using a JDBC driver from *Caucho Technology*, which are the developers of the Resin™ servlet container. The Struts framework provides a default data source implementation with the class `org.apache.struts.util.GenericDataSource`. As Table 4-2 shows, you can use your own implementation or one from a third-party by providing the class name in the *type* attribute. In Example 4-5, the default data source implementation from Struts is being used.

You can specify multiple data sources within the configuration file and assign each one a unique key and access a particular data source in the framework by its key. This gives you the ability to access multiple databases if your requirements call for it.

Although the data source functionality provided by the framework does work, your application may require a more robust data source implementation. There are several other popular data source implementations that you can use. Table 4-3 lists several of the more popular alternative implementations that other Struts developers have had success using.

Table 4-3. Alternative data source implementations

Name	Vendor	URL
Poolman	Open Source	http://sourceforge.net/projects/poolman/
Espresso	Jcorporate	http://www.jcorporate.com/
JDBC Pool	Open Source	http://www.bitmechanic.com/projects/jdbcpool

The creator of the *Poolman* open source libraries is not supporting it anymore. Although it's still available on *SourceForge* and works quite well, it has not been updated for quite some time. Of course, since it's open source, you can make necessary fixes and changes if you need to.

The *form-beans* Element

The *form-beans* element allows you to configure multiple `ActionForm` classes that are used by the views. Within the form-beans section, you can configure zero or more *form-bean* child elements. Each *form-bean* element also has several child elements of itself.

```
<!ELEMENT form-bean (icon?, display-name?, description?, set-property*, form-property*)>
```

Each form-bean element also has five attributes that you can specify. Two of the attributes are required and the rest are optional. Table 4-4 lists the attributes.

Table 4-4. The attributes of the form-bean element

Name	Description
id	Not currently used.
className	If you don't want to use the standard configuration bean <code>org.apache.struts.config.FormBeanConfig</code> , you can specify your own class here. It must extend the <code>FormBeanConfig</code> class. This attribute is optional and the framework will use an instance of the <code>FormBeanConfig</code> class if not specified.
dynamic	If the class identified by the type attribute is an instance of <code>org.apache.struts.action.DynaActionForm</code> , then this value should be set to true. Otherwise this value is false. This attribute is optional and the default value is false.
name	A unique identifier for this bean, which is used to reference it throughout the framework. This value is required and must be unique within a sub-application.
type	The fully qualified name of a Java class that extends the Struts <code>ActionForm</code> class. If this value is specified as <code>org.apache.struts.action.DynaActionForm</code> , then Struts will dynamically generate an instance of the <code>DynaActionForm</code> , which also extends <code>ActionForm</code> . This attribute is required.

As you learned from Chapter 3, a form bean is a JavaBean class that extends the `org.apache.struts.action.ActionForm` class. Example 4-6 shows how the *form-beans* element can be configured in the configuration file.

Example 4-6. A partial struts-config.xml file showing the form-beans section

```
<struts-config>
  <form-beans>
    <form-bean
      name="loginForm"
      dynamic="true"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="username" type="java.lang.String"/>
      <form-property name="password" type="java.lang.String"/>
    </form-bean>

    <form-bean
      name="shoppingCartForm"
      type="com.oreilly.struts.order.ShoppingCartForm"/>
  </form-beans>
```

| </struts-config> |

One of the `<form-bean>` elements in Example 4-6 uses a new feature to Struts 1.1 called dynamic action forms. The purpose of dynamic action forms was discussed in Chapter 3.

You can pass one or more dynamic properties to an instance of the `org.apache.struts.action.DynaActionForm` class using the `<form-property>` element. It's only supported when the `type` attribute of the surrounding `<form-bean>` element is "org.apache.struts.action.DynaActionForm", or descendant.

Each form-bean element also has five attributes that you can specify. Two of the attributes are required and the rest are optional. Table 4-4 lists the attributes.

Table 4-4. The attributes of the form-bean element

Name	Description
className	If you don't want to use the standard configuration bean <code>org.apache.struts.config.FormPropertyConfig</code> , you can specify your own class here. This attribute is not required.
initial	A string representation of the initial value for this property. If not specified, primitives will be initialized to zero and objects to null. This attribute is not required.
name	The JavaBeans property name of the property being described by this element. This attribute is required.
type	The fully qualified Java class name of the implementation class of this bean property, optionally followed by "[]" to indicate that this property is indexed. This attribute is required.

This is a very nice feature in 1.1 because it saves you development time from having to create a bunch of `ActionForm` classes that just pass data from the HTML pages to the backend. The `DynaActionForm` extends the `ActionForm` class so you will need to perform an explicit cast on the `ActionForm` object within the execute method of your Action class to retrieve the properties.

Since the `DynaActionForm` instance is created dynamically, the `validate()` method doesn't provide any default validation behavior, so you may have to perform the validation in your `Action` class. The `reset()` method will still be called and the values will be reset back to the default state. Action forms will be discussed further in Chapter 7.

The *global-exceptions* Element

The *global-exceptions* section allows you to configure exception handlers declaratively at the application level. The *global-exceptions* element can contain zero or more *exception* elements.

```
| <!ELEMENT global-exceptions (exception*)>
```

As you'll see later in this chapter when we talk about the action mappings, *<exception>* elements can also be specified at the action level. If an *<exception>* element is configured for the same type of exception both at a global level and at an action level, the action level will take precedence. If no exception element mapping is found at the action or application level, the framework will then look for exception mappings defined for the exception's parent class. Eventually, if a handler is not found, a *ServletException* or *IOException* will be thrown, depending on the type of the original exception. Chapter 10 deals with both declarative and programmatic exception handling in detail. This section will illustrate how to configure declarative exception handling for your applications.

The *<exception>* element describes a mapping between a Java exception that may occur during processing of a request and an instance of *org.apache.struts.action.ExceptionHandler* that is responsible for dealing with the thrown exception. The declaration of the *<exception>* element illustrates that it also has several child elements.

```
| <!ELEMENT exception (icon? display-name? description? set-property*)>
```

Probably more important than the child elements are the attributes that can be specified in the *<exception>* element. The attributes are shown in table 4-5.

Table 4-5. The attributes of the exception element

Name	Description
id	Currently not used.
className	The implementation class of the configuration bean that will hold the exception information. If specified, it must be a descendant of <i>org.apache.struts.config.ExceptionConfig</i> , which is the default class when no value is specified.
handler	The fully qualified Java class name of the exception handler that will process the exception. If no value is specified, the default exception handling class <i>org.apache.struts.action.ExceptionHandler</i> will be used. If a class is specified for this attribute, it must be a descendant of the <i>ExceptionHandler</i> class.
key	A message key that is specified in the resource bundle for this sub-application. This value is used by the <i>ActionError</i> instance.

path	The application-relative path of the resource to forward to if this exception occurs. This attribute is optional and the framework will default to the input attribute for the action mapping if no value is specified here.
scope	Identifier of the scope level, where the <code>ActionError</code> instance should be stored. The attribute value must either be “request” or “session”. This attribute is optional and will default to “request” scope if not specified.
type	The fully qualified Java class name of the exception that is to be handled. This attribute is required because it identifies the exception, which can't be assumed by the framework.

Example 4-7 shows a `<global-exception>` element.

Example 4-7. Configuring the global-exceptions element

```
<global-exceptions>
  <exception
    key="global.error.invalidlogin"
    path="/security/signin.jsp"
    scope="request"
    type="com.oreilly.struts.framework.exceptions.InvalidLoginException"/>
</global-exceptions>
```

Currently, there's no way to specify which bundle the key attribute in the `<exception>` element should come from. The only choice is the default bundle. There is however, an enhancement request entered for this feature to be added.

The `<global-forwards>` Element

The `<global-forwards>` section allows you to configure application level mappings of logical names to application relative URI paths. The `<global-forwards>` section consists of zero or more `<forward>` elements as shown here.

```
| <!ELEMENT global-forwards (forward*)>
```

The `<forward>` element maps a logical name to an application relative URI path. The application can then perform a forward or redirect, using the logical name rather than the literal URI. This helps to decouple the controller and model logic from the view. The `<forward>` element can be defined both at the global level and also at the action level. If a forward with the same name is defined at both places, the action level will take precedence.

The declaration of the `<forward>` element illustrates that it also has several child elements.

```
| <!ELEMENT forward(icon?, display-name?, description, set-property*)>
```

As with the exception element, the attributes are probably more interesting than the child elements. The attributes for the `<forward>` element are shown in table 4-6.

Table 4-6. The attributes of the forward element

Name	Description
className	The implementation class of the configuration bean that will hold the forward information. If specified, it must be a descendant of <code>org.apache.struts.config.ForwardConfig</code> . The class <code>org.apache.struts.action.ActionForward</code> is the default class when no value is specified. This attribute is not required.
contextRelative	Set to true to indicate that the resource specified in the path attribute should be interpreted as application relative if the path starts with a “/” character. This is so the resource specified by the path attribute can reside in another sub-application. This attribute is not required and the default value is false.
name	A unique value that is used to reference this forward in the application. This attribute is required.
path	An application relative or content relative URI to which control should be forwarded or redirected to. This attribute is required.
redirect	A boolean value that determines whether the <code>RequestProcessor</code> should perform a forward or a redirect when using this forward mapping. This attribute is not required and the default value is false, which means that a forward will be performed.

The *contextRelative* attribute for the forward element is very important. If the path attribute starts with a “/” and doesn’t provide a value for this attribute or specifies it as false, the controller will append the sub-application prefix onto the path when it performs the actual forward through the `RequestDispatcher`. This may cause a 404 error if by adding the sub-application prefix causes the resource not to be found.

The `org.apache.struts.action.ActionForward` class is used to hold the information configured in the controller element. The `ActionForward` class now extends `org.apache.struts.config.ForwardConfig` for backwards compatibility, but it has not yet been deprecated.

The *action-mappings* Element

The `<action-mappings>` element configures the mappings from submitted request paths to the corresponding `Action` classes for a particular sub-application. The `<action-mappings>` element can contain zero or more `<action>` elements.

```
| <!ELEMENT action-mappings (action*)> |
```

The `<action>` element describes a mapping from a specific request path to a corresponding `Action` class. The controller selects a particular mapping by matching the URI path in the request with the *path* attribute for an action mapping.

The action element contains the following child elements.

```
<!ELEMENT action (icon?, display-name?, description, set-property*, exception*, forward*)>
```

There are two child elements that may stand out in the list of children for the `<action>` element. They should stand out because you've already seen both of them earlier in the chapter. They are the `<exception>` element and the `<forward>` element, both of which you have seen before.

You saw the `exception` element earlier in this chapter when we discussed the `<global-exceptions>` element. We mentioned then that exception elements could be defined at the global, or at the action level. The exception elements defined here within the action element take precedence over the ones defined at the global level. The syntax and attributes are the same when defined here as they are in the `global-exceptions` element.

You've also seen the `forward` element earlier in reference to the `global-forwards` element. As with the exceptions, a forward element can be defined both at a global level and at the action level. The action level will take precedence if the same forward is defined in both locations.

The `<action>` element contains quite a few attributes. They are shown in table 4-7.

Table 4-7. The attributes for the action element

Name	Description
className	The implementation class of the configuration bean that will hold the action information. If specified, it must be a descendant of <code>org.apache.struts.config.ActionConfig</code> class. The <code>org.apache.struts.action.ActionMapping</code> class is the default class when no value is specified. This attribute is optional.
attribute	The name of the request or session scope attribute under which the form bean for this action can be accessed. A value is only allowed here if there is a form bean specified in the <code>name</code> attribute. This attribute is optional and has no default value.
forward	Application-relative path of the servlet or JSP resource that will process this request, instead of instantiating and calling the <code>Action</code> class specified by the <code>type</code> attribute. The attributes <code>forward</code> , <code>include</code> , and <code>type</code> are mutually exclusive and only one can be specified for an action. This attribute is optional.
include	Application -relative path of the servlet or JSP resource that will process this request, instead of instantiating and calling the <code>Action</code> class specified by the <code>type</code> attribute. The attributes <code>forward</code> , <code>include</code> , or <code>type</code> are mutually exclusive and only one can be specified for an action. This attribute is optional.

name	The name of the form bean, if any, that is associated with this action. This value must be the <i>name</i> attribute from one of the form-bean elements defined earlier. This attribute is optional and has no default value.
path	The application-relative path of the submitted request, starting with a “/” character and without the filename extension if extension mapping is used. In other words, this is the name of the action. For example, “/addToShoppingCart” is an example of a path for an action. This value is required. This attribute probably should have been called “name” because it really is the name of the action.
parameter	General purpose configuration parameter that can be used to pass extra information to the action instance selected by this action mapping. The core framework does not use this value in any way. If you provide a value here, you can obtain it in your <i>Action</i> subclass by calling the <i>getParameter()</i> method on the mapping passed to the <i>execute()</i> method.
prefix	The prefix used to match request parameter names to form bean property names. You can only provide a value here if the <i>name</i> attribute is specified.
scope	This attribute is used to identify which scope level the form bean is placed. It can either be “request” or “session”. It can only be specified if the <i>name</i> attribute is present. The default value is “session”.
suffix	The suffix is used to match request parameter names to form bean property names. You can only provide a value here if the <i>name</i> attribute is specified.
type	A fully qualified Java class name that extends <i>org.apache.struts.action.Action</i> class, which is used to process the request if the <i>forward</i> or <i>include</i> attribute is not specified. Only one of these three attributes can be specified for a particular action mapping.
unknown	A boolean value indicating if this action should be configured as the default for this application. If this attribute is set to true, this action will handle any request that is not handled by another action. Only one action per application can have this value set to true. This attribute is optional and defaults to false. This is a good place to setup a default action that will catch any invalid action URL entered by the user.
validate	A boolean value indicating whether the <i>validate()</i> method of the form bean, specified by the name attribute, should be called prior to calling the <i>execute()</i> method of this action. This attribute is optional and will default to true.

The *org.apache.struts.action.ActionMapping* class is used to hold the information configured in the controller element. The *ActionMapping* class now extends

`org.apache.struts.config.ActionConfig` for backwards compatibility, but has not yet been deprecated.

The *controller* Element

The `<controller>` element is new to version 1.1. Prior to version 1.1, the `ActionServlet` contained the controller functionality and you had to extend that class to override the functionality. In version 1.1 however, Struts has moved most of the controller functionality to the new `RequestProcessor` class. The `ActionServlet` still receives the requests, but then delegates the request handling to an instance of the `RequestProcessor` class that has been installed. This allows you to declaratively assign the processor class and modify its functionality.

If you're familiar with version 1.0, you'll notice that many of the parameters that were configured in the `web.xml` for the controller servlet are now configured for the `<controller>` element here. Since the controller and its attributes are defined in the `struts-config.xml`, you can define a separate `<controller>` element for each sub-application. The declaration of the `<controller>` element illustrates that it has a single child element.

```
| <!ELEMENT controller (set-property*)> |
```

The `<controller>` element can contain zero or more `<set-property>` elements and many different attributes. The attributes are shown in table 4-8.

Table 4-8. The attributes for the controller element

Name	Description
id	Currently not used.
className	The implementation class of the configuration bean that will hold the controller information. If specified, it must be a descendant of <code>org.apache.struts.config.ControllerConfig</code> , which is the default class when no value is specified. This attribute is not required.
bufferSize	The size of the input buffer used when processing file uploads. This attribute is optional and the default value is 4096.
contentType	The default content type and optional character encoding that gets set for each response. This attribute is not required and the default value is "text/html". Even when a value is specified here, an action or a JSP page may override it.
debug	The debugging level for this application. The value is used throughout the framework to determine how verbose the logging information should be for events that take place internally. The larger the value, the more verbose the logging is. This attribute is not required and the default value is 0, which causes little or no logging information to be written out.

locale	A boolean value indicating whether the user's preferred Locale is stored in the user's session if not already present. This attribute is not required and the default value is true.
maxFileSize	The maximum size (in bytes) of a file to be accepted as a file upload. This value can be expressed as a number followed by a "K", "M", or "G", which is interpreted to mean kilobytes, megabytes, or gigabytes respectively. This attribute is not required and the default value is "250M".
multipartClass	The fully qualified Java class name of the multipart request handler class to be used. This attribute is not required and the default value is <code>org.apache.struts.upload.DiskMultipartRequestHandler</code> .
nocache	A boolean value indicated that the framework should set nocache HTTP headers in every response. This attribute is not required and the default value is false.
processorClass	The fully qualified Java class name of the request processor class to be used to process requests. The value specified here should be a descendant of <code>org.apache.struts.action.RequestProcessor</code> . This attribute is not required and the default value is <code>org.apache.struts.action.RequestProcessor</code> .
tempDir	A temporary working directory that is used when processing file uploads. This attribute is not required and the servlet container will assign a default value for each web application.

The `org.apache.struts.config.ControllerConfig` class is used to hold the information configured in the controller element. The following fragment shows an example of how to configure the `<controller>` element.

```
<controller
  contentType="text/html; charset=UTF-8"
  debug="3"
  locale="true"
  nocache="true"
  processorClass="com.oreilly.struts.framework.CustomRequestProcessor"/>
```

The `message-resources` Element

The `<message-resources>` element specifies characteristics of the message resource bundles that contain the localized messages for an application. Each Struts configuration file can define one or more message resource bundles. Each sub-application can define its own bundles. The declaration of the `<message-resources>` element shows that it contains no child elements.

```
<!ELEMENT message-resources EMPTY>
```

The entire configuration for this element is done through the attributes as shown in table 4-9.

Table 4-9. The attributes for the message-resources element

Name	Description
id	Currently not used.
className	The implementation class of the configuration bean that will hold the message-resources information. If specified, it must be a descendant of <code>org.apache.struts.config.MessageResourcesConfig</code> , which is the default class when no value is specified. This attribute is optional.
factory	The fully qualified Java class name of the <code>MessageResourcesFactory</code> class that should be used. This attribute is optional. The default value is <code>org.apache.struts.util.PropertyMessageResourcesFactory</code> .
key	The servlet context attribute key that this message resource bundle will be stored under. This attribute is optional. The default value is specified by the string constant <code>Action.MESSAGES_KEY</code> .
null	A boolean value indicating how the <code>MessageResources</code> subclass should handle the case when a unknown message key is used. If this value is set to true, a null string will be returned. If set to false, a message that looks something like this <code>“???global.label.missing???”</code> will be returned. The actual message will contain the bad key. This attribute is optional. The default value is true.
parameter	This attribute is the base name of the resource bundle. For example, if the name of your resource bundle is <code>ApplicationResources.properties</code> , you should set the parameter value to <code>ApplicationResources</code> . A better name could have probably been found for this attribute. This attribute is required.

The following example shows how to configure multiple `<message-resources>` elements for a single application. Notice that the second element had to specify the key attribute, since there can be only one stored with the default key.

```
<message-resources parameter="StorefrontMessageResources" null="false"/>
<message-resources
  key="IMAGE_RESOURCE_KEY"
  parameter="StorefrontImageResources"
  null="false"/>
```

The *plug-in* Element

The *plug-in* element specifies a fully qualified class name of a general-purpose application plug-in module that receives notification of application startup and shutdown events. An instance of the specified class is created for each element and the `init` method is called when the application is started and the `destroy` method when the application is stopped. The class specified here must implement the `org.apache.struts.action.PlugIn` interface and implement the `init` and `destroy` methods.

This behavior is very similar to what servlet filters and/or application event listeners can provide based on the Servlet 2.3 specification. However, the difference is that the class doesn't have to be coupled to a servlet container.

The declaration of the *plug-in* element shows that it may contain zero or more *set-property* elements, so that extra configuration information may be passed to your class.

```
| <!ELEMENT plug-in (set-property*)>
```

The allowed attributes for the *<plug-in>* element are shown in table 4-10.

Table 4-9. The attributes for the plug-in element

Name	Description
id	Currently not supported.
className	The fully qualified Java class name of the plug-in class. It must implement the <code>PlugIn</code> interface.

The following fragment shows how the plug-in element can be configured in the Struts configuration file.

```
| <plug-in className="com.oreilly.struts.framework.database.DatabasePlugIn">  
|   <set-property property="debug" value="true"/>  
|   <set-property property="pathname" value="/WEB-INF/database.properties"/>  
| </plug-in>
```

Up to this point, you haven't seen a full example of a Struts configuration file. Example 4-8 provides a complete listing.

Example 4-8. A complete Struts configuration file

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE struts-config PUBLIC  
    "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"  
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">  
  
<struts-config>  
  
  <!--  
  <data-sources>  
    <data-source>  
      <set-property property="autoCommit" value="true"/>  
      <set-property property="description" value="Resin Data Source"/>  
      <set-property property="driverClass" value="com.caucho.jdbc.mysql.Driver"/>  
      <set-property property="maxCount" value="10"/>  
      <set-property property="minCount" value="2"/>  
      <set-property property="user" value="admin"/>  
      <set-property property="password" value="admin"/>  
      <set-property property="url" value="jdbc:mysql-  
caucho://localhost:3306/storefront"/>  
    </data-source>  
  </data-sources>  
  -->
```

```
<form-beans>
  <form-bean
    name="loginForm"
    type="com.oreilly.struts.storefront.security.LoginForm" />
  <form-bean
    name="itemDetailForm"
    dynamic="true"
    type="org.apache.struts.action.DynaActionForm">
    <form-property name="view" type="com.oreilly.struts.catalog.view.ItemView" />
  </form-bean>
</form-beans>

<global-exceptions>
  <exception
    key="global.error.invalidlogin"
    path="/security/signin.jsp"
    scope="request"
    type="com.oreilly.struts.framework.exceptions.InvalidLoginException" />
</global-exceptions>

<global-forwards>
  <forward name="Login" path="/security/signin.jsp" redirect="true" />
  <forward name="SystemFailure" path="/common/systemerror.jsp" />
  <forward
    name="SessionTimeOut"
    path="/common/sessiontimeout.jsp"
    redirect="true" />
</global-forwards>

<action-mappings>
  <action
    path="/viewsignin"
    parameter="/security/signin.jsp"
    type="org.apache.struts.actions.ForwardAction"
    scope="request"
    name="loginForm"
    validate="false"
    input="/index.jsp">
  </action>
  <action
    path="/signin"
    type="com.oreilly.struts.storefront.security.LoginAction"
    scope="request"
    name="loginForm"
    validate="true"
    input="/security/signin.jsp">
    <forward name="Success" path="/index.jsp" redirect="true" />
    <forward name="Failure" path="/security/signin.jsp" redirect="true" />
  </action>
  <action
    path="/signoff"
    type="com.oreilly.struts.storefront.security.LogoutAction"
    scope="request"
    validate="false"
    input="/security/signin.jsp">
```

```
<forward name="Success" path="/index.jsp" redirect="true"/>
</action>
<action
  path="/home"
  parameter="/index.jsp"
  type="org.apache.struts.actions.ForwardAction"
  scope="request"
  validate="false">
</action>
<action
  path="/viewcart"
  parameter="/order/shoppingcart.jsp"
  type="org.apache.struts.actions.ForwardAction"
  scope="request"
  validate="false">
</action>
<action path="/cart"
  type="com.oreilly.struts.storefront.order.ShoppingCartActions"
  scope="request"
  input="/order/shoppingcart.jsp"
  validate="false"
  parameter="method">
  <forward name="Success" path="/action/viewcart" redirect="true"/>
</action>
<action
  path="/viewitemdetail"
  name="itemDetailForm"
  input="/index.jsp"
  type="com.oreilly.struts.storefront.catalog.GetItemDetailAction"
  scope="request"
  validate="false">
  <forward name="Success" path="/catalog/itemdetail.jsp"/>
</action>

<action
  path="/begincheckout"
  input="/order/shoppingcart.jsp"
  type="com.oreilly.struts.storefront.order.CheckoutAction"
  scope="request"
  validate="false">
  <forward name="Success" path="/order/checkout.jsp"/>
</action>
<action
  path="/getorderhistory"
  input="/order/orderhistory.jsp"
  type="com.oreilly.struts.storefront.order.GetOrderHistoryAction"
  scope="request"
  validate="false">
  <forward name="Success" path="/order/orderhistory.jsp"/>
</action>
</action-mappings>

<controller
  contentType="text/html; charset=UTF-8"
  debug="3"
```

```
    locale="true"
    nocache="true"
    processorClass="com.oreilly.struts.framework.CustomRequestProcessor"/>

    <message-resources
        parameter="StorefrontMessageResources"
        null="false"/>
    <message-resources
        key="IMAGE_RESOURCE_KEY"
        parameter="StorefrontImageResources"
        null="false"/>
</struts-config>
```

Using Multiple Sub-Applications

Now that you've seen how to configure the default application for Struts, the last step is to discuss how you include multiple sub-applications. Fortunately, it's very easy. The steps necessary are to first create the additional Struts configuration files. Let's suppose that we created a second called *struts-order-config.xml*. The first step is to modify the *web.xml* file for the application and add an addition *init-param* for the sub-application. This was shown earlier in the chapter, but it's repeated here for convenience. Example 4-9 shows the servlet instance mapping from before with an addition *init-param* for the second Struts configuration file.

Example 4-9. A partial web.xml that illustrates how to configure multiple sub-applications

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

  <init-param>
    <param-name>config/order</param-name>
    <param-value>/WEB-INF/struts-order-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>
  <init-param>
    <param-name>validating</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
```

| `</servlet>` |

Notice that the *param-name* value for the non-default sub-application ex example 4-9 begins with *config/* as in *config/order*. All non-default sub-application *param-name* elements must begin with the *config/* value. The default application contains the *config* value alone. The part that comes after that is known as the sub-application prefix and is used throughout the framework for intercepting requests and returning the correct resources.

Fortunately, that is about all there is to configuring support for multiple sub-applications. The last item of interest when configuring sub-applications is that you should pay special attention to the configuration attributes available in the various Struts XML elements. Some of them, as mentioned in this chapter, have a profound effect on how an application operates in a multi-app environment.

Specifying a DOCTYPE Element

To ensure that your Struts configuration file is valid, it can and should be validated against the Struts DTD. To do this, you must include the *DOCTYPE* element at the beginning of your Struts configuration XML file, similar to the following fragment.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  etc...
</struts-config>
```

In earlier versions of the framework, there were some issues of applications not being able to startup if they weren't able to get to the Jakarta site and access the DTD from there. However, this is no longer true, since Struts provides local copies of the DTDs and registers them with the Digester.

Although this is not recommended, there are some rare circumstances where you might not want the framework to validate the Struts configuration file. You can configure Struts to bypass validation of the configuration file by adding an *init-param* to the *web.xml* file with the name *validating* and a value of false. If there's no *init-param* with the name *validating*, the default value is true and that's why the configuration file is validated by default.

Some users also prefer to specify a SYSTEM DOCTYPE tag, rather than a PUBLIC one. This allows you to specify an absolute path, over a relative one. Although this may solve a short-term problem, it creates more long-term ones. You can't always guarantee the directory structure from one target environment to another. Also, different containers seem to act differently when using a SYSTEM DOCTYPE tag. You are probably better

off not using it. However, if you decide that you need to do so, it should look something similar to the following:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config SYSTEM "file:///c:/dtds/struts-config_1_1.dtd">

<struts-config>
  etc...
</struts-config>
```

As you can see, the location of the DTDs is an absolute path. If the path of the target environment is not the same, you'll have to make modifications to the XML file. This is why this approach is not recommended.

Struts Console Tool

When developing on a small application, the Struts configuration is manageable. It's still XML, but typically it's small enough not to be much of a problem. Some developers use XML editors while others use ordinary text editors. Both of these are fine when the application is relatively small, but when you are working on a Struts project made up of many different developers, the size and complexity of the file is quite enormous.

This is one of the reasons that the Struts console application was created. The Struts Console was created by James Holmes and is a Java Swing application that provides an easy to use interface for editing the Struts configuration files.

[\[Editors – Because development I still going on for the console that supports Struts 1.1, I'm not able to complete this section yet. I will do so during author review. I have spoken with the creator of the tool and he will send me a beta copy when it's available so that I can make changes during author review. I will also need to provide some screen shots then because they are not currently available. Chuck\]](#)

You can download the Struts console for free, although it's not open source software. There is no license necessary, but you should check the web site to make sure this hasn't changed. You can download the Struts console from the following web site:

<http://www.jamesholmes.com/struts/console/>

You should be cautioned that the formatting of your Struts configuration file might be lost when saving the file using the Console. This is because it uses an XML parser to read in the configuration file and the parser can't maintain the complete knowledge of how the file was formatted. The Console however does a decent job of formatting the file itself and this really isn't a big problem.

Reloading the Configuration Files

The web deployment descriptor is loaded and parsed when the web container is first started. By default, changes to the descriptor are not detected and re-loaded while the container is running. In fact, not many containers even provide this functionality. This is typically not a wanted feature anyway due to the possible security issues involved.

The Struts configuration files are also loaded and parsed when the container is launched. Changes to these are also not automatically detected. The same security issues are present. Another problem occurs for any user currently using the system. The session information may be destroyed and the user will definitely get an ugly error the next time they try to perform an action.

Some developers and products still may required or desire the ability to reload the Struts configuration files without restarting the web container. If your application is one of them, there are ways that you can do it. There are two distinct approaches that you can try. One is where you create an actual Struts action that will re-initialize the `ActionServlet`. You would obviously want to put some form of protection around which set of users could call this action. Once the `ActionServlet` was re-initialized, everything would be new and the application would service requests, just like before.

A different and more active approach would be to create a `Thread` that monitored the *lastModifiedTime* of the configuration file. The `Thread` would sleep for a few seconds or minutes and when awoke would compare the *lastModifiedTime* of the file against one stored in a variable. If they were different, this means that the file has changed and it's time to reload the application. This approach is nice because you don't have to worry about an unwelcome user reloading your application. However, the time that it gets reloaded is entirely up to the `Thread`.

5

Struts Controller Components

As discussed in Chapter 1, the controller components are responsible for detecting user input, possibly updating the domain model, and selecting the next view for the client. The controller helps to separate the presentation of the model, from the actual model. This separation gives you much more freedom to develop different types of presentations based on the same domain model.

Using the concept of a controller allows provides a centralized point of control for the application where all client requests can be processed first. Because the controller acts as a mediator between the client input and the model, the controller is able to provide common functionality such as security, logging, and other important services on behalf of each client request.

Because all requests are filtered through the controller, the views are decoupled from the business operations and from other view components. It's entirely up to the controller, which view to return to the client. This might not seem like it adds much benefit, but it creates flexibility for the application that you would otherwise not have.

The Struts framework uses a servlet to process incoming requests, however it relies on many other components that are part of the controller domain, to help it carry out its responsibilities. The Struts controller components have been briefly mentioned in previous chapters, but it's time to take in-depth look at what components have responsibility for the controller functionality in the framework.

The Controller Mechanism

As the J2EE design pattern *Front Controller* describes using a single controller to funnel all client requests through a central point has many advantages. Services like security, internationalization, and logging can then be centralized in the controller, which applies these functions consistently for all requests. When the behavior of these services needs to be modified, the change only needs to be made to an isolated area of the application.

You learned in Chapter 1 that the Struts controller has several responsibilities. Chiefly among those are:

- Intercept requests from the clients.
- Translate and map each request to a business operation.
- Collect results from the business operation and make them available to the client.
- Determine the view to display to the client based on the current state and result of the business operation.

Within the Struts framework, there is not just one component that performs the controller duties, there are actually several. Figure 5-1 illustrates a simple class diagram of the components in the Struts framework that share some portion of that responsibility.

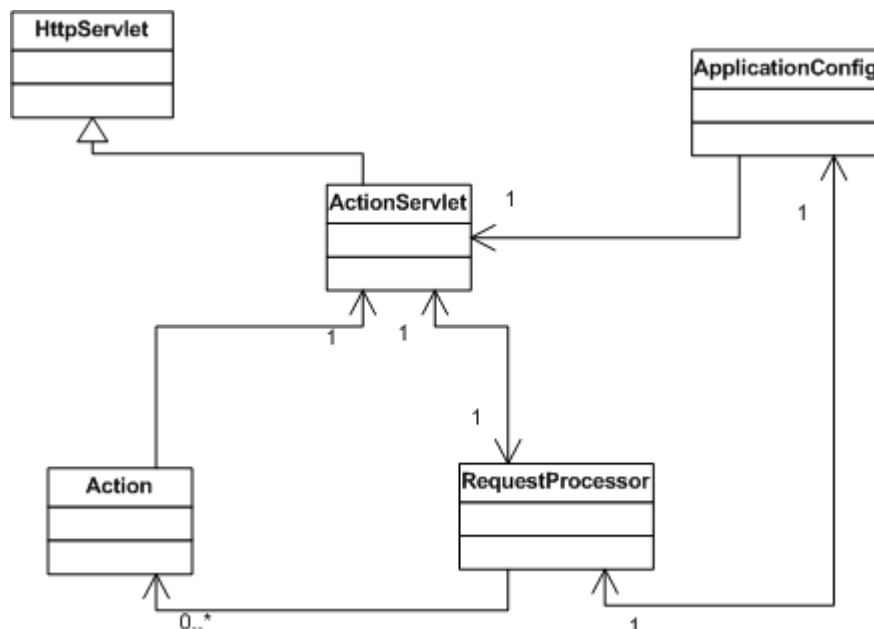


Figure 5-1. Several different components have controller responsibilities in the Struts framework

[\[Editors: This diagram is a place holder for a better one that will be dropped in later\]](#)

There are other secondary components that are used by these controller components to help fulfill their responsibilities. For now, let's focus on the ones in Figure 5-1. Most all of these classes have been introduced earlier in Chapters 2 and 3, but now will be discussed in depth.

The `ActionServlet` Class

The `org.apache.struts.action.ActionServlet` acts as the primary controller for the Struts framework. All requests from the client tier must pass through this component before proceeding anywhere else in the application.

When the `ActionServlet` receives an `HttpRequest`, either through the `doGet()` or `doPost()` method, the `process()` method is called to handle the request. The `process()` method of the `ActionServlet` is shown in Example 5-1.

Example 5-1. The `process()` method in `ActionServlet` processes every Struts request

```
protected void process(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    RequestUtils.selectApplication( request, getServletContext() );
    getApplicationConfig(request).getProcessor().process( request, response );
}
```

The `process()` method might not look very complicated, but the methods that are being invoked within it, surely are. First, the static `selectApplication()` method in the `org.apache.struts.util.RequestUtils` class is called and passed the current request and the `ServletContext` for the web application. The job of the `selectApplication()` method is to select a sub-application to handle the current request, by matching the path returned from the `request.getServletPath()` to the *prefix* of each sub-application that has been configured.

If you only use a single Struts configuration file, then you will only have a single application. This is known as the default application. To make processing requests for the default application and the sub applications simple and consistent, the default application is treated as just another sub application. Therefore, any requests that don't contain an application suffix will get routed and handled by the default application.

This `selectApplication()` method will store the appropriate `ApplicationConfig` and `MessageResources` objects into the request. This makes it easier for the rest of the framework to know which application and application components should be utilized for the request.

Extending the `ActionServlet` Class

Prior to version 1.1, the `ActionServlet` contained much of the code to process each user request. Starting with 1.1 however, most of that functionality has been moved to `org.apache.struts.action.RequestProcessor` class, which will be discussed in the next section. This new class has been added to help relieve the `ActionServlet` from most of the controller burden.

Although the framework still allows you to extend the `ActionServlet`, the benefit is not as great as with earlier versions, since most of the functionality lies in the new `RequestProcessor` class. If you would still like to use your own version, you just need to create a class that extends `ActionServlet` and configure the framework to use this class instead of the one from Struts. Example 5-2 shows a Java servlet that extends the Struts `ActionServlet` and overrides the `init()` method.

Example 5-2. The Struts `ActionServlet` can be extended to perform custom initialization

```
package com.oreilly.struts.storefront.framework;

import javax.servlet.ServletException;
import javax.servlet.UnavailableException;
import org.apache.struts.action.ActionServlet;
import com.oreilly.struts.storefront.service.IStorefrontService;
import com.oreilly.struts.storefront.service.StorefrontServiceImpl;
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;
/**
 * Extend the Struts ActionServlet to perform your own special
 * initialization.
 */
public class ExtendedActionServlet extends ActionServlet {

    public void init() throws ServletException {

        // Make sure to always call the super's init() first
        super.init();

        // Initialize the persistence service
        try{
            // Create an instance of the service interface
            StorefrontServiceImpl serviceImpl = new StorefrontServiceImpl();

            // Store the service into the application scope
            getServletContext().setAttribute( IConstants.SERVICE_INTERFACE_KEY,
                                                serviceImpl );
        }catch( DatastoreException ex ){
            // If there's a problem initializing the service, disable the web app
            ex.printStackTrace();
            throw new UnavailableException( ex.getMessage() );
        }
    }
}
```

Overriding the `init()` method was chosen just as an example. You can override any method that you need to. If you do override the `init()` method as Example 5-2 does, make sure that you call the super `init()` method, so that the default initialization occurs.

Don't worry about what the code in Example 5-2 is doing for now. The goal is to understand how to extend the `ActionServlet`.

To configure the framework to use your `ActionServlet`, instead of the default one in the Struts framework, you'll also need to modify the `web.xml` file as was shown in Chapter 4, and which is repeated here.

```
<servlet>
  <servlet-name>storefront</servlet-name>
  <servlet-class>
    com.oreilly.struts.storefront.framework.ExtendedActionServlet
  </servlet-class>
</servlet>
```

Struts Initialization Process

Depending on the initialization parameters configured in the `web.xml` file, the servlet container will either load the Struts `ActionServlet` when the container is first started, or when the first request arrives for the servlet. In either case, like any other Java servlet, the `init()` method is guaranteed to be called and must finish before any request is processed by the servlet. The Struts framework performs all of the compulsory initialization when this method is called. This section takes an inside look at what all goes on during that initialization process. Understanding these details will make debugging and extending your applications that much easier.

There are many steps that occur when the `init()` method of the Struts `ActionServlet` is invoked by the container.

1. Initialize the framework's internal message bundle. These messages are used to output informational, warning, and error messages to the log files. The `org.apache.struts.action.ActionResources` bundle is used to obtain the internal messages.
2. Load the initialization parameters from the `web.xml` file that control various behaviors of the `ActionServlet`. These parameters include `config`, `debug`, `detail`, and `validating`. For information on how these and other servlet parameters affect the behavior of an application, refer to "Declaring the Initialization Parameters" in Chapter 4.
3. Load and initialize the servlet name and servlet mapping information from the `web.xml` file. These values will be used throughout the framework, mostly by tag libraries to output correct URL destinations when submitting HTML forms. During this initialization, the DTDs that are used by the framework are also registered. The DTDs are used to validate the configuration file in the next step.

4. Load and initialize the Struts configuration data for the default application, which is specified by the `config` initialization parameter. The default Struts configuration file is parsed and an `ApplicationConfig` object is created and stored in the `ServletContext`. The `ApplicationConfig` object for the default application is stored in the `ServletContext` with a key value of `"org.apache.struts.action.APPLICATION"`.
5. Each message resource that is specified in the Struts configuration file for the default application is loaded, initialized, and stored in the `ServletContext` at the appropriate location based on the key attribute specified in each `<message-resources>` element. If no key is specified, the message resource will be stored at the key value `"org.apache.struts.action.MESSAGE"`. Obviously only one message resource can be stored as the default, since keys have to be unique.
6. Next, load and initialize each data source that is has been declared in the Struts configuration file. If no data sources are specified, this step is skipped.
7. Load and initialize each plug-in that is specified in the Struts configuration file. The `init()` method will be called on each and every plug-in specified.
8. Once the default application has been properly initialized, the servlet `init()` method will determine if there are any sub-applications specified and if so, it will repeat steps 4 through 7 for each and every sub-application. The `ServletContext` key where objects are stored will include the sub-application suffix value, so as not to overwrite the default application values.

Figure 5-2 uses a sequence diagram to illustrate the 8 major steps that occur during the initialization of the `ActionServlet`.

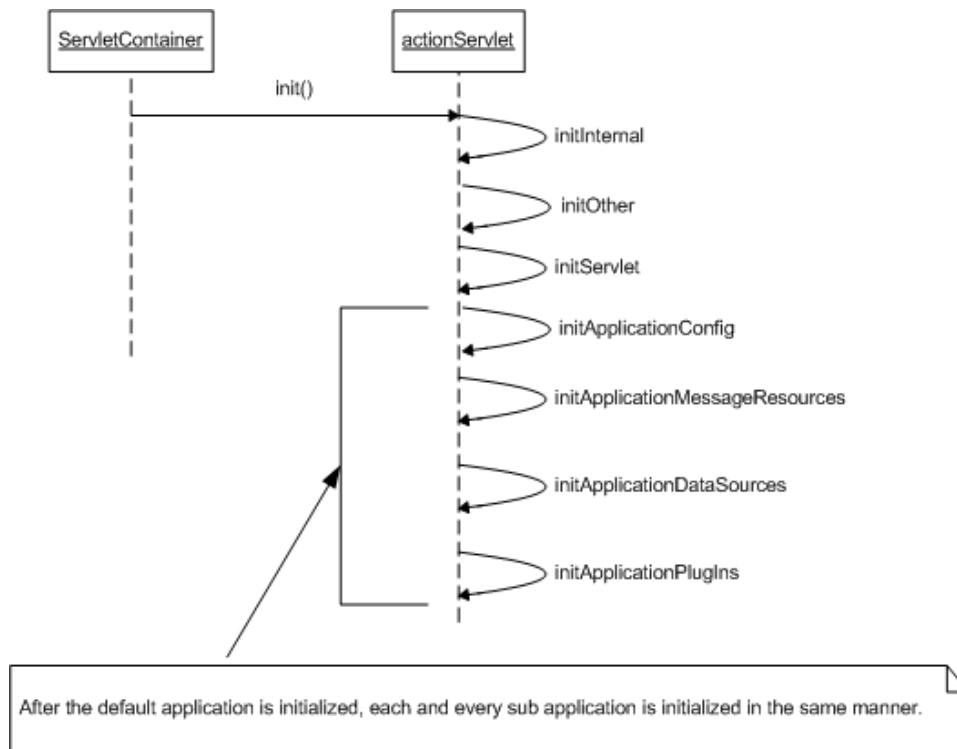


Figure 5-2. Sequence Diagram of the `init()` method in the `ActionServlet`

You may be tempted to setup multiple Struts controller servlets for a single application, in an attempt to achieve better performance. This will most likely not result in better performance or scalability. Servlets are multi-threaded and allow many clients to execute simultaneously. A single servlet is capable of servicing many simultaneous clients.

The second step in Example 5-1 is to call the `process()` method of the `org.apache.struts.action.RequestProcessor`. It's called by the `ActionServlet` instance and passed the current request and response objects.

The RequestProcessor Class

The `org.apache.struts.action.RequestProcessor` was added to framework to allow developers to customize the request handling behavior for an application. Although this type of customization was possible in previous versions by extending the `ActionServlet`, it was necessary to introduce this new class to give each sub-application the ability to have its own customized request handler. The `RequestProcessor` class contains many methods that can be overridden if you need to modify the default functionality.

As shown in Example 5-1, once the correct application for the request has been selected, the `process()` method on the `RequestProcessor` is called to handle the request. The behavior of the `process()` method in the `RequestProcessor` class is very similar to how it looked in earlier versions of the `ActionServlet`. It was just moved to the new class. Example 5-3 shows the implementation of the `process()` method

Example 5-3. The `process()` method of the `RequestProcessor` class handles every request

```
public void process(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {

    // Wrap multipart requests with a special wrapper
    request = processMultipart(request);

    // Identify the path component we will use to select a mapping
    String path = processPath(request, response);
    if (path == null) {
        return;
    }
    if (log.isDebugEnabled()) {
        log.info("Processing a '" + request.getMethod() + "' for path '" + path + "'");
    }

    // Select a Locale for the current user if requested
    processLocale(request, response);

    // Set the content type and no-caching headers if requested
    processContent(request, response);
    processNoCache(request, response);

    // General purpose preprocessing hook
    if (!processPreprocess(request, response)) {
        return;
    }

    // Identify the mapping for this request
    ActionMapping mapping = processMapping(request, response, path);
    if (mapping == null) {
        return;
    }

    // Check for any role required to perform this action
    if (!processRoles(request, response, mapping)) {
        return;
    }

    // Process any ActionForm bean related to this request
    ActionForm form = processActionForm(request, response, mapping);
    processPopulate(request, response, form, mapping);
    if (!processValidate(request, response, form, mapping)) {
        return;
    }
}
```

```
// Process a forward or include specified by this mapping
if (!processForward(request, response, mapping)) {
    return;
}
if (!processInclude(request, response, mapping)) {
    return;
}

// Create or acquire the Action instance to process this request
Action action = processActionCreate(request, response, mapping);
if (action == null) {
    return;
}

// Call the Action instance itself
ActionForward forward =
    processActionPerform(request, response, action, form, mapping);

// Process the returned ActionForward instance
processActionForward(request, response, forward);
}
```

As Example 5-3 shows, there's quite a lot going on in the `process()` method of the `RequestProcessor`. Let's go through the method step by step.

1. The first thing that occurs is a call to the `processMultipart()` method. If the `HttpServletRequest` method is a *POST* and the `contentType` of the request starts with "multipart/form-data", then the standard request object is wrapped with a special version from the Struts framework that deals exclusively with multipart requests. If the request method is a *GET* or the `contentType` is not a multipart, the original request is returned. Unless your application supports uploading files, you will not need to worry about multipart functionality in Struts.
2. The `processPath()` method is called to determine the path component from the URI for the request. Among other things, this information is used to select the appropriate Struts action to invoke.
3. The `processLocale()` method is called to determine the locale of the user making the request and to store a `Locale` object into the user's `HttpSession` object. Depending on the locale attribute in the `<controller>` configuration element, a `Locale` may not be obtained and stored into the user's session in all cases. See Chapter 4 for more details on the attributes of the `<controller>` element.
4. Determine the content type and optional encoding of the request by calling the `processContent()` method. The content type may be configured in the configuration settings and also over-ridden by the JSP pages. The default content type is *text/html*.
5. The `processNoCache()` method is called to determine if the *noCache* attribute is set to true. If it is, add the proper header parameters in the response object to prevent the pages from being cached in the browser. The header parameters include *Pragma*, *Cache-Control*, and *Expires*.

6. The `processPreprocess()` method is called next. It's a general-purpose pre-processing hook that by default just returns true. However, subclasses can override this method and perform conditional logic to decide whether or not to continue processing the request. Because this method gets called before an Action is invoked, this is a good place to validate whether or not the user contains a valid session. If this method returns true, processing of the request will continue. If false, processing of the request will stop.
7. Determine the `ActionMapping` for the request using the path information by calling the `processMapping()` method. If a mapping can't be found using the path information, an error response will be returned.
8. Check to see if there are any security roles configured for the action and if so, that the user has at least one of them. If this step fails, processing will end here. Chapter 14 covers implementing security within the Struts framework.
9. Call the `processActionForm()` method to determine if an `ActionForm` is configured for the `ActionMapping`. If an `ActionForm` has been configured for the mapping, an attempt will be made to find an existing instance in the appropriate scope. Once an `ActionForm` is either found or created, it will be stored within the proper scope using a key that is configured in the `name` attribute for the mapping. If a new `ActionForm` instance needs to be created, the `reset()` method will be called on it.
10. The `processPopulate()` method is called next and if an `ActionForm` is configured for the mapping, it is populated from the request parameter values.
11. The `processValidate()` method is called and if an `ActionForm` has been configured and if the `validate` attribute is set to true for the mapping, then call the `validate()` method on the `ActionForm`. If the `validate()` method detects errors, it will store an `ActionErrors` object into the request scope and the request is automatically forwarded to the resource specified by the input attribute for the mapping. If no errors were detected from the `validate()` method or there wasn't an `ActionForm` for the mapping, processing of the request will continue.
12. Determine if there is a forward or an include attribute configured for the mapping. If so, call the `forward()` or `include()` method on the `RequestDispatcher`, depending on which one is configured in the mapping. The processing of the request would end at this point if either one of these are configured. Otherwise, continue to process the request.
13. Call the `processActionCreate()` method to create or acquire an `Action` instance to process the request. An action cache will be checked to see if the `Action` instance has already been created. If so, that instance will be used to process the request. Otherwise a new instance will be created and stored into the cache.
14. Call the `processActionPerform()` method, which in turn calls the `execute()` method on the `Action` instance. The `execute()` call is wrapped

with a try/catch block so that exceptions can be handled by the `RequestProcessor` if necessary.

15. Call the `processActionForward()` method and pass it the `ActionForward` object that was returned from the `execute()` method. The `processActionForward()` method determines if a redirect or a forward should occur by checking with the `ActionForward` object, which in turn depends on the `<forward>` element.

Extending the `RequestProcessor` Class

It's very easy to create your own custom `RequestProcessor` class. Let's look at an example of how and why you might do this. Suppose your application had a requirement to allow the user to change their Locale at anytime during the application. The default behavior of the `processLocale()` method in the `RequestProcessor` is to set the user's Locale only if it hasn't already been stored in the session, which typically happens during the first request.

The default behavior of the Struts framework is not to store the Locale in the user's session. This can be easily overridden using the `locale` attribute of the `<controller>` element.

Example 5-4 shows a customized `RequestProcessor` class that checks the request each time and updates the user's session if it has changed from the previous one. This allows the user to change their locale preference at any point during the application.

Example 5-4. A customized `RequestProcessor` that overrides the `processLocale` method

```
package com.oreilly.struts.framework;

import javax.servlet.http.*;
import java.util.Locale;
import org.apache.struts.action.Action;
import org.apache.struts.action.RequestProcessor;

/**
 * A customized RequestProcessor that checks the user's preferred Locale
 * from the request each time. If a Locale is not in the session or
 * the one in the session doesn't match the request, the Locale in the
 * request is set in the session.
 */
public class CustomRequestProcessor extends RequestProcessor {

    protected void processLocale(HttpServletRequest request,
                                HttpServletResponse response) {

        // Are we configured to select the Locale automatically?
        if (!appConfig.getControllerConfig().getLocale()) {
            return;
        }
    }
}
```

```
    }

    // Get the Locale (if any) that is stored in the user's session
    HttpSession session = request.getSession();
    Locale sessionLocale = (Locale)session.getAttribute(Action.LOCALE_KEY);

    // Get the user's preferred Locale from the request
    Locale requestLocale = request.getLocale();

    // If the Locale was never added to the session or it has changed, set it
    if (sessionLocale == null || (sessionLocale != requestLocale) ){
        if (log.isDebugEnabled()) {
            log.debug(" Setting user locale '" + requestLocale + "'");
        }
        // Set the new Locale into the user's session
        session.setAttribute( Action.LOCALE_KEY, requestLocale );
    }
}
```

To configure the `CustomizedRequestProcessor` for your application, you will need to add a `<controller>` element to the Struts configuration and include the `processorClass` attribute as shown in this fragment.

```
<controller
  contentType="text/html; charset=UTF-8"
  debug="3"
  locale="true"
  nocache="true"
  processorClass="com.oreilly.struts.framework.CustomRequestProcessor"/>
```

You need to specify the fully qualified class name of the `CustomizedRequestProcessor` as shown in the fragment. Although not every application has a reason to create a custom request processor, having one available in your application can act as a placeholder for future customizations. Therefore, it's a good idea to go ahead and create one for your application and specify it in the configuration file. It doesn't have to override anything in the beginning, but you can add to it as the need unfolds. For more information on the `<controller>` element, see "The Struts Configuration DTD" in Chapter 4.

The Action Class

The `org.apache.struts.action.Action` class is at the heart of the framework. Its purpose is to act as a bridge between the client request and the business operation.

Each `Action` class is typically designed to perform a single task or business operation on behalf of a client. A single task doesn't mean that the action can only perform one business operation, but rather the operations that it does perform, should be cohesive and centered around a single functional unit. In other words, the business operations performed by the Action should be related and have a purpose of fulfilling one functional requirement for the application.

For example, you shouldn't create an Action that performs Shopping Cart functionality as well as handle login and logout responsibilities. These areas of the application are not closely related and shouldn't be combined. On the other hand, you might have one Action class that handles the Login functionality and a separate Action class that handles the Logout.

Later in this chapter, we'll introduce the `DispatchAction` provided by the framework, which supports multiple operations in a single class. These multiple operations should still be cohesive and cover a common functional unit of the application.

Once the correct Action instance is determined, the `processActionPerform()` method is invoked. The `processActionPerform()` method of the `RequestProcessor` is shown in Example 5-5.

Example 5-5. The `processActionPerform` method of the `RequestProcessor`

```
protected ActionForward processActionPerform(HttpServletRequest request,
                                             HttpServletResponse response,
                                             Action action,
                                             ActionForm form,
                                             ActionMapping mapping)
    throws IOException, ServletException {
    try {
        return (action.execute(mapping, form, request, response));
    } catch (Exception e) {
        return (processException(request, response, e, form, mapping));
    }
}
```

The `processActionPerform()` method is responsible for calling the `execute()` method on the `Action` instance. In earlier versions of the Struts framework, the `Action` class only contained a `perform()` method. The `perform()` method has been deprecated in favor of the `execute()` method. This new method is necessary because the `perform()` method declares that it only throws `IOException` and `ServletException`. Due to the declarative exception handling that has been added, the framework needs to catch all instances of `java.lang.Exception` from the `Action` class.

Instead of changing the method signature for the `perform()` method and breaking backwards compatibility, the `execute()` method was added. Currently, the `execute()` method just turns around and invokes the `perform()` method anyway. You should use the `execute()` method in place of the `perform()` method in all of your `Action` classes.

If you look at the source code or documentation for the `Action` class, you may have noticed that there are two different versions of the

methods `execute()` and `perform()`. One version takes a non-HTTP request and response, while the second version contains the HTTP versions. Generally, you'll only ever need to use the HTTP version, unless you are using a non-HTTP servlet. For now, the non-HTTP versions just attempt to cast the request and response objects to their HTTP counterparts and invoke the HTTP version of their respective method.

Although the `org.apache.struts.action.Action` class is not abstract, the `execute()` method doesn't perform any useful functionality. You will need to extend the `Action` class with ones of your own. Example 5-6 shows the `LoginAction` from the storefront application.

Example 5-6. The LoginAction from the Storefront Web Application

```
package com.oreilly.struts.storefront.security;

import java.util.Locale;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.exceptions.BaseException;
import com.oreilly.struts.storefront.framework.UserContainer;
import com.oreilly.struts.storefront.framework.StorefrontBaseAction;
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.service.IStorefrontService;

/**
 * Implements the logic to authenticate a user for the storefront application.
 */
public class LoginAction extends StorefrontBaseAction {
    /**
     * Called by the controller when the a user attempts to login to the
     * storefront application.
     */
    public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
        throws Exception{

        // Get the user's login name and password. They should have already
        // validated by the ActionForm.
        String email = ((LoginForm)form).getEmail();
        String password = ((LoginForm)form).getPassword();

        // Login through the security service
        IStorefrontService serviceImpl = getStorefrontService();
        UserView userView = serviceImpl.authenticate(email, password);

        UserContainer existingContainer = null;
        HttpSession session = request.getSession(false);
        if ( session != null ){
```



```
        existingContainer = getUserContainer(request);
        session.invalidate();
    }else{
        existingContainer = new UserContainer();
    }

    // Create a new session for the user
    session = request.getSession(true);
    existingContainer.setUserView(userView);
    session.setAttribute(IConstants.USER_CONTAINER_KEY, existingContainer);

    return mapping.findForward(IConstants.SUCCESS_KEY);
}
}
```

When the `execute()` method in the `LoginAction` is called, the email and password values are retrieved and passed to the `authenticate()` method. If no exception is thrown by the `authenticate()` business operation, a new `HttpSession` is created and a `JavaBean` that contains user information is stored into the user's session.

A common bug that inexperienced Struts developers sometimes introduce into their applications, is to not implement the `execute()` method exactly. If you misspell it or don't implement the signature exactly, the method will never be called. Unfortunately, you will not get a compiler error or even a runtime error telling you of this. This is due to the fact that the Struts `Action` class, which all action classes must extend, has a default `execute()` method that just returns a null.

The `UserView` contains simple properties like `firstName` and `lastName` that can be used by the presentation. These types of presentation `JavaBeans` are commonly referred to as value objects, but are more formally called data transfer objects because they are used to transfer data from one layer to another. In this example, the data is being transferred from the security service to the presentation layer. The `UserView` class is shown in Example 5-7.

Example 5-7. The UserView Data Transfer Object

```
package com.oreilly.struts.storefront.customer.view;

import com.oreilly.struts.storefront.framework.view.BaseView;
/**
 * Mutable data representing a user of the system.
 */
public class UserView extends BaseView {
    private String lastName;
    private String firstName;
    private String emailAddress;
    private String creditStatus;

    public UserView(){
        super();
    }
}
```

```
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getLastName() {
    return lastName;
}

public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}

public void setCreditStatus(String creditStatus) {
    this.creditStatus = creditStatus;
}

public String getCreditStatus() {
    return creditStatus;
}
}
```

Value objects or data transfer objects will be further discussed in the next chapter.

Example 5-6 also uses a class called `UserContainer`. This class has not been fully introduced yet; so don't worry about having missed something. The `UserContainer` is a wrapper around any data that might normally be put directly into the user's session. By using this object and storing everything within it, retrieval and clean up becomes much easier. It's not part of the Struts framework, but is a class that we'll make a case for in Chapter 21 "Struts Design Strategies".

The Action Class Cache

Because Action instances are supposed to be thread-safe, only a single instance of each Action class is created for an application. All client requests share the same instance and are able to invoke the `execute()` method at the same time.

In actuality, client requests are not processed at exactly the same time unless the server has multiple processors and the software is written to take advantage of this. The CPU is really context switching so fast; that it seems like parallel processing is occurring. But for this discussion, we'll assume that the requests are being processed at the same time.

The `RequestProcessor` contains a `HashMap`, where the keys are the names of all of the Action classes that are specified in the configuration file, and the value for each key is the single instance of that Action.

During the `processActionCreate()` method of the `RequestProcessor` class, the framework checks the `HashMap` to see if an instance has already been created. If it has, this instance is returned. Otherwise, a new instance of the `Action` class is created, put into the `HashMap`, and returned. The section of the code that creates a new Action instance is synchronized to ensure that only one thread will create an instance. Once a thread creates an instance and inserts it into the `HashMap`, all future threads will use the instance from the cache.

The `ActionForward` Class

As you saw in the discussion of the `Action` class, the `execute()` method returns an `ActionForward` object. The `ActionForward` class represents a logical abstraction of a web resource. This resource is typically a JSP page or a Java servlet.

The `ActionForward` class acts as a wrapper around the actual resource so that there's less coupling of the application to the physical resource. The physical resource is specified only in the configuration file and not in the code itself. The `RequestDispatcher` may perform either a forward or redirect for an `ActionForward`, depending on what has been configured in the `<forward>` element.

To return an `ActionForward` from an `Action`, you can either create one dynamically in the action class or use the action mapping to locate one that has been pre-configured in the configuration file, which is more common. The following code fragment illustrates how you can use the action mapping to locate an `ActionForward` based on its logical name.

```
| return mapping.findForward( "Success" ); |
```

The code fragment shows that an argument of `"Success"` is passed to the `findForward` method. The argument in the `findForward` method must match one of the names specified in either the `<global-forwards>` section or one specific to the action from where it's being called. The following fragment shows `<forward>` elements defined for the `"/signin"` action mapping:

```
| <action  
|   input="/security/signin.jsp"  
|   name="loginForm"  
|
```

```
path="/signin"
scope="request"
type="com.oreilly.struts.storefront.security.LoginAction"
validate="true">
  <forward name="Success" path="/index.jsp" redirect="true"/>
  <forward name="Failure" path="/security/signin.jsp" redirect="true"/>
</action>
```

The `findForward()` method in the `ActionMapping` class first calls the `findForwardConfig()` method to see if there's a `<forward>` element specified at the action level with the corresponding name. If not, the `<global-forwards>` section is checked. When an `ActionForward` is found that matches, it's returned back to the `RequestProcessor` from the `execute()` method. Here's the `findForward()` method from the `ActionMapping` class.

```
public ActionForward findForward(String name) {
    ForwardConfig config = findForwardConfig(name);
    if (config == null) {
        config = getApplicationConfig().findForwardConfig(name);
    }
    return ((ActionForward) config);
}
```

If the `findForward()` method does not find a forward that matches the `name` argument, it will not complain. A null will be returned and you will receive a blank page because no output will be written to the response.

Extending the ActionMapping Class

Some developers find it necessary or desirable to extend the `ActionMapping` class and add more parameters or attributes that can be made available to their `Action` class. Although this is rare, it's not difficult to do. It basically involves three steps:

1. Create a class that extends the `ActionMapping` class and add the additional properties as instance variables to it. You can't extend `ActionConfig` because the most of the APIs still require the `ActionMapping` class, for example the `execute()` method has `ActionMapping` as one of its parameters.
2. Set the `className` attribute in the `<action>` element to be the fully qualified name of your extension.
3. Use the `<set-property>` element to initialize the properties of your instance.

Since the parameter type of the mapping in the `execute()` method is `ActionMapping`, you will need to cast the mapping object to your specialized class before you can access the additional properties.

Creating Multi-threaded Action Classes

A single `Action` instance is created for each `Action` class in the framework. Every client request will share the same instance, just as every client request shares the same `ActionServlet` instance. Because of this, you must ensure that your Action classes operate properly in a multi-threaded environment, just as servlets are required to do.

To be thread-safe, it's important that your Action classes not use instance variables to hold client-specific state. You are free to use instance variables to hold state information; it just shouldn't be specific to one client or request. For example, you might create an instance variable of type `org.apache.commons.logging.Log` to hold onto a logger, as the Struts `RequestProcessor` class does. The log instance can be used by all requests because the logger is thread-safe and does not hold state for a specific client or request.

For client-specific state however, you should declare the variables inside the `execute()` method. These local variables are allocated in a different memory space than instance variables. Each thread that enters the `execute()` method has its own stack to put variables, and therefore there's less chance of overriding the state of other threads.

Business Logic and the Action Class

Some developers get confused about what logic belongs in an `Action` class. The Action class is not the proper place to put your application's business logic. If you look back to Figure 3-6 from Chapter 3, you can see that the Action class is still part of the controller. It's just been separated out from the `ActionServlet` and `RequestProcessor` for the sake of convenience.

Business logic belongs in the domain model. These may be EJBs, CORBA objects, or maybe just a layer of services written on top of a data source and a connection pool. The point is that that business domain should be unaware of the type of presentation that is using it. This allows your model components to be more easily reused by other applications. Example 5-8 illustrates the `GetItemDetailAction` from the storefront application that calls on the model to retrieve the detail information for an item in the catalog.

Example 5-8. The Action class should delegate the business logic to a model component

```
package com.oreilly.struts.storefront.catalog;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import com.oreilly.struts.storefront.framework.exceptions.BaseException;
import com.oreilly.struts.storefront.framework.UserContainer;
import com.oreilly.struts.storefront.framework.StorefrontBaseAction;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
```

```
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.service.IStorefrontService;

/**
 * An action that gets an ItemView based on an id parameter in the request and
 * then inserts the item into an ActionForm and forwards to whatever
 * path is defined as Success for this action mapping.
 */
public class GetItemDetailAction extends StorefrontBaseAction {
    public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
        throws Exception {
        // Get the primary key of the item from the request
        String itemId = request.getParameter( IConstants.ID_KEY );

        UserContainer userContainer = getUserContainer(request);

        // Call the storefront service and ask it for an ItemView for the item
        IStorefrontService serviceImpl = getStorefrontService();
        ItemDetailView itemDetailView = serviceImpl.getItemDetailView( itemId );

        // Set the returned ItemView into the Dynamic Action Form
        // The parameter name 'view' is what is defined in the struts-config
        ((DynaActionForm)form).set("view", itemDetailView);

        // Return the ActionForward that is defined for the success condition
        return mapping.findForward( IConstants.SUCCESS_KEY );
    }
}
```

Although the `GetItemDetailAction` class in Example 5-8 performs some presentation logic, it delegates the real work of getting the item information to the storefront service. The reason that this is the best approach is that the action doesn't know the internals of the storefront service or the `getItemDetailView()` method. It could be a local object that performs JDBC calls, a session bean performing a remote call to an application server, or some other implementation. If the model implementation changed, the action would be protected from that change. Because the storefront service is unaware of the type of client using it, other types of clients can use it, not just Struts. Decoupling of the Action classes from the business objects will be explored further in the next chapter.

Using the Pre-Built Struts Actions

The Struts framework includes five out-of-the-box `Action` classes, which you can easily integrate into your applications and save yourself development time. Some of these are more useful than others, but all of them deserve some attention. The classes are contained within the `org.apache.struts.actions` package. We'll briefly cover each one and discuss the pros and cons of using these in your applications.

The `org.apache.struts.actions.ForwardAction`

There are many situations where you just need to forward from one JSP page to another, without really having a need to go through an `Action` class. However, calling a JSP directly should be avoided. As the beginning of this Chapter explained, the controller has the responsibility of selecting the application to handle the request, and to store the `ApplicationConfig` and `MessageResources` for that application in the request. If this step is bypassed, functionality like selecting the correct messages from the resource bundle may not work properly.

Another reason that calling a JSP directly is not a good idea is that it violates the component responsibilities of MVC. The controller is supposed to process all requests and select a view for the client. If your application were allowed to call the page directly, the controller is not getting the opportunity to fulfill its obligations to the MVC contract.

To solve these problems and to prevent you from having to create an `Action` class that just performs a simple forward, you can use the provided `ForwardAction`. This `Action` simply performs a forward to a URI that is configured for it. In the Struts configuration file, you specify an `<action>` element using the `ForwardAction` as the type attribute:

```
<action
  input="/index.jsp"
  name="loginForm"
  path="/viewsignin"
  parameter="/security/signin.jsp"
  scope="request"
  type="org.apache.struts.actions.ForwardAction"
  validate="false"/>
</action>
```

When the `"/viewsignin"` action is selected, the `perform()` method of the `ActionForward` class gets called.

Although the `ActionForward` is part of 1.1 and it extends the Struts `Action` class, it still uses the `perform()` method, rather than the `execute()` method. At the time of this writing, Struts is still in beta and this might be fixed by the time it's released for *General Availability* (GA).

Notice that when using the `ForwardAction` in a `<action>` element, the `parameter` attribute is used to specify where to forward to, instead of an actual `<forward>` element. Other than this difference, you call the action in the same way as any other action in the configuration file.

The `ActionForward` class comes in handy when you need to integrate your Struts application with other servlets or JSP pages, but still take advantage of the controller

functionality. The `ActionForward` class is one of the most valuable of the pre-built Action classes included with the framework

The `org.apache.struts.actions.IncludeAction`

The `IncludeAction` is similar in some respects to the `ForwardAction`. The `IncludeAction` was originally created to make it easier to integrate existing servlet-based components into a Struts based web application. If your application is using the `include()` method on the `RequestDispatcher`, you can implement the same behavior using the `IncludeAction`.

You specify the `IncludeAction` in an action mapping in the same manner that you do for the `ForwardAction`, except you use the `IncludeAction` in the `type` attribute.

```
<action
  input="/subscription.jsp"
  name="subscriptionForm"
  path="/saveSubscription"
  parameter="/path/to/processing/servlet"
  scope="request"
  type="org.apache.struts.actions.IncludeAction"/>
```

You must include the `parameter` attribute and specify a path to the servlet that you wish to include.

The `org.apache.struts.actions.DispatchAction`

The purpose of the `DispatchAction` is to allow multiple operations that would be scattered throughout multiple Action classes, to reside in a single class. The idea is that there is related functionality for a service and instead of spreading this behavior in multiple Action classes, it should be kept together in the same class. For example, if you think about an application that contains a typical *ShoppingCart* service. It usually needs the ability to add items to the cart, view the items in the cart, and update the items and quantity of the cart. One design is to create three separate Actions classes, `AddItemAction`, `ViewShoppingCartAction`, and `UpdateShoppingCartAction`.

Although this solution would work and would be a valid approach, there probably is similar functionality that all three `Action` classes must perform before carrying out their assigned business operation. By combining the functionality, you are also making it easier to maintain the application. If the current *ShoppingCart* implementation was exchanged for an alternate version, all of the code is located in a single class.

To use the `org.apache.struts.actions.DispatchAction`, create a class that extends the `org.apache.struts.actions.DispatchAction`. Add a method for each and every function that you need to perform on the service. Your class should not contain the typical `execute()` method as other Action classes do. The `execute()` method is implemented by the abstract `DispatchAction`.

You must include one method in your `DispatchAction` for every action you want to invoke for this `DispatchAction`. Look at Example 5-9 to better understand this.

Example 5-9. The functionality of the ShoppingCart is put into a single DispatchAction

```
package com.oreilly.struts.storefront.order;

import java.io.IOException;
import java.text.Format;
import java.text.NumberFormat;
import java.util.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.actions.DispatchAction;
import com.oreilly.struts.storefront.service.IStorefrontService;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.framework.UserContainer;
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.framework.ShoppingCartItem;
import com.oreilly.struts.storefront.framework.ShoppingCart;
import com.oreilly.struts.storefront.framework.StorefrontDispatchAction;
/**
 * Implements all of the functionality for the ShoppingCart.
 */
public class ShoppingCartActions extends StorefrontDispatchAction {
    /**
     * This method just forwards to the success state, which should represent
     * the shoppingcart.jsp page.
     */
    public ActionForward view(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
        throws Exception {

        // Call to ensure that the user container has been created
        UserContainer userContainer = getUserContainer(request);
        return mapping.findForward(IConstants.SUCCESS_KEY);
    }
    /**
     * This method updates the items and quantities for the shopping cart from the
     * request.
     */
    public ActionForward update(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
        throws Exception {

        updateItems(request);
        updateQuantities(request);
        return mapping.findForward(IConstants.SUCCESS_KEY);
    }
}
```

```
/**
 * This method adds an item to the shopping cart based on an id and qty
 * parameters from the request.
 */
public ActionForward addItem(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {

    UserContainer userContainer = getUserContainer(request);

    // Get the id for the product to be added
    String itemId = request.getParameter( IConstants.ID_KEY );
    String qtyParameter = request.getParameter( IConstants.QTY_KEY );

    int quantity = 1; // Default value to add
    if(qtyParameter != null) {
        Locale userLocale = userContainer.getLocale();
        Format nbrFormat = NumberFormat.getNumberInstance(userLocale);
        try {
            Object obj = nbrFormat.parseObject(qtyParameter);
            quantity = ((Number)obj).intValue();
        }
        catch(Exception ex) {
            // Just use the default qty already set
        }
    }

    // Call the storefront service and ask it for an ItemView for the item
    IStorefrontService serviceImpl = getStorefrontService();
    ItemDetailView itemDetailView = serviceImpl.getItemDetailView( itemId );

    // Add the item to the cart and return
    userContainer.getCart().addItem(new ShoppingCartItem(itemDetailView,
quantity));
    return mapping.findForward(IConstants.SUCCESS_KEY);
}

/**
 * Update the items in the shopping cart. Currently, only deletes occur
 * during this operation.
 */
private void updateItems(HttpServletRequest request) {
    // Multiple checkboxes with the name "deleteCartItem" are on the
    // form. The ones that were checked are passed in the request.
    String[] deleteIds = request.getParameterValues("deleteCartItem");

    // Build a List of item ids to delete
    if(deleteIds != null && deleteIds.length > 0) {
        int size = deleteIds.length;
        List itemIds = new ArrayList();
        for(int i = 0; i < size; i++) {
            itemIds.add(deleteIds[i]);
        }
    }
}
```

```
    }
    // Get the ShoppingCart from the UserContainer and delete the items
    UserContainer userContainer = getUserContainer(request);
    userContainer.getCart().removeItems(itemIds);
  }
}

/**
 * Update the quantities for the items in the shopping cart.
 */
private void updateQuantities(HttpServletRequest request) {
    Enumeration enum = request.getParameterNames();
    // Iterate through the parameters and look for ones that begin with
    // "qty_". The qty fields in the page were all named "qty_" + itemId.
    // Strip off of the id of each item and the corresponding qty value.
    while(enum.hasMoreElements()) {
        String paramName = (String)enum.nextElement();
        if(paramName.startsWith("qty_")) {
            String id = paramName.substring(4, paramName.length());
            String qtyStr = request.getParameter(paramName);
            if(id != null && qtyStr != null) {
                ShoppingCart cart = getUserContainer(request).getCart();
                cart.updateQuantity(id, Integer.parseInt(qtyStr));
            }
        }
    }
}
```

The `com.oreilly.struts.storefront.order.ShoppingCartActions` class contains the methods `addItem()`, `update()`, and `view()`. Each of these methods would normally be put into separate Action class. With the `DispatchAction`, they can all be kept together into a single Action class.

There are two other methods in the `ShoppingCartActions` class that we didn't mention, `updateItems()` and `updateQuantities()`. These methods are private utility methods used by the other action methods within the class. They are not called outside of this Action class. You can tell this by noting they do not have the required method signature.

To use your specialized `DispatchAction` class, you need to configure each `<action>` element that uses it, a little differently than other mappings. Example 5-10 illustrates how the `ShoppingCartActions` class from Example 5-9 is declared in the configuration file.

Example 5-10. The parameter attribute must be specified when using a subclass of the `DispatchAction`

```
<action path="/cart"
        input="/order/shoppingcart.jsp"
        parameter="method"
```

```
scope="request"
type="com.oreilly.struts.storefront.order.ShoppingCartActions"
validate="false">
<forward name="Success" path="/order/shoppingcart.jsp" redirect="true"/>
</action>
```

The “/cart” action mapping shown in Example 5-10 specifies the *parameter* attribute and sets the value to be the literal string “*method*”. The value specified here becomes very important to the *DispatchAction* when invoked by a client. The *DispatchAction* uses this attribute value to determine which method in your specialized *DispatchAction* to invoke. Instead of just calling the action mapping “/cart”, an additional request parameter is passed where the key is the value specified for the *parameter* attribute from the mapping. The value of this request parameter must be the name of the method to invoke. For example, to invoke the *addItem()* method, you would do call the action like this:

```
| "cart.do?method=addItem"
```

The request parameter and value *method=addItem* is used by the *DispatchAction* class to determine which method to invoke. You must have a method in your *DispatchAction* subclass that matches the value on the right side of the equal sign. The method name must match exactly and the method must include the parameters normally found in the *execute()* method. The following fragment highlights the method signature for the *addItem()* method from Example 5-9.

```
public ActionForward addItem( ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception;
```

The *DispatchAction* uses reflection to locate a method that matches exactly the same name as the request parameter value and contains the same number and type of arguments. Once found, the method will be invoked and the *ActionForward* object returned just as any other *Action* class does.

Although the *DispatchAction* does use Java reflection to invoke the correct method, the performance of the reflection APIs in Java 1.3 and newer are so much better that it's not a problem for this small amount of work. Since reflection is being used however, the method must be declared public or the *perform()* method in the abstract *DispatchAction* will not be able to invoke it. You can still declare other private or protected methods, but the ones to be called by the *DispatchAction* must be declared public.

The `org.apache.struts.actions.LookupDispatchAction`

The `LookupDispatchAction` as you might have guessed, is a subclass of the `DispatchAction` class. From a high level, it performs a similar task as the `DispatchAction`.

Just like the `DispatchAction`, the `LookupDispatchAction` allows you to specify a class with multiple methods, where one of the methods is invoked based on the value of a special request parameter that is specified the configuration file. That's about where the similarity ends. The `DispatchAction` uses the value of the request parameter to determine which method to invoke, however, the `LookupDispatchAction` uses the value of the request parameter to perform a reverse lookup from the resource bundle using the parameter value, and match it to a method in the class.

An example will help you understand this better. You first need to create a class that extends the `LookupDispatchAction` and which implements the `getKeyMethodMap()` method. This method returns a `java.util.Map`, which contains a set of key/value pairs.

The keys of this Map should match those from the resource bundle. The value that is associated with each key in the Map should be the name of the method in your `LookupDispatchAction` subclass that will be invoked when a request parameter is included that is equal to the message from the resource bundle for the key.

The following fragment shows an example of the `getKeyMethodMap()` method for the `ProcessCheckoutAction` in the storefront application.

```
protected Map getKeyMethodMap() {  
    Map map = new HashMap();  
    map.put("button.checkout", "checkout" );  
    map.put("button.saveorder", "saveorder" );  
    return map;  
}
```

For the purposes of this discussion, let's suppose we have the following resources in the message resource bundle:

```
button.checkout=Checkout  
button.saveorder=Save Order
```

and that we have specified the following `<action>` element in the Struts configuration file:

```
<action path="/processcheckout"  
    input="/checkout.jsp"  
    name="checkoutForm"  
    parameter="action"  
    scope="request"  
    type="com.oreilly.struts.storefront.order.ProcessCheckoutAction">  
    <forward name="Success" path="/order/ordercomplete.jsp"/>  
</action>
```

```
| </action>
```

Then when you have a JSP page that performs a POST using the *processcheckout* action, a URL parameter of action=Checkout will be sent in the request header. Example 5-11 shows the JSP page that calls the *processcheckout* action.

Example 5-11. The checkout.jsp which calls the ProcessCheckoutAction when posted

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html:html>
<head>
<title>Virtual Shopping with Struts</title>
<html:base/>
<script language=javascript src="include/scripts.js"></script>
<link rel="stylesheet" href="../stylesheets/format_win_nav_main.css"
type="text/css">
</head>

<body topmargin="0" leftmargin="0" bgcolor="#FFFFFF">

<!-- Header Page Information -->
<%@ include file="../include/head.inc"%>

<!-- Nav Bar -->
<%@ include file="../include/menubar.inc"%>

<br>

    Display order summary and take credit card information here

    <html:form action="/processcheckout">
        <html:submit property="action">
            <bean:message key="button.checkout"/>
        </html:submit>
    </html:form>

<br><br>
<%@ include file="../include/copyright.inc"%>
</body>

</html:html>
```

The key to understanding how all of this works is that the submit button in Example 5-11 will have a name of *action* and it's value will be the value returned from the *<bean:message>* tag. This is more evident when you see the HTML source that is generated from the JSP above. The following fragment only shows the source generated inside the *<html:form>* tag.

```
| <form
|   name="checkoutForm"
|   method="POST"
```

```
    action="/storefront/action/processcheckout">
      <input type="submit" name="action" value="Checkout" alt="Checkout">
    </form>
```

You can see in this HTML source that when the form `checkoutForm` is posted, the `action=Checkout` URL parameter will be included. The `ProcessCheckoutAction` will take the value `Checkout` and find the message resource key that has this value. In the case of this example, the key will be `button.checkout` which according to the `getKeyMethodMap()` method shown earlier, maps to the method `checkout()`.

Whew! That's a long way to go just to determine which method to invoke. The intent of this class is to make it easier when you have an HTML form with multiple submit buttons with the same name. One submit button may be a Checkout action and another might be a Save Order action. Both buttons would have the same name, for example `"action"` as with the earlier example. But the value of each button would be different. This may not be an `Action` class that you will use often, but in certain situations, it can save development time for your applications.

The `org.apache.struts.actions.SwitchAction`

The `SwitchAction` is new to the framework. It was added to support switching from one sub-application to another and then forwarding control to a resource within the sub-application.

There are two request parameters that are required to be present when this Action is invoked. The `prefix` request parameter specifies the application prefix, beginning with a `"/"`, of the sub-application to which control should be switched. If you need to switch to the default application, use a zero-length string `""`. The appropriate `ApplicationConfig` object will be stored into the request, just as it is when a new request arrives at the `ActionServlet`.

The second request parameter that is required is the `page` parameter. This parameter should specify an application-relative URI, beginning with a `"/"`, to which control should be forwarded to once the correct sub-application is selected. This Action is very straightforward. However, you'll only have need for it if you use more than one Struts sub-application.

The Utilities Classes

When building web applications, much of the tasks to retrieve and process requests are quite repetitive. Like any good framework, Struts places most of this tedious functionality into utility classes where it can be shared and used by many different components and applications. This separation of utilitarian functionality from regular application specific functionality allows for greater reuse and less redundancy throughout the framework and within your own applications.

The utility classes used by Struts are located in several packages. Several of the utility components were so generic and beneficial to applications, that they have been moved out of the Struts framework and into the larger Jakarta Commons Project. These packages include *BeanUtils*, *Collections*, and the *Digester* component mentioned in Chapter 3.

One of the Java packages that remains in the Struts package hierarchy is the `org.apache.struts.util` package. Everything from the `MessageResources` class to the `StrutsValidatorUtil`, which is part of the new validator component added to the core framework in 1.1, is part of this package. This package contains many different classes, with different purposes and responsibility. Although the Struts framework classes have strong dependencies on the utility classes, in general, utility classes should only have dependencies on other utility classes and framework components that are lower on the food chain. This is mostly true in the Struts framework, with a few minor exceptions that are not too serious.

The utility classes within the `util` package assist the rest of the framework in solving mundane problems that all web applications encounter. We will not cover all of the classes in the package, but instead will highlight some of the more often used components.

RequestUtils

The `org.apache.struts.util.RequestUtils` class provides general-purpose utility methods that are common when processing a servlet request within Struts. You have already seen several examples that utilize the `RequestUtils` class. For example, one of the most important and first to be invoked for a request is the `selectApplication()` method that is called by the `ActionServlet` when a new request arrives. Every method in the `RequestUtils` class is designed to be thread-safe and doesn't declare any instance variables. In fact, every method in the class is also static.

You seldom have a need to modify any of the functionality within the `RequestUtils` class. However, you should get to know the functionality that does exist, to prevent you from creating functionality in your application that might already exist there. Another reason to get familiar with the methods is that it helps in understanding what the entire framework is doing on your behalf.

ResponseUtils

The purpose of the `org.apache.struts.util.ResponseUtils` class is very similar to the `RequestUtils` class, except that its purpose is to aid in building a response, rather than aiding with the request.

There are only a few methods within the class, but the JSP tag libraries included with Struts use them extensively for filtering and writing data destined for the response object.

The Commons BeanUtils Package

The `org.apache.commons.beanutils` package contains several classes that are used throughout the Struts framework. From the standpoint of the Struts framework, the two most important are the `BeanUtils` and `PropertyUtils` classes.

As you might have guessed, the `BeanUtils` class is used with JavaBeans. The Struts components primarily use just three of the methods in the `BeanUtils` class:

- `populate()`
- `getProperty()`
- `getArrayProperty()`

The `populate()` method is used to fill a JavaBean with data, using a map of key/value pairs. The method signature for the `populate()` method is shown here.

```
public static void populate( Object bean, Map properties )  
    throws IllegalAccessException, InvocationTargetException;
```

The `getProperty()` method returns a String representation of the property stored in the variable with the name that matches the value of the `name` parameter.

```
public static String getProperty( Object bean, String name )  
    throws IllegalAccessException, InvocationTargetException, NoSuchMethodException;
```

Regardless of the type of property that the name argument references, it will be converted and returned as a String.

The `getArrayProperty()` method returns the value of the specified array property of the specified bean, as a String array. Here is the method signature for the `getArrayProperty()` method.

```
public static [] getArrayProperty( Object bean, String name )  
    throws IllegalAccessException, InvocationTargetException, NoSuchMethodException;
```

Although the Java language provides reflection and introspection as part of its core API's, the `BeanUtils` class provides convenience wrappers around these API's.

The other class that is used by the Struts framework is that `PropertyUtils` class. However, only one method is currently used, the `getProperty()` method.

```
public static Object getProperty( Object bean, String name )  
    throws IllegalAccessException, InvocationTargetException, NoSuchMethodException;
```

The `getProperty()` method in the `PropertyUtils` class returns the value of the specified property, without attempting to convert the type. Much of the code that is in the `PropertyUtils` class was originally implemented in the `BeanUtils` class. It was moved to its own class due to the size that `BeanUtils` was becoming.

The Commons Collection Package

Although starting with version 1.3, the Java core libraries included much requested and needed collection classes, however there are a few holes left unfilled. These holes are what the classes within the Commons Collection package aims to fill.

Among the features of the collection package are:

- Implementations of Lists and Maps that are designed for fast access.
- Methods to utilize set-theory properties of collections, such as unions, intersections, and the closure properties.
- Adaptor classes that allow conversions between Java 1.1 containers and Java 1.2 style containers.

Currently, the Struts framework only uses the `FastHashMap` class from the collections package. The `FastHashMap` is designed to operate in a multi-threaded environment, where most of the calls are read-only. The `FastHashMap` extends `java.util.HashMap` and provides two different modes, *slow* and *fast*. In slow mode, all access is synchronized. This is appropriate for when initialization is taking place. Once initialization is complete and mostly read-only calls happen, the Map can be switched to *fast* mode by calling the `setFast(true)`. In *fast* mode, read access is not synchronized and write calls use cloning for performance.

Security in the Action Classes

Action classes, if designed and scoped properly, can perform some very important functionality for an application. To prevent unauthorized users from finding a way to execute an action, it seems reasonable that Action classes should have a way to authorize a user to perform the intended action. The `processRoles()` method is designed to check to see if there are any roles defined in the configuration file this the Action, and if so, call the `isUserInRole()` method on the request. The problem with this approach is that not all applications can defined their roles ahead of time. In some applications, roles can be dynamically added and removed and they can't be enumerated beforehand. In this case, there must be an alternative approach to dealing with user, roles, and permissions. Chapter 14 "Security in your Struts Web Applications" will cover security in depth, including an approach to handling roles in a Struts application.

6

Struts Model Components

This chapter introduces the components that make up the model portion of a Struts application. The model represents the business data for an application and should closely resemble the real-world entities and business processes for the organization. We will explore the roles and responsibilities of the model components within the Struts framework and focus on building an architecturally correct implementation for the storefront application. Special attention will be given to using a persistence framework that can be easily and effortlessly integrated into a Struts application.

[Editors: Because these chapters are being shown before the entire book is finished, the source code within these will not be the final versions. The Storefront application is being built as the book is written and the source code will undergo more and more functionality as we go along. You'll have to bear with me while this happens.]

The “M” in MVC

As Chapter 1 outlined, the “M” in the acronym MVC, stands for Model. The model components of an application are arguably the most valuable software artifacts to an organization. The model includes the business entities and the rules that govern access and modification of the data. It's vital that this is done in a single location in order to maintain valid data integrity, reduce redundancy, and increase reuse.

The model should remain independent of the type of client that's being used to access the business objects and their associated rules. In fact, the components within the model should not even be aware of what type of client or framework that's using it. There's an axiom that goes, “dependencies go down, data goes up.” The idea is that when using a *layered architecture*, the upper layer may have dependencies on lower layers, but the

lower layers should not depend on those layers above them. Figure 6-1 illustrates how this principle is applied to a typically Struts architecture.

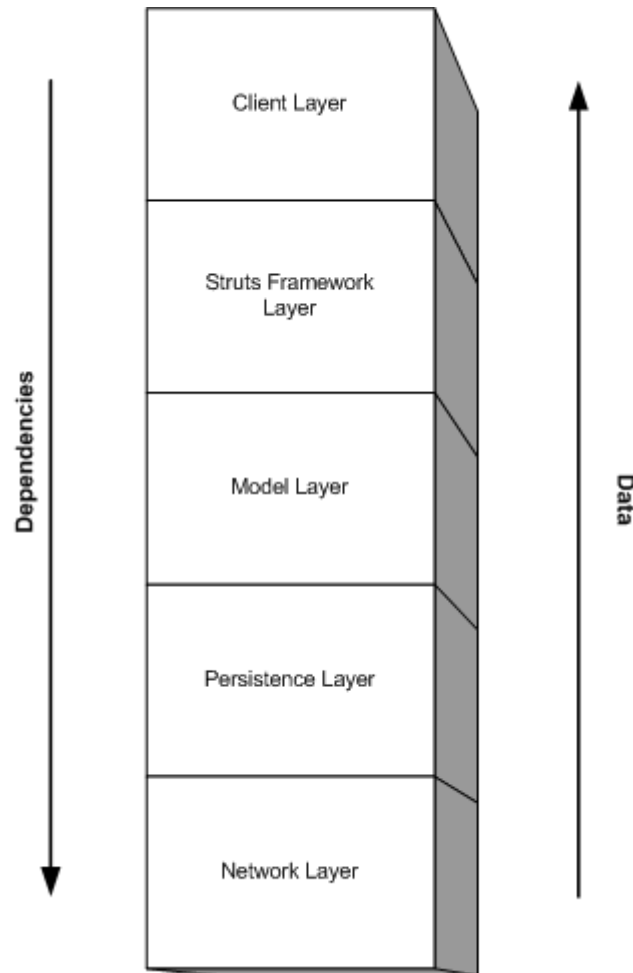


Figure 6-1. Application layers should only depend on lower layers

If you find yourself importing packages or classes from the Struts framework into your model, then you are violating this principle. Coupling a lower layer to an upper one will make maintenance, reuse, and future enhancements more difficult.

Before we get into the details of designing and building a model for a Struts application, it's best to understand the different types of models and how each one is relevant to a project.

The Different Types of Models

The term model has many different meanings. In very general terms, a model is a representation of some aspect of reality. Whether the model represents a shop where products are bought and sold, an auction house where bids are placed, or a way to predict how a weather storm will move. All of these examples are based on real concepts. The main purpose of creating a model is to help understand, describe, or simulate how things work in the “real” world.

In software development, the term model is used to indicate both the logical representation of real world entities, as well as the physical creation of classes and interfaces that can be utilized by a program. The first step however, should always be to perform a thorough analysis of the problem domain. Once use cases are complete, the next step should be to develop a conceptual model.

Developing a Conceptual Model

During analysis of the problem domain, a *conceptual* model should be developed based on the real-life entities within the problem space. The entities in a conceptual model have less to do with the software components of the system and more to do with the physical entities that are fundamental to the business. The conceptual model usually illustrates the concepts, the relationships between them, and the attributes that belong to each concept. The behavior is not represented in this type of model.

The conceptual model is developed from the set of use-cases for the system. Its purpose is to help to identify the entities that will most likely become classes in the design stage and also to help understand the problem domain better. Figure 6-2 illustrates a conceptual model for the storefront application.

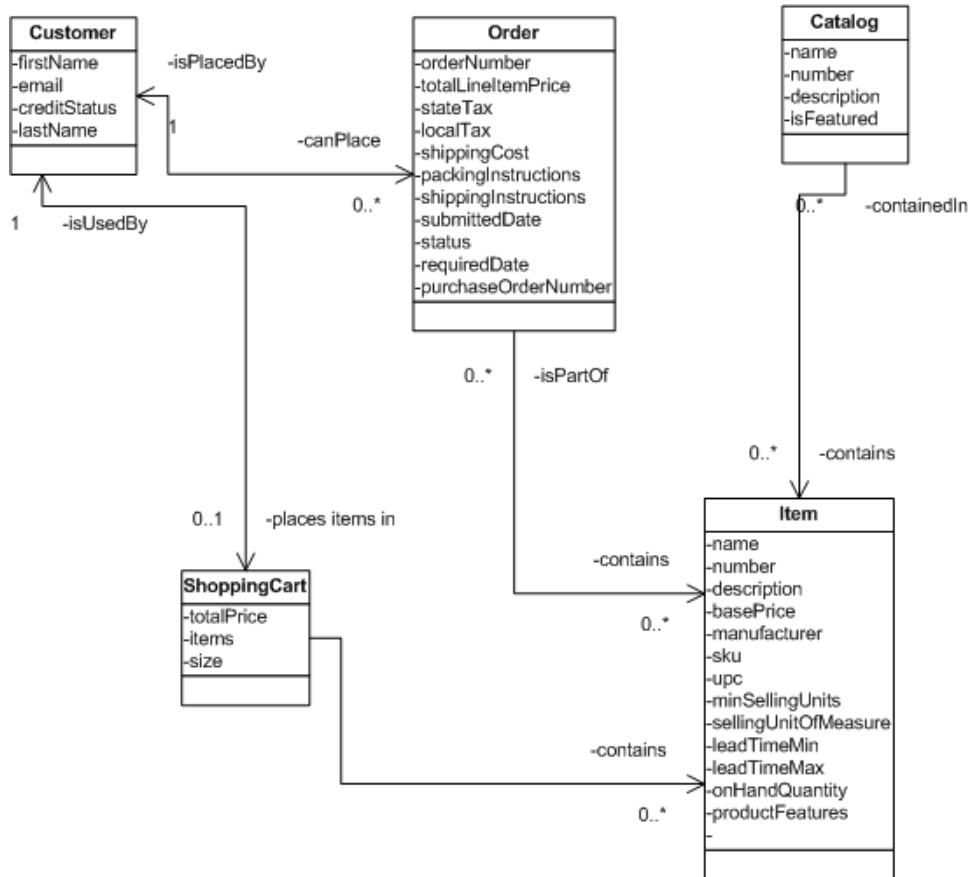


Figure 6-2. The Storefront Conceptual Model

In Figure 6-2, you'll notice that only relationships and the attributes for the entities are shown. No methods are specified during the analysis stage.

If you are familiar with Entity-Relationship (E-R) diagrams, then you shouldn't be too confused by the conceptual model. They are very similar.

The value of a conceptual model is that it clearly shows the entities that are used by problem domain. Everyone involved in the business, technical or not, should be able to look at the conceptual model and make sense of it. They should also be able to quickly point out problems with the model. For example, maybe an item can't belong to more than one catalog at a time as Figure 6-2 indicates. By examining the conceptual model, someone would be able to quickly point this out, and the analysts could make the change early. Remember, the later in the design and development cycle that a change is required, the more costly that change becomes.

The Design Model

The conceptual model is just one artifact of the analysis stage; there can be many others. In smaller development groups or on smaller projects, the conceptual model may be skipped in favor of moving to the design stage faster. The risk of doing this however is that you might leave the analysis stage without a clear and concise understanding of the requirements. The better you understand the requirements for a project, the closer the end product will be.

Even if you build a conceptual model, you still will need to create the appropriate design documents. This usually includes class diagrams, interaction diagrams, and possibly others artifacts like state diagrams. At a very minimum, your design stage artifacts should include a class diagram for the application's business objects. Figure 6-3 illustrates a class diagram based on the conceptual model for the storefront application illustrated in Figure 6-2.

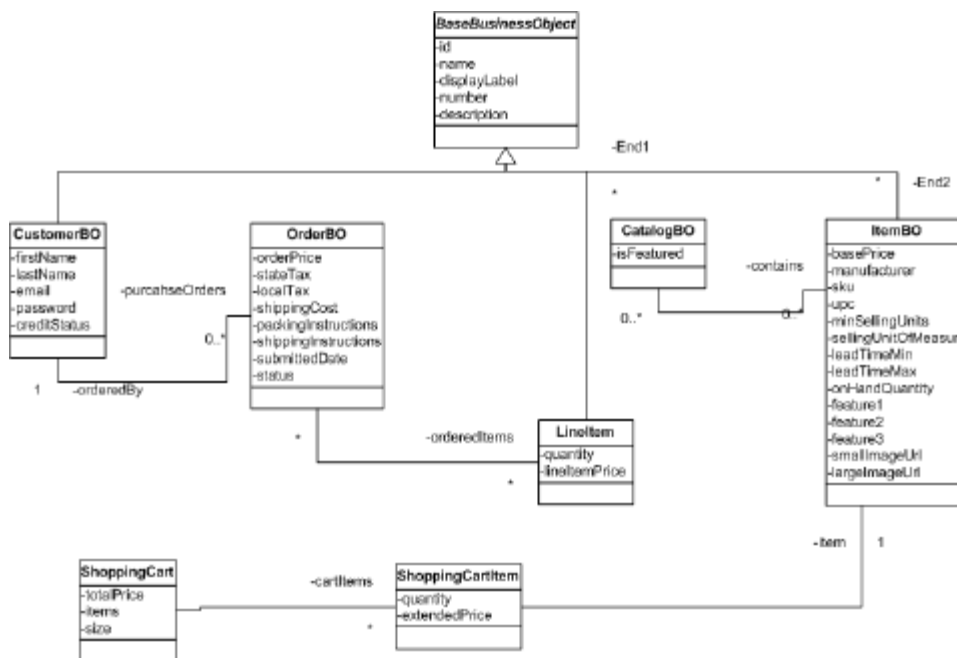


Figure 6-3. The Class Diagram for the Storefront Business Objects

The class diagram in Figure 6-3 shows the business objects used within the Storefront application. For brevity, only the attributes and relationships are shown here.

The way in which you arrive at the proper design for your application is definitely beyond the scope of this book. There are a multitude of excellent analysis and design books available. One of the more popular references is “Applying UML and Patterns” by Craig Larman.

The exact type and structure of business objects and processing rules is obviously dependant on the domain of your application, and not two applications are exactly the same. Even within a single application, it's understood that requirements do change. However, conducting a thorough analysis of the problem domain and constructing a resilient design goes a long way in protecting your application from unexpected change.

What is a Business Object?

This question may sound a little too simplistic for a book on Struts. However, the term “business object” unfortunately has many connotations. Some developers use the term when they really mean something else. A *Business Object* (BO) is simply, a software abstract of a real-world entity. It represents a person, place, thing, or concept from the business domain. So, concepts like items, orders, and customers are all business objects from the Storefront business domain.

Business objects consist of state and behavior. The *OrderBO* for example, is aware of information relating to a single customer purchase order, including price, tax, and the order status. You can also ask an order business object who the customer is and it should be available to provide this information. Having both state and behavior is one very important criterion for a class to be considered a business object. Let's examine a few other business object characteristics.

What are the Requirements for Business Objects

For a class to be considered a Business Object, there are several conditions that should be met.

- Consists of state and behavior
- Represent a person, place, thing, or concept from the business domain
- Reusable

Business objects can also be grouped into different classifications. Generally, there are three types:

- Entity Business Object
- Process Business Object
- Event Business Object

Probably the most familiar, an *Entity BO* represents a person, place, thing, or concept. They are usually extracted straight from the business domain by considering the nouns in the business. Again, these are concepts like customers, orders, items, and so on. In an EJB application, these are modeled as entity beans, thus the name. In a more traditional web application, these may be regular JavaBeans that contain the state and behavior of the business application.

The *Process BO* represents business processes or workflow tasks within an application. They are usually dependant on entity BO objects and are the verbs of the business. In an EJB application, these are normally modeled as session beans, or in some cases, message-driven beans. In a non-EJB application, these may be regular JavaBeans that contain specialized behavior to operate as a manager or controller for the application. Even though these types of business objects are used for processing workflow, they are still allowed to hold state for an application. With EJB for example, there are stateless and stateful session beans.

The final category of business object is the *Event BO*. An Event BO represents some event in the application that causes an action or results from one. These might be exceptions, alerts, timed events, and so on. In a Java Swing application for example, when you press a button, an event is raised notifying the underlying framework, so that an event handler can handle the button press.

[\[Editors: I would like your opinion on the above section. Does it add any value or should it be chopped?\]](#)

The Importance of Business Objects

Using business objects in an application has several relevant benefits. Probably the most important is that they provide common terminology and ideas that can be shared across an organization for both technical and non-technical people alike. Because they represent “real-world” concepts and ideas, they are very intuitive and should make sense to the entire organization. If multiple applications from the same business domain exist, it’s very likely that the same business objects exist across the application boundaries. This reuse of information and behavior allows for faster application development and reduces the redundancy.

Business objects also have the ability to evolve with the organization through modifications to the original object, or through proper specialization. This is very important because as an organization changes, the information and behavior must adapt and change with it.

Finally, business objects have well-defined interfaces. Not interfaces in the Java sense, but a very clear and cohesive set of functionality. The internal implementation should be hidden from the client, in order to protect the callers from changes to the implementation details. For example, suppose you had a business object that used a `java.util.ArrayList`. Instead of exposing the type `ArrayList`, you should

expose `java.util.List`. If internally, the implementation changed from `ArrayList` to `LinkedList`, the caller would not be impacted, because the client is programming against the `List` interface, not the implementation class.

By now, you should be full aware of the importance that business objects have in an organization. They are present in all but the most trivial applications. You also learned that they contain state and behavior, which in most cases, acts on that data. So the next question should be, where does that state originate from and where does it go when the application is stopped. This leads us into the topic of object persistence.

Persistence

In general, *persistence* means that the data that is input into an application, either by a human user, or by other means, will exist beyond the lifetime of the application. Even though the application may exit, or the computer itself may shutdown, the information will survive. This is obviously very important to an organization. Every small, medium, and large organization, has the need to persist data.

Persisting Business Objects

When objects are created in memory for an application, they can't stay there forever. Eventually, they must either be cleaned up or persisted to a data store. Memory is volatile and an application might crash or need to be stopped for maintenance. In Java, the garbage collector will reclaim the memory for any objects that don't have any references. Remember, the data is of the utmost importance to your organization. Without data persistence, there's no record to indicate what was ordered or who to charge, for example.

Business objects represent information that must be kept. Orders, items, and customer information, must be persisted for an application like the Storefront to be of value. Taking in a customer's order and losing it is not going to make the customer a customer for long. Once the data has been persisted, it can be retrieved and used to reconstruct the business objects.

Storing Objects into a Relational Model

Although there are many different types of data stores, relational databases are used quite frequently to store the data for an organization, especially with applications like the Storefront example. Relational databases are a necessity and are very widespread. However, there are several obstacles that must be overcome if you are going to be successful in this endeavor. One of the biggest challenges to overcome is known as the "Impedance Mismatch".

The Impedance Mismatch

Objects hold state and behavior and can be traversed through their relationships with other objects. The relational paradigm on the other hand, is based on storing data and joining sets of data, based on matching of overlapping fields. Essentially, a relational database is a very “flat” view of the data. This difference leads to a very challenging mismatch between the two worlds. Essentially, the objects must be flattened before they can be stored into a relational database. The relationships that objects have to one another must also be persisted, in order for the object hierarchy to be correctly reassembled.

There's not enough room in this chapter for a detailed tutorial on mapping objects to a relational model. Suffice to say, there are many challenges. Fortunately, there are many resources and references available to help in overcoming the challenges. A definitive source of information on how to correctly map objects to a relational database can be found in Scott Ambler's white paper titled “Mapping Objects to Relational Databases”. The white paper can be found at:

<http://www.amblysoft.com/mappingObjects.pdf>

As you'll see shortly, there are many *Object-to-Relational Mapping* (ORM) frameworks that make this job much easier for the Java developer. Using one of these frameworks doesn't completely eliminate your need for a good understanding of the problems, however the frameworks can hide many of the ugly chores that no developer wants to have to perform.

What does Struts Offer For the Model?

To be honest, the Struts framework doesn't offer much in the way of building model components, but this is probably as it should be. There are many frameworks and component models already available for dealing with the business domain of an application, including Enterprise JavaBeans, Java Data Objects (JDO), or you can use regular JavaBeans and an Object-to-Relational Mapping framework (ORM). The good news is that the Struts framework does not limit you to one particular model implementation. This chapter will present one approach and then in Chapter 13 “Struts and Enterprise JavaBeans (EJB),” we'll take a completely different approach and see how the framework is affected by this change.

ActionForms are not the Model

Many developers get confused when they read and learn about the `ActionForm` class. Although the `ActionForm` can hold state and behavior for an application, the state that it holds should be limited and constrained to the “user input” that is received from the client and should only hold it until it can be validated and transferred to the business objects for an application.

You’ve already seen why it’s important to use business objects in an application. Business objects can be persisted and should contain the business logic for an application. They should also be reusable. This set of criteria does not match up well when compared against `ActionForms`. For one, the `ActionForm` class is most certainly tied to the Struts framework and explicitly to a web container, since it imports `javax.servlet` packages. It would be very difficult or problematic to port `ActionForm` classes to a different type of framework, like a Swing application. There are also issues that would make persisting `ActionForm` classes difficult.

As the next chapter will indicate, `ActionForms` are designed to capture the HTML data from a client, allow “presentation validation” to occur, and to provide a transport vehicle for the data back to the more persistent business object. Chapter 7 will provide more details about the `ActionForm` class and its limited role in the framework.

Before we discuss the model implementation for the Storefront application, it’s a good time to discuss Struts works with JavaBeans.

Building the Storefront Model

After all of this discussion of what constitutes a model for a Struts application, it’s finally time to apply some hands on using the storefront application as the business domain. Obviously, the storefront is a fictitious example and doesn’t represent a complete model for what a “real” e-commerce application would need to support. However, it does provide enough of an object model for you to understand the semantics of this chapter.

Accessing a Relational Database

The state of the Storefront application will be persisted using a relational database. This is in fact, how it would be done if Storefront were a real application. Of course, an ERP system is often used in conjunction, but many e-commerce applications use a relational

database closer to the front-end for performance and ease of development. When both are deployed in an enterprise, there's usually a middleware service to keep the data between the two synchronized, either in real-time or using batch mode.

As you probably are aware, there are many relational databases that one can choose from. Depending on your particular criteria, you can choose from one of several major database vendors or if your requirements don't call for such a large and expensive implementation, you can choose to use one of the cheaper products on the market. Since we will not be building out every aspect of the application and our intended user load is small, our requirements for a database are not very stringent. With that being said, the database-specific examples that will be shown should be fine for most database platforms. If you understand SQL *Data Definition Language* (DDL), you can tweak the DDL for the database that's available to you.

We have quite a bit of work to do before we can start using the Storefront model. In fact, the following tasks need to be completed before we are even ready to involve the Struts framework:

- Create the Business Objects for the Storefront application
- Create the Database for the Storefront application
- Map the Business Object to the Database
- Test that the Business Objects can be persisted in the Database

As you can see, none of these tasks mention the Struts framework. This is intentional because you should approach this part of the development phase without a particular "client" in mind. The Struts Storefront web application is just one potential type of client to the business objects. If designed and coded properly, many different types may be used. The business objects are used to query and persist information regarding the Storefront business, it should have no coupling to a presentation client.

To help insulate the Struts framework from changes that may occur in the business objects, we will also look at using the *Business Delegate Pattern* within the Storefront application. The Business Delegate acts as a client-side business abstraction. It hides the implementation of the actual business service, which helps to reduce the coupling between the client and the business objects.

Creating the Storefront Business Objects

Business objects contain data and behavior, this we know. They are a virtual representation of one or more records within a database. In the Storefront application for example, an `OrderBO` object represents a physical purchase order placed by a customer. It also contains the business logic that helps to ensure that the data is valid and remains valid.

Where does Business Validation Belong?

Deciding where to put your validation logic in a Struts application can be frustrating. On one hand, it seems like it belongs within the framework itself, since this is the first place that the user data can be obtained and validated. The problem with placing business logic validation within the Action or ActionForm classes is that the validation then becomes coupled to the Struts framework. This prevents the validation logic from being reused by any other clients.

Although business logic doesn't belong in the Struts framework, there is a different type of validation that can and should occur within the framework. This type of validation is commonly referred to as *Presentation Validation*.

Presentation validation or "input validation" can be grouped into three distinct categories:

- Lexical
- Syntactic
- Semantic

Lexical validation checks to make sure data is well formed. For example, is the quantity value an integer? Syntactic validation goes one step further and makes sure that values made from a composite are valid and well formed. Date fields in a browser are typically accepted as *month/day/year* values. Syntactic validation ensures that the value entered is in the proper format. It doesn't necessarily ensure that the values make a valid date, but rather the format of the value is correct. Ensuring that the date entered is valid and is meaningful is the job for semantic validation. Semantic validation ensures that the values entered have meaning for the application. For example, putting a quantity value of *-3* in the order quantity field for an item is lexically and syntactically valid, but not semantically.

Presentation validation belongs within the Struts framework, but business validation does not. The business objects have the final responsibility of ensuring the any data inserted into the database is valid and therefore, it should have the rules necessary to perform this duty.

The first step is to create the business objects that we'll need to interact with. For this implementation, they will just be regular JavaBean objects. Many component models are specific to a single implementation. Entity beans for example, will only work within an EJB container. For this example, the Storefront business objects will not be specific to a particular implementation. If later, we wanted to use these same business objects with an

EJB container, we could wrap them with entity beans or just delegate the call from a session bean method to one of these objects. In Chapter 13 “Struts and Enterprise JavaBeans”, we’ll show an example of this can be done without impacting the Storefront application. For now, we’ll concentrate on building business objects that are not dependant on any particular container implementation.

Because all the business objects share several common properties, we are going to create an abstract super class for the business objects. Every business object will be a subclass of the **BaseBusinessObject** class shown in Example 6-1.

```
Example 6-1. The BaseBusinessObject is the super class for all business objects
package com.oreilly.struts.storefront.businessobjects;

/**
 * An abstract super class that many business objects will extend.
 */
abstract public class BaseBusinessObject implements java.io.Serializable {
    private Integer id;
    private String displayLabel;
    private String description;

    public Integer getId() {f
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDisplayLabel(String label) {
        this.displayLabel = label;
    }

    public String getDisplayLabel() {
        return displayLabel;
    }
}
```

The **BaseBusinessObject** prevents each business object from needing to declare the same properties over and over. We can also put common business logic here, if the opportunity presents itself.

Example 6-1 shows the **OrderBO** business object that is represents a customer purchase order in the Storefront application. There’s nothing that special about the **OrderBO**

class. It's an ordinary JavaBean object that other than the `recalculatePrice ()` method, just provides setter and getter methods for the order properties.

Example 6-2. The OrderBO business object represents an order placed by a customer

```
package com.oreilly.struts.storefront.businessobjects;

import java.math.BigDecimal;
import java.sql.Timestamp;
import java.util.Iterator;
import java.util.List;
import java.util.LinkedList;

/**
 * The Order Business Object, which represents a purchase order that a customer
 * has or is about to place.
 */
public class OrderBO extends BaseBusinessObject {
    // A list of line items for the order
    private List lineItems;
    // The customer who placed the order
    private CustomerBO customer;
    // The current price of the order
    private double totalPrice;
    // The id of the customer
    private Integer customerId;
    // Whether the order is inprocess, shipped, canceled, etc...
    private String orderStatus;
    // The date and time that the order was received
    private Timestamp submittedDate;

    /**
     * Default NoArg Constructor
     */
    public OrderBO() {
        super();
        // Initialize the line items as a linked list to keep them in order
        lineItems = new LinkedList();
    }

    /**
     * Additional constructor that takes the necessary arguments to initialize
     */
    public OrderBO( Integer id, Integer custId, String orderStatus,
        Timestamp submittedDate, double price ){
        this.setId(id);
        this.setCustomerId(custId);
        this.setOrderStatus(orderStatus);
        this.setSubmittedDate(submittedDate);
        this.setTotalPrice(price);
    }

    public void setCustomer( CustomerBO owner ){
        customer = owner;
    }
}
```



```
public CustomerBO getCustomer(){
    return customer;
}

public double getTotalPrice(){
    return this.totalPrice;
}

private void setTotalPrice( double price ){
    this.totalPrice = price;
}

public void setLineItems( List lineItems ){
    this.lineItems = lineItems;
}

public List getLineItems(){
    return lineItems;
}

public void addLineItem( LineItemBO lineItem ){
    lineItems.add( lineItem );
}

public void removeLineItem( LineItemBO lineItem ){
    lineItems.remove( lineItem );
}

public void setCustomerId(Integer customerId) {
    this.customerId = customerId;
}

public Integer getCustomerId() {
    return customerId;
}

public void setOrderStatus(String orderStatus) {
    this.orderStatus = orderStatus;
}

public String getOrderStatus() {
    return orderStatus;
}

public void setSubmittedDate(Timestamp submittedDate) {
    this.submittedDate = submittedDate;
}

public Timestamp getSubmittedDate() {
    return submittedDate;
}

/**
 * Recalculate the price of the entire order. If one or more of the line
 * items doesn't have a price for some reason, set the totalPrice of the
```

```
* order to null, since it can't be truly calculated. By setting it to null,
* even if a few of the line items have a price, this will alert someone
* that there's a problem. This should also prevent orders from being
* accepted by the system when there was a problem with one or more line
* items. If this wasn't done in this manner, a customer might receive
* goods for free.
*/
private void recalculatePrice(){
    double totalPrice = 0.0;

    if ( getLineItems() != null ){
        Iterator iter = getLineItems().iterator();
        while( iter.hasNext() ){
            // Get the price for the next line item and make sure it's not null
            Double lineItemPrice = ((LineItemBO)iter.next()).getUnitPrice();
            // Check for an invalid lineItem. If found, return null right here
            if (lineItemPrice == null){
                return;
            }
            // It's not null, so add it to the total
            totalPrice += lineItemPrice.doubleValue();
        }
        // Set the price for the order from the calculated value
        setTotalPrice( totalPrice );
    }
}
```

We won't show all of the business objects here, they all have similar implementations to the `OrderBO` class.

When designing your business objects, you should not give any thought to how they will be mapped to the database. There will be plenty of time for that. Don't be afraid to use object-oriented techniques such as inheritance and polymorphism, just as you would with any other object model. The `BaseBusinessObject` in Example 6-1 will not actually be mapped to a table in the database, but its properties will get mapped with the respective subclasses. Even though most persistence mapping frameworks support multiple approaches to mapping inheritance in the database, by adding the properties to each concrete table, the number of SQL joins are fewer and this has a positive impact on performance.

The Storefront Data Model

Once all of the business objects have been created for the Storefront application, we need to create a database model and schema for them. The details of creating a database schema for the Storefront application are beyond the scope of the lesson of this chapter and obviously the book. Seemingly, it's easy to throw a bunch of tables into a database and add columns to them. However, it's quite another thing to understand the tradeoffs

between database normalization and issues that surface due to the object-relational mismatch that was discussed earlier in the chapter.

If the application is small enough, it's quite true that almost anyone can create a database schema, especially with the tools that are available from the database vendors and third-party sources. If your schema is more than just a few tables or the complexity of foreign keys, triggers, and indexes are high, then it's best to leave creating a schema to the experts. The Storefront schema is quite small, mainly because we've chosen to implement only a portion of what would normally be required. Figure 6-4 shows the data model that will be implemented for the storefront application.

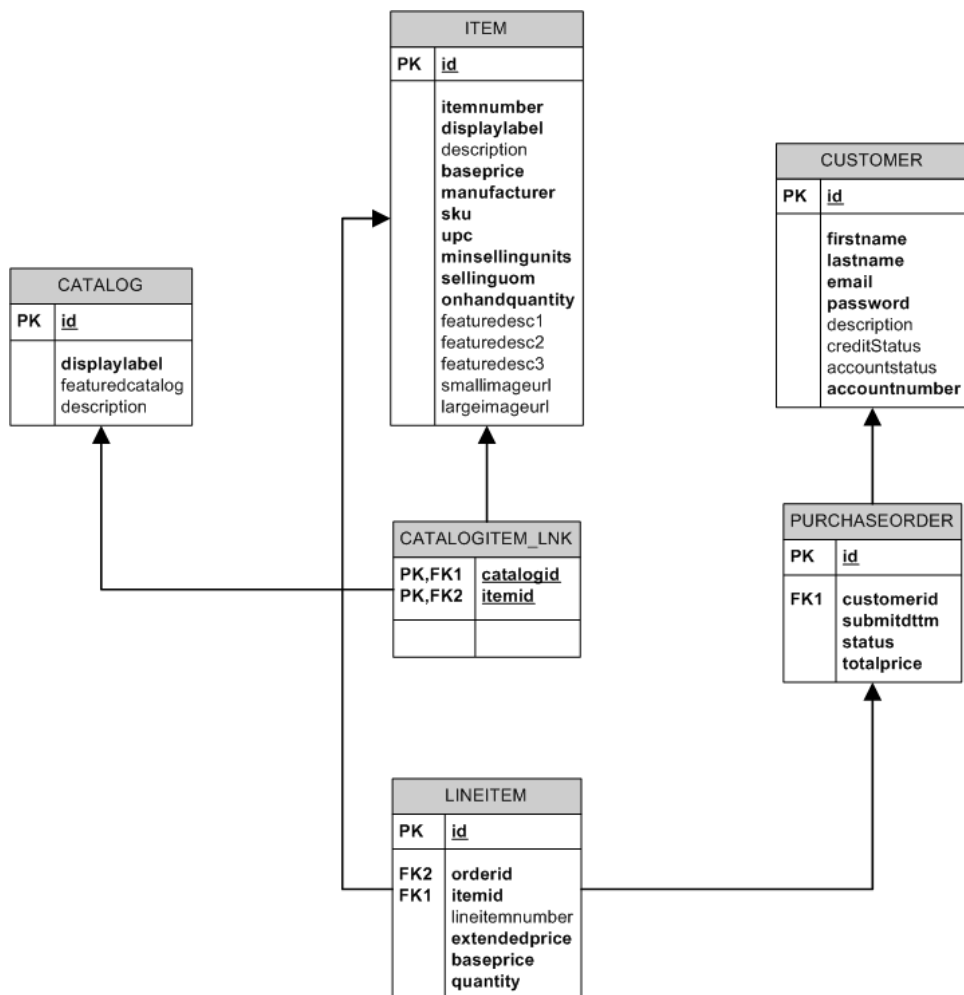


Figure 6-4. The Storefront Data Model

The table definitions in Figure 6-4 are fairly self-explanatory. There are several items of interest that should be pointed out. The first is that every table, except for the many-to-

many link table `CATALOGITEM_LNK`, has been assigned an object identifier. An object identifier (OID) simplifies the navigation between objects. OIDs should have no business meaning at all. Values that are based on business semantics will sooner or later change and basing your keys on values that change is very problematic. In the database world, using the OID strategy is known as using surrogate keys.

To generate the schema for the data model shown in Figure 6-4, we need to create the DDL. The SQL DDL is used to create Meta objects in the database. The Storefront SQL DDL that will create the set of tables in Figure 6-4 is shown in Example 6-3.

Example 6-3. The Storefront SQL DDL

```
# The SQL DDL for the Storefront Application
# (Apache Struts by O'Reilly)
# Chuck Cavaness

# Execute the next line if you need to clear the storefront database
DROP DATABASE storefront;

# Creates the initial database
CREATE DATABASE storefront;

# Make sure you are creating the tables in the storefront tablespace
use storefront;

CREATE TABLE CATALOG(
  id int NOT NULL,
  displaylabel varchar(50) NOT NULL,
  featuredcatalog char(1) NULL,
  description varchar(255) NULL
);

ALTER TABLE CATALOG ADD
  CONSTRAINT PK_CATALOG PRIMARY KEY(id);

CREATE TABLE CUSTOMER (
  id int NOT NULL ,
  firstname varchar(50) NOT NULL,
  lastname varchar(50) NOT NULL,
  email varchar(50) NOT NULL ,
  password varchar(15) NOT NULL,
  description varchar(255) NULL,
  creditStatus char(1) NULL,
  accountstatus char(1) NULL,
  accountnumber varchar(15) NOT NULL
);

ALTER TABLE CUSTOMER ADD
  CONSTRAINT PK_CUSTOMER PRIMARY KEY(id);

CREATE TABLE ITEM (
  id int NOT NULL ,
  itemnumber varchar (255) NOT NULL ,
```

```
displaylabel varchar(50) NOT NULL,
description varchar (255) NULL,
baseprice decimal(9,2) NOT NULL,
manufacturer varchar (255) NOT NULL,
sku varchar (255) NOT NULL,
upc varchar (255) NOT NULL,
minsellingunits int NOT NULL,
sellinguom varchar (255) NOT NULL,
onhandquantity int NOT NULL,
featuredesc1 varchar (255) NULL,
featuredesc2 varchar (255) NULL,
featuredesc3 varchar (255) NULL,
smallimageurl varchar (255) NULL,
largeimageurl varchar (255) NULL
)

ALTER TABLE ITEM ADD
    CONSTRAINT PK_ITEM PRIMARY KEY(id);

CREATE TABLE CATALOGITEM_LNK(
    catalogid int NOT NULL ,
    itemid int NOT NULL
)

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT PK_CATALOGITEM_LNK PRIMARY KEY(catalogid, itemid);

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT FK_CATALOGITEM_LNK_CATALOG FOREIGN KEY
        (catalogid) REFERENCES CATALOG(id);

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT FK_CATALOGITEM_LNK_ITEM FOREIGN KEY
        (itemid) REFERENCES ITEM(id);

CREATE TABLE PURCHASEORDER (
    id int NOT NULL,
    customerid int NOT NULL,
    submitdtm timestamp NOT NULL ,
    status varchar (15) NOT NULL,
    totalprice decimal(9,2) NOT NULL,
)

ALTER TABLE PURCHASEORDER ADD
    CONSTRAINT PK_PURCHASEORDER PRIMARY KEY(id);

ALTER TABLE PURCHASEORDER ADD
    CONSTRAINT FK_PURCHASEORDER_CUSTOMER FOREIGN KEY
        (customerid) REFERENCES CUSTOMER(id);

CREATE TABLE LINEITEM (
    id int NOT NULL,
    orderid int NOT NULL,
    itemid int NOT NULL,
```

```
    lineitemnumber int NULL,  
    extendedprice decimal(9, 2) NOT NULL,  
    baseprice decimal(9, 2) NOT NULL,  
    quantity int NOT NULL  
)  
  
ALTER TABLE LINEITEM ADD  
    CONSTRAINT PK_LINEITEM PRIMARY KEY(id);  
  
ALTER TABLE LINEITEM ADD  
    CONSTRAINT FK_LINEITEM_ORDER FOREIGN KEY  
        (orderid) REFERENCES PURCHASEORDER(id);  
  
ALTER TABLE LINEITEM ADD  
    CONSTRAINT FK_LINEITEM_ITEM FOREIGN KEY  
        (itemid) REFERENCES ITEM(id);
```

The DDL shown in Example 6-3 has been tested on Oracle 8.1.7 and Microsoft SQL Server 2000. If you plan to use it with other database platforms, it might be necessary to modify the **ALTER** statements. For example, due to the limitations of foreign keys with *MySQL*, you may have to eliminate the **FOREIGN KEY** statements entirely. The only parts that are absolutely necessary to run the example are the **CREATE TABLE** sections and the primary keys, which all databases should accept.

Once you have executed the DDL from Example 6-3, you will need to insert some data for the tables. To make it easier for you when it comes time to run the example, sample data has been provided with the source code for this Chapter. The sample data is located in a file called *storefront-data.sql* located in the **source/sql** directory for this chapter. To import the data, just open up a query tool for the proper database and execute the SQL within this file. If your database vendor provides a batch mode, you can also import it that way as well. The data will need to be in the database before the Storefront application will work properly.

Mapping the Business Objects to the Database

When it comes time to connecting, or mapping, the business objects to the database, there are a variety of approaches that you can choose from. Which one you choose depends on several factors that may change from application to application and situation to situation. A few of the approaches to choose from are:

- Use Straight JDBC calls
- Use a “Home-Grown” ORM approach (also known the roll-your-own approach)
- Use a Proprietary Object-Relational Mapping Framework
- Use a Non-Intrusive, Non-Proprietary Object-Relational Mapping Framework

- **Use an Object Database**

Keeping in mind that some tasks are better to do in-house and others are better left to the experts, building a Java persistence mechanism is one that typically you should avoid doing. Remember that the point of building an application is to solve a business problem. In many cases, you are better off acquiring a persistence solution from a third-party.

There are many more complicated issues that must be dealt with than just issuing a SQL select statement through JDBC, including transactions, nested transactions, primary key increments, caching, and connecting pooling. Building a mapping and persistence framework is an entire project in and of itself. You shouldn't be spending valuable time and resources on something that isn't the core business problem. The next section lists several solutions that are available.

Object to Relational Mapping Frameworks (ORM)

There is an overwhelmingly number of ORM products available for you to choose from for a project. Some of them are commercially available and have a cost that is near or exceeds that of most application servers. Others are open source and have no monetary cost associated with them. Table 6-1 presents several of the commercial and non-commercial solutions that you can choose from.

Table 6-1. Object-to-Relational Mapping Frameworks

Product	URL
TopLink	http://www.objectwave.com/html/Main.html
CocoBase	http://www.cocobase.com
Torque	http://jakarta.apache.org/turbine/torque/index.html
ObJectRelationalBridge	http://objectbridge.sourceforge.net
FrontierSuite	http://www.objectfrontier.com
Castor	http://castor.exolab.org
FreeFORM	http://www.chimu.com/projects/form
Espresso	http://www.jcorporate.com
JRelationalFramework	http://jrf.sourceforge.net
VBSF	http://www.objectmatter.com
JGrinder	http://sourceforge.net/projects/jgrinder

Although Table 6-1 is not an exhaustive list of available products, it does present many solutions to choose from. Regardless of whether you select a commercial or non-commercial product, you should make sure that the mapping framework implementation does not “creep” into your application. If you remember the idea from Figure 6-1 at the beginning of this chapter, dependencies should always go down the layers and there

should not be a top layer depending on the persistence framework. It's even advantageous to keep the business objects ignorant about how they are being persisted. Some persistence frameworks force you to import their classes and interfaces in order for it to work. This is very problematic if you ever need to change your persistence mechanism. Later in the chapter, you'll see how the use of the Business Delegate pattern and a that we'll introduce called *Data Access Object* (DAO), can be used to limit the intrusion of the persistence framework.

You can find a product comparison between these and other mapping products at <http://www.object-relational.com/object-relational.html>. There is a cost for the compiled information, but if your application is of decent size, it may be money well spent.

Another thing to be careful of is that a few of the persistence frameworks need to alter the Java byte code of the business objects after they are compiled. Depending on how you feel about this, it could introduce some issues. Just make sure you fully understand how a persistence framework needs to interact with your application before investing time and resources into using it.

The Storefront Persistence Framework

We could have chosen almost any solution from Table 6-1 and successfully mapped the storefront business objects to the database. Our requirements are not that stringent and the model isn't that complicated. Several were evaluated and one was finally selected for use in the Storefront example. The selection process was very informal and quick, an approach you should not follow for any serious project. The criteria that they were judge against were:

- The amount of intrusion the persistence mechanism needed
- How good was the available documentation
- The cost of the solution

Cost was a big factor. We needed a solution that could be distributed with the book and you would not incur any monetary cost. All of the solutions evaluated for this example performed pretty well and were relatively easy to use, but the product *ObjectRelationalBridge* was finally chosen and that's what the Storefront example will use.

Just because this solution was chosen for this example, don't assume that it will be the best solution for your application. Make sure to take the necessary time and evaluate the products on your specific set of criteria.

The documentation for *ObjectRelationalBridge* is fairly good, considering its open source and documentation for open source projects tend to be one of the last tasks to

complete. Essentially, the entire mapping of the business objects to the database tables take place in a single XML file called *repository_user.xml*. The file is parsed at run time by the mapping framework and used to execute SQL to the database. The portion of the Storefront mapping file that maps the customer business object is shown in Example 6-4.

Example 6-4. The mapping XML for the CustomerBO Class

```
<ClassDescriptor id="120">
  <class.name>com.oreilly.struts.storefront.businessobjects.CustomerBO</class.name>
  <table.name>CUSTOMER</table.name>

  <FieldDescriptor id="1">
    <field.name>id</field.name>
    <column.name>id</column.name>
    <jdbc.type>INTEGER</jdbc.type>
    <PrimaryKey>true</PrimaryKey>
    <autoincrement>true</autoincrement>
  </FieldDescriptor>

  <FieldDescriptor id="2">
    <field.name>firstName</field.name>
    <column.name>firstname</column.name>
    <jdbc.type>VARCHAR</jdbc.type>
  </FieldDescriptor>

  <FieldDescriptor id="3">
    <field.name>lastName</field.name>
    <column.name>lastname</column.name>
    <jdbc.type>VARCHAR</jdbc.type>
  </FieldDescriptor>

  <FieldDescriptor id="4">
    <field.name>email</field.name>
    <column.name>email</column.name>
    <jdbc.type>VARCHAR</jdbc.type>
  </FieldDescriptor>

  <FieldDescriptor id="5">
    <field.name>password</field.name>
    <column.name>password</column.name>
    <jdbc.type>VARCHAR</jdbc.type>
  </FieldDescriptor>

  <FieldDescriptor id="6">
    <field.name>accountStatus</field.name>
    <column.name>accountstatus</column.name>
    <jdbc.type>CHAR</jdbc.type>
  </FieldDescriptor>

  <FieldDescriptor id="7">
    <field.name>creditStatus</field.name>
    <column.name>creditstatus</column.name>
    <jdbc.type>CHAR</jdbc.type>
  </FieldDescriptor>
```

```
<CollectionDescriptor id="1">
  <cdfield.name>submittedOrders</cdfield.name>
  <items.class>com.oreilly.struts.storefront.businessobjects.OrderBO</items.class>
  <inverse_fk_descriptor_ids>2</inverse_fk_descriptor_ids>
</CollectionDescriptor>
</ClassDescriptor>
```

The rest of the mappings are mapped in a similar manner. Once all of the mappings are specified in the XML file, you must configure the database connection information to allow the JDBC driver to connect to the correct database. With the *ObjectRelationalBridge* product, you configure the connection information in the *repository.xml* file. This is shown in Example 6-5.

Example 6-5. The repository.xml file contains the Database Connection Information

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- defining entities for include-files -->
<!DOCTYPE MappingRepository SYSTEM "repository.dtd" [
<!-- ENTITY user SYSTEM "repository_user.xml" -->
<!-- ENTITY junit SYSTEM "repository_junit.xml" -->
<!-- ENTITY internal SYSTEM "repository_internal.xml" -->
]>

<MappingRepository>
  <JdbcConnectionDescriptor id="default">
    <dbms.name>MsSQLServer2000</dbms.name>
    <jdbc.level>1.0</jdbc.level>
    <driver.name>com.ddtek.jdbc.sqlserver.SQLServerDriver</driver.name>
    <url.protocol>jdbc</url.protocol>
    <url.subprotocol>datadirect:sqlserver</url.subprotocol>
    <url.dbalias>
      //localhost:1433;DatabaseName=storefront;SelectMethod=cursor
    </url.dbalias>
    <user.name>sa</user.name>
    <user.passwd></user.passwd>
  </JdbcConnectionDescriptor>

  <!-- include user defined mappings here -->
  &user;

  <!-- include ojb internal mappings here -->
  &internal;
</MappingRepository>
```

You need to configure the settings in this file for your specific environment. That's really all there is to configuring the persistence framework for your application. To initialize the framework within your application, all you need to do is call three methods. This will be shown later in the section.

The *ObjectRelationalBridge* offers two different APIs that you can use, the *PersistenceBroker* and the *ODMG* implementation. You can read

the documentation to understand the differences, but the ODMG is much more powerful, including having an Object Query Language (OQL). However, with that extra functionality comes a little more complexity. The Storefront application will use the ODMG API due to its better features. The product also allows the framework to be used in stand-alone mode and in client/server mode. The client/server mode is useful when there are multiple servers in the same environment. For the Storefront example, we have chosen to use it in stand-alone mode, since the example application will not ever need to scale for multiple servers. This means that the persistence framework is running within the same JVM as the Storefront web application itself.

There's not enough room in this chapter for a better explanation of the framework than what's been given. For a better explanation, review the documentation for the product at <http://objectbridge.sourceforge.net>.

Don't forget that you will need to be sure to have an appropriate Database and a JDBC driver in your classpath.

The Business Delegate and DAO Patterns in Action

The final piece of the puzzle is to create a service interface that the Storefront Action classes can use, instead of interacting with the persistence framework directly. Again, the idea is to decouple the persistence from as much of the application as possible. Before we show the details of how we are going to accomplish this for the Storefront example, we need to briefly discuss the Data Access Object (DAO) pattern.

The purpose of the DAO pattern is to decouple the business logic of an application from the data access logic. In the case where a persistence framework is being used, the pattern should help to decouple the business objects from it. A secondary goal is to allow the persistence implementation to easily be replaced with another, without negatively affecting the business objects.

There are actually two independent design patterns contained within the DAO, the Bridge and Adaptor, both of which are structural design patterns explained in "Design Patterns: Elements of Reusable Object-Oriented Software".

For the Storefront application, we are going to combine the DAO and Business Delegate patterns to insulate the **Action** and business object classes from the persistence implementation. The abstract approach is shown in Figure 6-5.

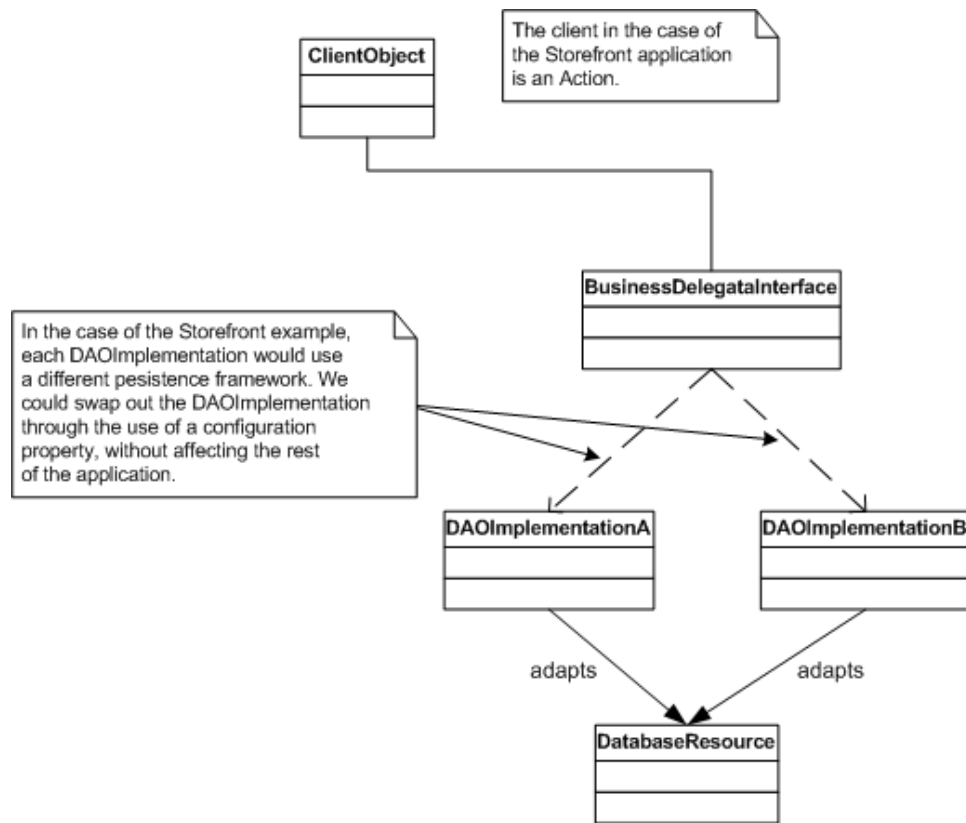


Figure 6-5. The Business Delegate and the DAO Patterns combined

The client object in Figure 6-5 represents the Struts **Action** classes. They will acquire a reference to a service interface, which is referred to in the diagram as the **BusinessDelegataInterface**. The Storefront business interface is shown in Example 6-6.

Example 6-6. The Storefront business interface

```
package com.oreilly.struts.storefront.service;

import java.util.List;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.catalog.view.ItemSummaryView;
import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;
import com.oreilly.struts.storefront.framework.security.IAuthentication;
/**
 * The business interface for the Storefront Application. It defines all
 * of the methods that a client may call on the Storefront application.
 *
 * This interface extends the IAuthentication interface to create a
 * cohesive interface for the Storefront application.
 */
```

```
public interface IStorefrontService extends IAuthentication {  
    public List getFeaturedItems() throws DatastoreException;  
    public ItemDetailView getItemDetailView( String itemId )  
        throws DatastoreException;  
}
```

The `IStorefrontService` interface in Example 6-6 defines all of the methods a client may call on the Storefront application. In our case, the client will be the set of Action classes in the Storefront application. The `IStorefrontService` is designed so that there is not a web dependency. It's feasible that other types of clients could use this same service.

The implementation for the business interface is shown in Figure 6-5 as `DAOImplementationA` and `DAOImplementationB`. We could swap out persistence implementations as long as the new implementation implemented the `IStorefrontService` interface. No client would be affected because they are programmed against the interface, not the implementation.

The persistence implementation for the chosen persistence product, *ObjectRelationalBridge*, is shown in Example 6-7.

Example 6-7. The Storefront service implementation class

```
package com.oreilly.struts.storefront.service;  
  
import java.sql.Timestamp;  
import java.util.List;  
import java.util.ArrayList;  
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;  
import com.oreilly.struts.storefront.catalog.view.ItemSummaryView;  
import com.oreilly.struts.storefront.framework.security.IAuthentication;  
import com.oreilly.struts.storefront.customer.view.UserView;  
import com.oreilly.struts.storefront.businessobjects.*;  
// Import the exceptions used  
import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;  
import com.oreilly.struts.storefront.framework.exceptions.InvalidLoginException;  
import com.oreilly.struts.storefront.framework.exceptions.ExpiredPasswordException;  
import com.oreilly.struts.storefront.framework.exceptions.AccountLockedException;  
  
// Import the implementation specific packages  
import org.odmg.*;  
import ojb.odmg.*;  
  
public class StorefrontServiceImpl implements IStorefrontService{  
    // Implementation specific references  
    Implementation odmg = null;  
    Database db = null;  
  
    /**  
     * Create the service, which includes initializing the persistence
```

```
* framework.
*/
public StorefrontServiceImpl() throws DatastoreException {
    super();
    init();
}
/**
 * Return a list of items that are featured.
 */
public List getFeaturedItems() throws DatastoreException {
    // Start a transaction
    Transaction tx = odmg.newTransaction();
    tx.begin();

    List results = null;
    try{
        OQLQuery query = odmg.newOQLQuery();
        // Set the OQL select statement
        query.create( "select featuredItems from " + ItemBO.class.getName() );
        results = (List)query.execute();
        tx.commit();
    }catch( Exception ex ){
        // Rollback the transaction
        tx.abort();
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
    }
    int size = results.size();
    List items = new ArrayList();
    for( int i = 0; i < size; i++ ){
        ItemBO itemBO = (ItemBO)results.get(i);
        ItemSummaryView newView = new ItemSummaryView();
        newView.setId( itemBO.getId().toString() );
        newView.setName( itemBO.getDisplayLabel() );
        newView.setUnitPrice( itemBO.getBasePrice() );
        newView.setSmallImageURL( itemBO.getSmallImageURL() );
        newView.setProductFeature( itemBO.getProductFeature() );
        items.add( newView );
    }
    return items;
}
/**
 * Return an detailed view of an item based on the itemId argument.
 */
public ItemDetailView getItemDetailView( String itemId )
throws DatastoreException{
    // Start a transaction
    Transaction tx = odmg.newTransaction();
    tx.begin();

    List results = null;
    try{
        OQLQuery query = odmg.newOQLQuery();

        // Set the OQL select statement
```

```
String queryStr = "select item from " + ItemBO.class.getName();
queryStr += " where id = $1";
query.create(queryStr);
query.bind(itemId);

// Execute the transaction
results = (List)query.execute();
tx.commit();
} catch( Exception ex ){
    // Rollback the transaction
    tx.abort();
    ex.printStackTrace();
    throw DatastoreException.datastoreError(ex);
}

//
if (results.isEmpty() ){
    throw DatastoreException.objectNotFound();
}

ItemBO itemBO = (ItemBO)results.get(0);

// Build a ValueObject for the Item
ItemDetailView view = new ItemDetailView();
view.setId( itemBO.getId().toString() );
view.setDescription( itemBO.getDescription() );
view.setLargeImageUrl( itemBO.getLargeImageUrl() );
view.setName( itemBO.getDisplayLabel() );
view.setProductFeature( itemBO.getFeature1() );
view.setUnitPrice( itemBO.getBasePrice() );
view.setTimeCreated( new Timestamp(System.currentTimeMillis()) );
view.setModelNumber( itemBO.getModelNumber() );
return view;
}

/**
 * Authenticate the user's credentials and either return a UserView for the
 * user or throw one of the security exceptions.
 */
public UserView authenticate(String email, String password) throws
    InvalidLoginException,ExpiredPasswordException,AccountLockedException,
    DatastoreException {

    // Start a transaction
    Transaction tx = odmng.newTransaction();
    tx.begin();

    // Query the database for a user that matches the credentials
    List results = null;
    try{
        OQLQuery query = odmng.newOQLQuery();
        // Set the OQL select statement
        String queryStr = "select customer from " + CustomerBO.class.getName();
        queryStr += " where email = $1 and password = $2";
        query.create(queryStr);
```

```
        // Bind the input parameters
        query.bind( email );
        query.bind( password );

        // Retrieve the results and commit the transaction
        results = (List)query.execute();
        tx.commit();
    }catch( Exception ex ){
        // Rollback the transaction
        tx.abort();
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
    }

    // If no results were found, must be an invalid login attempt
    if ( results.isEmpty() ){
        throw new InvalidLoginException();
    }

    // Should only be a single customer that matches the parameters
    CustomerBO customer = (CustomerBO)results.get(0);

    // Make sure the account is not locked
    String accountStatusCode = customer.getAccountStatus();
    if ( accountStatusCode != null && accountStatusCode.equals( "L" ) ){
        throw new AccountLockedException();
    }

    // Populate the Value Object from the Customer business object
    UserView userView = new UserView();
    userView.setId( customer.getId().toString() );
    userView.setFirstName( customer.getFirstName() );
    userView.setLastName( customer.getLastName() );
    userView.setEmailAddress( customer.getEmail() );
    userView.setCreditStatus( customer.getCreditStatus() );

    return userView;
}

/**
 * Log the user out of the system.
 */
public void logout(String email){
    // Do nothing with right now, but might want to log it for auditing reasons
}

/**
 * Opens the database and prepares it for transactions
 */
private void init() throws DatastoreException {
    // get odmng facade instance
    odmng = OJB.getInstance();
    db = odmng.newDatabase();
    //open database
    try{
```



```
        db.open("repository.xml", Database.OPEN_READ_WRITE);
    } catch( Exception ex ){
        throw DatastoreException.datastoreError(ex);
    }
}
```

The service implementation provides all of the required methods of the `IStorefrontService` interface. Since the `IStorefrontService` interface extends the `IAuthentication` interface, the `StorefrontServiceImpl` class must also implement the security methods. Again, notice that the implementation knows nothing about the Struts framework or web containers in general. This allows it to be reused across many different types of applications. This was our goal when we set out at the beginning of this chapter.

We mentioned earlier that we have to call a few methods of the *ObjectRelationalBridge* framework, so that the mapping XML can be parsed and the connections to the database made ready. This initialization is shown in the `init()` method in Example 6-7. When the constructor of this implementation is called, the XML file is loaded and parsed. Upon successful completion of the constructor, the persistence framework is ready to be called.

The constructor needs to be called by the client. In the case of the Storefront application, we could configure it as a Struts plugin, a servlet filter if we were using a 2.3 container, or just create it in the `init()` method of the `ActionServlet`. The first two choices would force the `StorefrontServiceImpl` class to become coupled to the servlet container and although this might not be a huge problem, we are going to take the looser coupling route and just instantiate it in the servlet `init()` method and place the object into the `ServletContext`. This helps to enforce the idea that upper layers should be dependent on lower, but not the other way around.

The `init()` method of the `ActionServlet` is shown in the following fragment.

```
public void init() throws ServletException {
    // Make sure to always call the super's init() first
    super.init();

    // Attempt to initialize the persistence service
    try{
        // Create an instance of the service interface
        StorefrontServiceImpl serviceImpl = new StorefrontServiceImpl();

        // Store the service into application scope
        getServletContext().setAttribute(
            IConstants.SERVICE_INTERFACE_KEY, serviceImpl );
    } catch( DatastoreException ex ){
        // If there's a problem initializing the service, disable the web app
        ex.printStackTrace();
        throw new UnavailableException( ex.getMessage() );
    }
}
```

The final step that needs to be shown is how we invoke the Storefront service interface from an **Action** class. The relevant methods are highlighted in Example 6-8.

Example 6-8. The LoginAction from the Storefront application

```
package com.oreilly.struts.storefront.security;

import java.util.Locale;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.exceptions.BaseException;
import com.oreilly.struts.storefront.framework.UserContainer;
import com.oreilly.struts.storefront.framework.StorefrontBaseAction;
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.service.IStorefrontService;

/**
 * Implements the logic to authenticate a user for the storefront application.
 */
public class LoginAction extends StorefrontBaseAction {
    /**
     * Called by the controller when the a user attempts to login to the
     * storefront application.
     */
    public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
        throws Exception{

        // Get the user's login name and password. They should have already
        // validated by the ActionForm.
        String email = ((LoginForm)form).getEmail();
        String password = ((LoginForm)form).getPassword();

        // Login through the security service
        IStorefrontService serviceImpl = getStorefrontService();
        UserView userView = serviceImpl.authenticate(email, password);

        UserContainer existingContainer = null;
        HttpSession session = request.getSession(false);
        if ( session != null ){
            existingContainer = getUserContainer(request);
            session.invalidate();
        }else{
            existingContainer = new UserContainer();
        }

        // Create a new session for the user
        session = request.getSession(true);
        existingContainer.setUserView(userView);
        session.setAttribute(IConstants.USER_CONTAINER_KEY, existingContainer);

        return mapping.findForward(IConstants.SUCCESS_KEY);
    }
}
```

```
}  
}
```

The first highlighted line calls the `getStorefrontService()` method. This method is located in the super class called `StorefrontBaseAction`, because every Action class will need to call this method. The implementation for the `getStorefrontService()` method just returns the `StorefrontServiceImpl` object that was created during the `init()` method of the `ActionServlet`.

The `StorefrontBaseAction` class, which includes the `getStorefrontService()` method, is shown in Example 6-9.

Example 6-9. The StorefrontBaseAction class

```
package com.oreilly.struts.storefront.framework;  
  
import java.util.Collection;  
import java.util.LinkedList;  
import java.util.List;  
import java.util.Locale;  
import java.util.Iterator;  
import javax.servlet.http.*;  
import org.apache.struts.action.*;  
import com.oreilly.struts.storefront.framework.util.IConstants;  
import com.oreilly.struts.storefront.framework.exceptions.*;  
import com.oreilly.struts.storefront.service.IStorefrontService;  
/**  
 * An abstract Action class that all store front action classes should  
 * extend.  
 */  
abstract public class StorefrontBaseAction extends Action {  
    protected IStorefrontService getStorefrontService(){  
        return (IStorefrontService)  
            getApplicationObject(IConstants.SERVICE_INTERFACE_KEY );  
    }  
  
    /**  
     * Retrieve a session object based on the request and the attribute name.  
     */  
    protected Object getSessionObject( HttpServletRequest req,  
                                       String attrName) {  
        Object sessionObj = null;  
  
        // Don't create a session if one isn't already present  
        HttpSession session = req.getSession(true);  
        sessionObj = session.getAttribute(attrName);  
        return sessionObj;  
    }  
  
    protected String getLoginToken(HttpSession session) {  
        return (String)session.getAttribute(IConstants.LOGIN_TOKEN_KEY);  
    }  
}
```

```
public boolean isLoggedIn( HttpServletRequest request ){
    UserContainer container = this.getUserContainer(request);
    return ( container != null && container.getUserView() != null );
}

/**
 * Return the instance of the ApplicationContainer object.
 */
protected ApplicationContainer getApplicationContainer() {
    return (ApplicationContainer)
        getApplicationObject( IConstants.APPLICATION_CONTAINER_KEY );
}

protected void removeLoginToken( HttpSession session ) {
    session.removeAttribute( IConstants.LOGIN_TOKEN_KEY );
}

/**
 * Retrieve the UserContainer for the user tier to the request.
 */
protected UserContainer getUserContainer( HttpServletRequest request ) {
    UserContainer userContainer =
        (UserContainer)getSessionObject( request, IConstants.USER_CONTAINER_KEY );

    // Create a UserContainer for the user if it doesn't exist already
    if( userContainer == null ) {
        userContainer = new UserContainer();
        userContainer.setLocale( request.getLocale() );
        HttpSession session = request.getSession();
        session.setAttribute( IConstants.USER_CONTAINER_KEY, userContainer );
    }
    return userContainer;
}

/**
 * Retrieve an object from the application scope by its name. This is
 * a convenience method.
 */
protected Object getApplicationObject( String attrName ) {
    return servlet.getServletContext().getAttribute( attrName );
}

protected void setLoginToken( HttpSession session,
                               String path ) {
    session.setAttribute( IConstants.LOGIN_TOKEN_KEY, path );
}
}
```

The `getApplicationObject()` method is just a convenience method for the Storefront Action classes, which calls the `getAttribute()` method on the `ServletContext` object.

Finally, the second highlighted line in Example 6.8 actually invokes a service method on the implementation. The `authenticate()` method is called and a value object called `UIView` is returned back to the Action.

```
|    UIView userView = serviceImpl.authenticate(email, password); |
```

This object is placed inside a session object and the Action returns. If a user isn't found with a matching set of credentials, the `authenticate()` method will throw an `InvalidLoginException`. This can be seen in the in Example 6-7.

Notice that the Action class is programming against the `IStorefrontService` interface, not the actual implementation object. As we said, this is important to prevent alternate implementations from having a rippling effect on the Action classes. For example, if we decided to use an alternate persistence framework, all we would have to do is create a new class that implemented the `IStorefrontService` interface and initialize it at runtime instead of the current one. If it was necessary to be able to switch back and forth, it might be worthwhile to initialize the Storefront service using a different approach than what was shown. Instead of hardcoding the service implementation in the `init()` method of the `ActionServlet`, you can read in a fully qualified class name to instantiate and store in the application. This value could be added to the init parameters of the `web.xml` file and then nothing would have to change when alternate implementations were used.

Conclusion

We covered a lot of ground in this chapter. It may be little overwhelming if you are new to the concept of a model and persistence mechanism. This is definitely one of the chapters that you might want to go back and re-read again. The rest of the book will assume that these topics are familiar to you and will not spend any time discussing them. Make sure you understand this material before moving on.

7

Struts View Components

This chapter introduces the components that make up the view portion of the Struts framework. The framework uses the view components to render dynamic content for the client. Based primarily on JavaServer Pages, the components provide support for internationalized applications, as well extensive support for user input acceptance, validation, and error handling, all of which make it easier for the developer to focus on business requirements. This chapter concludes the three-part discussion of how the Struts framework implements the MVC pattern.

This chapter will also briefly discusses the up and coming JavaServer Faces specification. Although still in early draft, it appears that this specification will impact the Struts framework in the future.

What exactly is a View?

In a general sense, the view represents a display of the domain model in a UI. There doesn't have to be just a single view of the model, there can actually be many different ways to represent the same model. As you learned in Chapter 5, the domain model contains the business entities, which hold state for the application. Metaphorically speaking, a view is a window that clients can use to peek at the state of the model. The perspective may be different depending on which window a client looks through. For example, in the Storefront application, the front page shows a set of featured items in the catalog. It doesn't show all of the information about each item, only a small portion. This "summary" view is used in the Storefront application where real estate is scarce as in the first page. Figure 7-1 shows the main page of the Storefront application.

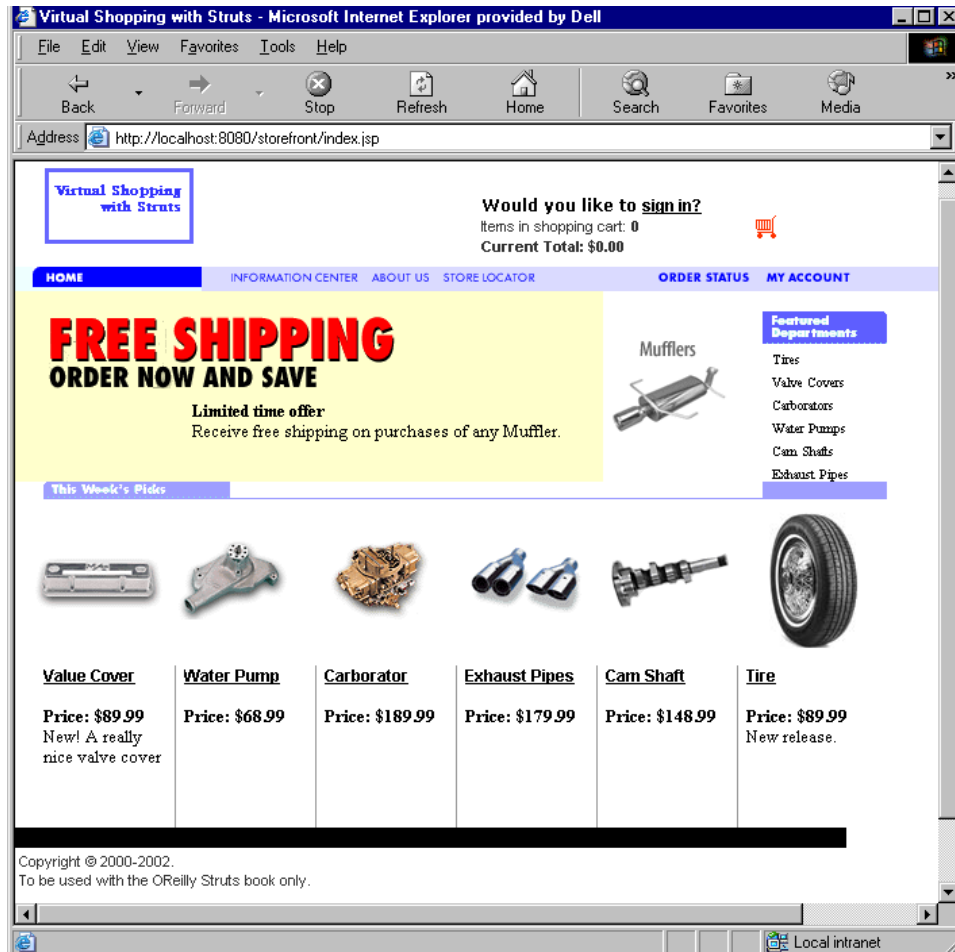


Figure 7-1. The Storefront Main Page is One Perspective of the Model

When a user selects one of the items for sale, all of the details of the item are displayed. The user is still looking at the same business model, but the view is different. This alternate view of the model is shown in Figure 7-2.

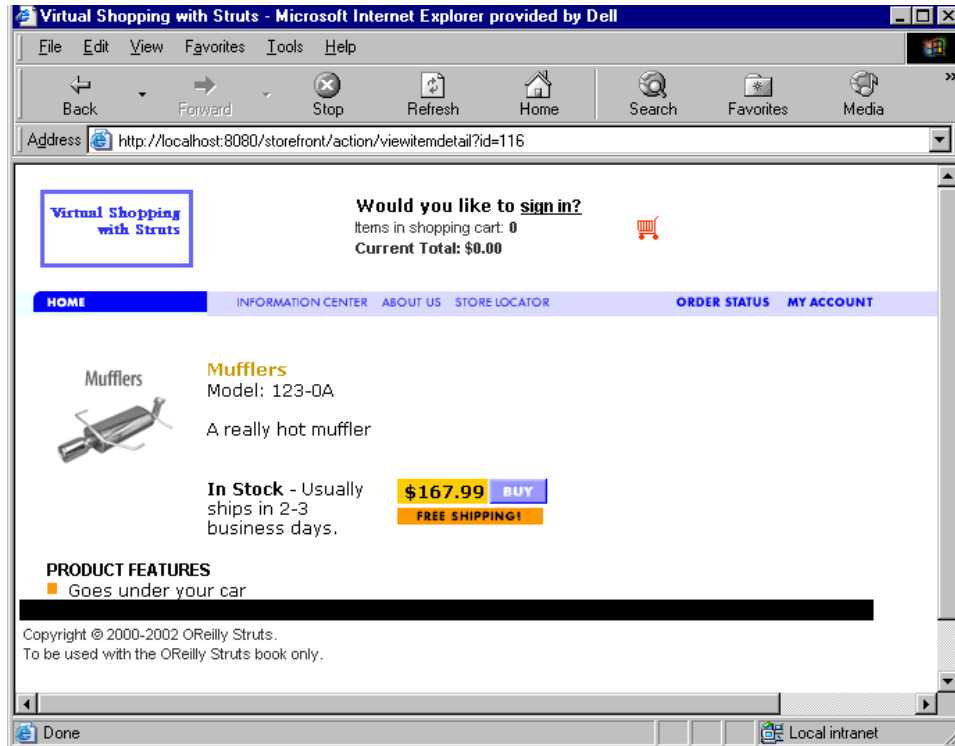


Figure 7-2. An Alternate View of the Storefront Model

A more detailed view of the business model is necessary for this page. There may be different JSP pages, images, multimedia files, and other view related components. These two different perspectives are in fact two different views of the same model.

Because business objects don't have a natural way of representing themselves externally, it's up to the view components to present the domain model information to the clients. This presentation may be in the form of XSLT and XML, Simple Object Access Protocol (SOAP) messages returned to a web service client, or in the case of the Storefront application, HTML rendered in a browser. In any case, the model is used to represent the state of the application, while the views are used to present the model, or a portion of it, to the client.

It wouldn't be hard to imagine a different type of client, a wireless device for example, looking at a completely different set of views, but still using the same model. Although the Storefront application shown throughout this book is geared towards B2C, it's possible to have a B2B application using the same model to make part ordering functionality available to partners. As long as the proper separation is maintained between the model and the presentation layer, you can build any number of views for any number of clients on top of the same domain model.

Using Views within the Struts Framework

Generally speaking, the views in the Struts framework are built using JSP pages. Other presentation technologies can be used, but JSP is widely used within the Struts community. The JSP technology is a very popular choice for generating dynamic content using the Java language. There are other approaches and frameworks available for performing the same behavior. The alternatives will be discussed in Chapter 16 *Using Alternate Presentation Technologies*.

Although JSP pages are the main components used by most Struts applications to render the views, there are additional components that can be used by or in conjunction with the JSP pages.

- HTML Documents
- JSP Custom Tag Libraries
- JavaScript and Style Sheets
- Multimedia Files
- Message Bundles
- ActionForm Classes

HTML Documents

Although HTML documents are only able to generate static content, there's nothing wrong with using standard HTML documents within your Struts applications. Obviously they are unable to render any dynamic data, but they are still valid to use whenever the view is static and you don't need the dynamic capabilities of JSP.

For example, a login page may just be an ordinary HTML document that invokes a Struts login *Action*. To take advantage of many of the automatic features of the Struts framework, especially ones related to custom tags, you would need to utilize JSP instead of straight HTML.

JSP Custom Tags

The JSP custom tags can play a very important role within a Struts application. Although an application is not required to use them, in some particular scenarios, it would be very difficult without them. In future version of the framework, they will maintain their importance, but the framework will make it easier to use alternate strategies.

JavaScript and Style Sheets

The Struts framework doesn't prevent you from using JavaScript within an application. On the contrary, it provides functionality within the tag libraries to help facilitate using JavaScript. The JavaScript support within custom tags will be discussed in Chapters 8.

Style sheets are used to help web designers gain more control over the appearance of a web site. Features such as character size, color, font, and other look and feel characteristics can be changed in a central location and will have an immediate effect throughout the entire site.

The Struts framework does not prevent you from using style sheets. Just like with standard HTML files, you can include style sheets in the JSP page and it will be rendered on the browser just as they would be for standard HTML pages.

Multimedia Files

Multimedia files are used in just about every web application. These include but are not limited to:

- Images (.gif, .jpg, etc.)
- Audio (.wav, .mp3, etc.)
- Video (.avi, .mpg, etc.)

Images are probably the most widely used, although for B2B applications, audio and video files are also prevalent. The Struts framework supports using multimedia files within an application. The support is achieved mainly through the use of custom tags, but you are also free to use standard HTML to render these types of resources.

There are a few sore spots when it comes to rendering images using the custom tags. It mainly has to do with the differences between absolute and relative associations. These issues however, are more related to web applications in general, rather than a particular issue within the Struts framework.

Message Bundles

The message resources, or resource bundles as they are commonly referred, are a very important component for Struts applications. Not only do they provide a means to support localization, but they also help to reduce maintenance time and redundancy throughout an application.

For example, suppose your web application uses certain text labels or messages in multiple locations. Instead of hard-coding the string in every page, you can specify it in the bundle and retrieve it using one of the custom tags. Then if the text label or message

needs to change, only the one in the bundle needs to be modified. This helps to reduce maintenance time because it only needs to be changed in one location.

Some might consider the resource bundles belonging to the model, rather than the view. However, because the bundle also contains labels and strings for such things as text fields and checkbox labels, page titles, and so on, a valid argument can be made either way.

Using JavaBeans within the View Components

It might seem weird to talk about JavaBeans within a chapter on views. However, since JavaBeans make up a large portion of how the model data is used within the framework, it makes sense to briefly discuss them.

JavaBean Quick Refresher

JavaBeans itself is a portable, platform independent component model written in the Java programming language. The JavaBeans component architecture allows developers to create reusable components that can be used on any platform that supports a JVM. The JavaBeans model supports properties, events, methods, and persistence.

The extent to which the Struts framework and Java web applications in general, utilize JavaBeans is much less than what's described by the JavaBeans specification. They are used much more like ordinary Java objects, however they must follow certain guidelines for them to be useful in Struts applications:

- Must provide a zero-argument constructor
- Should provide both a `get<PropertyName>` and `set<PropertyName>` method for all properties defined within the bean
- For boolean properties, if the method `is<PropertyName>` is present, it will be used to read the property value
- For indexed properties where there is a property defined like `<PropertyElement>[]`, then the methods `get<PropertyName>(int a)` and `set<PropertyName>(int a, PropertyElement b)` should be present

These guidelines are necessary for the beans to be introspected at runtime by the framework.

One of the common traps that Struts developers fall into when dealing with JavaBeans is to use a different return type from the parameter type. If you create a method in a JavaBean that passes a String as an argument:

```
public void setDescription( String newDescription );
```

then you must have a corresponding get method that returns the same type:

```
public String getDescription();
```

If the return type differs from the parameter type, the Struts framework may not recognize it as a bean and you will likely get an error message stating that “No getter method could be found” or “No setter method could be found” for the property name.

How are JavaBeans used by Struts Applications?

In a strict MVC architecture, the view gets updates directly from the model when it receives a notification that something has changed within the model. Figure 7-3 shows a diagram illustrating this.

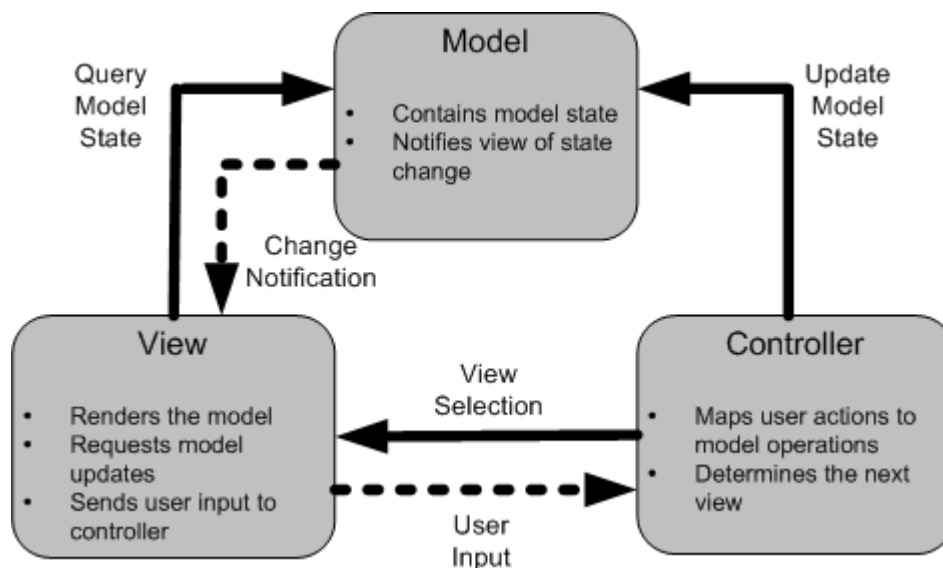


Figure 7-3. The view queries the model for state information

With web applications, this notification is not possible, or at least is difficult to achieve. Normally, it's up to the client to issue a request to the controller to refresh a view from the model state. It's up to the client to “pull” the view from the model, instead of the model “pushing” changes out to the view.

The Data Transfer Object Pattern

Chapter 6 discussed one approach to building the model components for a Struts application. The one thing that was left intentionally out until now is exactly how the view accesses the data from the model. To understand this, it will help to understand the Data Transfer Object pattern.

The *Data Transfer Object* (DTO) pattern is used quite frequently in J2EE applications, where distributed components making remote calls can suffer serious performance impacts from using too remote invocations. It also has been used in other technologies as well, but under different names and disguises.

In some software literature, it is also referred to as the Value Object or Replicate Object pattern.

A DTO is a coarse-grained view of what is typically fine-grained information. It aggregates various attributes from one or more business entities and puts this information into a JavaBean instance. This instance can then be passed throughout the local application, or even serialized and sent over the network, and clients can retrieve the model information from a local object and not suffer any negative performance impact.

The DTO doesn't normally provide any business logic or validation, but does provide access to the properties of the bean. Some documentation on this pattern suggests that the bean should be immutable to help reinforce that the object is local and changes will not be reflected in the system. However, this can cause problems because the JavaBean specification requires that all private properties have a `getXXX` and `setXXX` method. It is up to you to determine how best to handle the mutability of DTOs based on your requirements.

DTOs are effectively part of the model, since they are just local, possibly immutable copies of the business objects. Within Struts applications, they are used by the view to deliver the model data that is rendered along with the static information in the page.

What are ActionForms?

Almost every web application has a requirement to accept input from users. Some examples of user input are credit card information, billing and shipping address information, or even something as small as a username and password. The HTML language provides the necessary components to render the input fields in a browser, including text boxes, radio buttons, check boxes, buttons, and many more. When building these types of pages, the input components must be nested inside of an HTML form element. Example 7-1 illustrates a very basic *Signin* page similar to the one used in the Storefront application.

Example 7-1. Form fields must be nested inside of a form element

```
<html>
<title>Example 7-1. O'Reilly Struts Book</title>
<link rel="stylesheet" href="stylesheets/main.css" type="text/css">
</head>

<body>

<form method="post" action="/action/signin">

<!-- The Table layout for the email and password fields -->
<table BORDER="0" cellspacing="0" cellpadding="0">
<tr>
  <td>Email: </td>
  <td>&nbsp;</td>
  <td>
    <input type="text" name="email" size="20" maxlength="20"/>
  </td>
</tr>

<tr>
  <td>Password:</td>
  <td>&nbsp;</td>
  <td class="alignformslist">
    <input type="text" name="password" size="20" maxlength="25"/>
  </td>
</tr>

<!-- The Table layout for the signin button -->
<table width="250" border="0">
<tr>
  <td>
    <input type="submit" name="Submit" value="Signin" class="Buttons">
  </td>
</tr>
</table>
</form>
</body>
</html>
```

When the user presses the *Signin* button on the HTML form from Example 7-1, the values within the fields are submitted along with the HTTP request. The server application can retrieve the values that were entered, perform input validation on the data, and then pass the data to another component in the application where the actual authentication process can occur. If the input data fails the input validation rules, the application should return back to the previous location, redisplay some or all of the values entered, and display an error message that the login attempt failed.

Performing all of this functionality manually, retrieving the values, executing the validation, returning back and displaying error messages on a failure, can be a daunting task. This type of behavior is typically performed in many places throughout a web

application. It would be nice to have this functionality taken care of for you by the framework and be able to reuse it across applications.

Fortunately, the Struts framework does provide this functionality and will handle these tasks on behalf of your application. The Struts framework relies on the `org.apache.struts.action.ActionForm` class as the key component for handling these tasks.

The `ActionForm` class is used to capture input data from an HTML form and transfer it to the `Action` class. Since the HTML input components don't natively include an input buffer and users quite often enter invalid data, web applications need a way to store the input data temporarily, so that it can be redisplayed when an error occurs. In this sense, the `ActionForm` class acts as a buffer to hold the state of what the user entered while it is being validated. The `ActionForm` also acts as a "firewall" for your application in that it helps to keep suspect or invalid input out of your business tier until it can be scrutinized by the validation rules. Finally, the `ActionForm` is also used to transfer data from the `Action` class back to the HTML form. This allows more consistency for your HTML forms, because they are always pulling data from the `ActionForm`.

When the user input data does pass input validation, the `ActionForm` is passed into the `execute()` method of the `Action` class. From there, the data can be retrieved from the `ActionForm` and passed on to the business tier.

Because the `ActionForm` imports packages from the Servlet API, you shouldn't pass the `ActionForm` to the business tier. Doing so would couple the business methods to the Servlet API and make it more difficult to reuse the business tier components. Instead, the data within the `ActionForm` should be transferred to an object from the domain model instead of being passed as an argument to the business tier. A common approach is to create a data transfer object and populate it with the data from the `ActionForm`.

You don't have to declare an `ActionForm` for every HTML form in your application. The same `ActionForm` can be associated with one or more action mappings. This means that they can be shared across multiple HTML forms. For example, if you had a wizard interface, where a set of data was entered and posted across multiple pages, a single `ActionForm` can be used to capture all of this data, a few fields at a time.

ActionForms and Scope

ActionForms can have two different levels of scope, request and session. If request scope is used, an ActionForm will only be available until the end of the request/response cycle. Once the response has been returned to the client, the ActionForm and the data within it is no longer accessible.

If you need to keep the form data around longer than the request, you can configure an `ActionForm` to have session scope. This might be necessary if your application captures data across multiple pages, similar to a wizard dialog. An `ActionForm` that has been configured with session scope will remain in the session until it's removed, replaced with another object, or until the session times out. The framework doesn't have a built-in facility for cleaning up session scoped `ActionForm` objects automatically. Like any other object placed into the `HttpSession`, it's up to the application to routinely perform clean up on the resources stored there. This is slightly different from objects placed into the request, because once the request is finished, they can be reclaimed by the garbage collector since they can no longer be referenced.

Unless you need to specifically hold the form data across multiple requests, you should use request scope for your `ActionForm` objects.

If you don't specify the *scope* attribute for an `action` mapping, the `ActionForm` will default to session scope. To be safe, you should always explicitly specify the scope of the `ActionForm`. To see how to specify the scope for an action element, see "The Struts Configuration DTD" in Chapter 4.

When the controller receives a request, it will attempt to recycle an `ActionForm` instance from either the request or the session, depending on the scope that the `ActionForm` has in the `action` element. If no instance is found, a new instance will be created.

The Lifecycle of an ActionForm

The section "Using the Struts ActionForm" in Chapter 3 described the steps that are taken by the framework when an `ActionForm` is being used by an application. Understanding these steps, it's easy to get a picture of the life cycle of an `ActionForm`. Figure 7-4 illustrates the main steps that are taken by the framework that have some effect on the `ActionForm`.

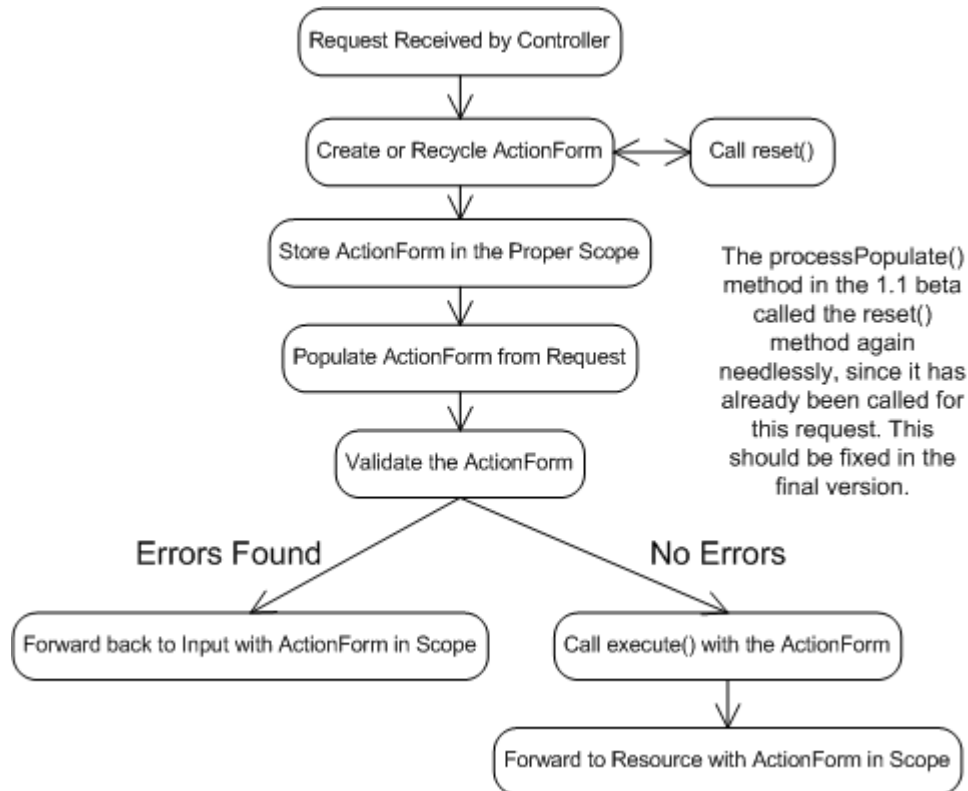


Figure 7-4. The Life Cycle of an ActionForm

All of the steps that a request goes through during request processing are not shown in Figure 7-4, only the ones that are relevant to an **ActionForm**. Notice that when an **ActionForm** detects one or more validation errors, it performs a forward back to the resource identified in the `input` attribute. The data that was sent in the request is left in the **ActionForm** so that it can be used to re-populate the HTML fields.

Creating an ActionForm

The **ActionForm** class provided by the Struts framework is abstract. You need to create subclasses of it to capture your application specific form data. Within your subclass, you should define a property for each field that you wish to capture from the HTML form. For example, suppose we want to capture the email and password fields from a form, similar to the one in Example 7-1. Example 7-2 illustrates the **LoginForm** for the Storefront application that can be used to store and validate the email and password fields.

Example 7-2. The LoginForm Stores the Email and Password Fields

```
package com.oreilly.struts.storefront.security;
```

```
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;
/**
 * Form bean for the user signon page.
 */
public class LoginForm extends ActionForm {
    private String email = null;
    private String password = null;

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

    public String getEmail() {
        return (this.email);
    }

    public String getPassword() {
        return (this.password);
    }

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

    /**
     * Validate the properties that have been sent from the HTTP request,
     * and return an ActionErrors object that encapsulates any
     * validation errors that have been found. If no errors are found, return
     * an empty ActionErrors object.
     */
    public ActionErrors validate(ActionMapping mapping, HttpServletRequest request){
        ActionErrors errors = new ActionErrors();

        if( getEmail() == null || getEmail().length() < 1 ){
            errors.add("email", new ActionError("security.error.email.required"));
        }
        if( getPassword() == null || getPassword().length() < 1 ){
            errors.add("password", new ActionError("security.error.password.required"));
        }
        return errors;
    }

    public void reset(ActionMapping mapping, HttpServletRequest request){
        /** Because this ActionForm should be request scope, do nothing here because
         * the fields will be reset when a new instance is created. We could
         * have just not overrode the parent reset() method, but did so just
         * to provide an example of the reset() method signature.
         */
    }
}
```

When the form is submitted, an instance of the `LoginForm` will be created and populated from the request parameters. The framework does this by matching each

request parameter name against the corresponding property name in the `ActionForm` class.

The `ActionForm` is populated from request parameters, not request attributes. If you are forwarding from one action to another, you can't add a request attribute and expect that the `ActionForm` will be populated from it. Request parameters and request attributes are two separate resources.

The `ActionForm.validate()` Method

[\[Note: Brett, how should a method name like above be formatted in a heading 3?\]](#)

The `validate()` method may be called by the `RequestProcessor` for every request. Whether it's called or not depends on two things. First, an `ActionForm` must be configured for an `action` mapping. This means that the `name` attribute for an `action` element must correspond to the `name` attribute of one of the `form-bean` elements in the configuration file.

The second condition that must be met before the `RequestProcessor` will invoke the `validate()` method is that the `validate` attribute must have a value of "true". The following fragment shows an `action` element that uses the `LoginForm` from Example 7-2 and meets both requirements mentioned.

```
<action
  path="/signin"
  type="com.oreilly.struts.storefront.security.LoginAction"
  scope="request"
  name="loginForm"
  validate="true"
  input="/security/signin.jsp">
  <forward name="Success" path="/index.jsp" redirect="true"/>
  <forward name="Failure" path="/security/signin.jsp" redirect="true"/>
</action>
```

When the `signin` action is invoked, the framework will populate an instance of a `LoginForm` using values it finds in the request. Because the `validate` attribute has a value of "true", the `validate()` method in the `LoginForm` will be called. Even if the `validate` attribute is set to "false", the `ActionForm` will still be populated from the request if an `ActionForm` is configured for the action.

The `validate()` method in the base `ActionForm` class simply returns null. If you want to perform validation on the data that is submitted with the request, you'll need to override the `validate()` method in your `ActionForm` subclasses, as in Example 7-2.

The `validate()` method may return an `ActionErrors` object, depending on whether or not any validation errors were detected. You also can return null if there are

no errors; the framework will check for both null and an empty `ActionErrors` object. This saves you from having to create an instance of `ActionErrors` when there are no errors. The `ActionErrors` class and its parent class, `ActionMessages`, will be discussed later in this chapter.

The `ActionForm.reset()` Method

The `reset()` method has been a bane for much of the Struts user community at one time or another. Exactly when the `reset()` method is called and what should be done within it is almost always misinterpreted. This doesn't mean that one implementation is more correct than another, but there are misconceptions that many new Struts developers pick up and then have a hard time shaking regarding the `reset()`.

As Figure 7-4 shows, the `reset()` method is called for each new request, regardless of the scope of the `ActionForm`. It's called before the `ActionForm` has been populated from the request. The method was added to the `ActionForm` class originally to help facilitate resetting `boolean` properties back to their defaults. To understand why they need to be reset, it's helpful to know how the browser and the HTML form submit operation processes checkboxes.

When an HTML form contains checkboxes, only the values for the checkboxes that are checked are sent in the request. Those that are not checked are not included as a request parameter. Therefore, the `reset()` method was added to allow applications to reset the `boolean` properties in the `ActionForm` back to false, since false wasn't included in the request and the `boolean` values would possibly be stuck in the "true" state.

The `reset()` method in the base `ActionForm` contains no default behavior, since no properties are defined in this abstract class. Applications that extend the `ActionForm` class are allowed to override this method and reset the `ActionForm` properties to whatever state they wish. This may include setting `boolean` properties to true or false, setting String values to null or some initialized value, or even instantiating instances of other objects that the `ActionForm` holds on to. For an `ActionForm` that has been configured with request scope, the framework will essentially create a new instance for each new request. Since a new instance is created, there's not much need to `reset()` the values back to any default state. ActionForms that are configured with session scope are different however. This is the time that the `reset()` method comes in handy.

Declaring ActionForms in the Struts Configuration File

Once you have created a class that extends `ActionForm`, you need to configure the class in the Struts configuration file. The first step is to add a new `form-bean` element to the `form-beans` section of the file:

```
<form-beans>
  <form-bean
    name="loginForm"
```

```
        type="com.oreilly.struts.storefront.security.LoginForm" />
    </form-beans>
```

The value for the `type` field must be a fully qualified Java class name that is a descendant of `ActionForm`.

Once you have defined your `form-bean`, you can now use it in one or more `action` elements. It's very common to share one `ActionForm` across several actions. For example, suppose there was an admin application that managed the items in the Storefront application. There would need to be an HTML form where a new item could be added to the system. This might be the `createItem` action. There would also need to be a `getItemDetail` action to show the details of an existing item. Both of these HTML forms would look similar, but maybe submit to a different action. Since the HTML forms contained the same properties, both of these could use the same `ActionForm`.

To use an `ActionForm` in an `action` element, you need to specify a few attributes for each action mapping that uses the `ActionForm`. These attributes are `name`, `scope`, and `validate`:

```
<action
  path="/signin"
  input="/security/signin.jsp"
  name="loginForm"
  scope="request"
  type="com.oreilly.struts.storefront.security.LoginAction"
  validate="true">
  <forward name="Success" path="/index.jsp" redirect="true"/>
  <forward name="Failure" path="/security/signin.jsp" redirect="true"/>
</action>
```

For more information on the attributes of the action element, see “The Struts Configuration DTD” in Chapter 4.

Using an ActionForm in an Action

Once you have configured the `ActionForm` for a particular `Action`, you can then insert and retrieve values from it within the `execute()` method, as Example 7-3 illustrates.

```
Example 7-3. The ActionForm is available within the execute() Method

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

    // Get the user's login name and password. They should have already
    // validated by the ActionForm.
    String email = ((LoginForm)form).getEmail();
    String password = ((LoginForm)form).getPassword();
```

```
// Login through the security service
IStorefrontService serviceImpl = getStorefrontService();
UIView userView = serviceImpl.authenticate(email, password);

UserContainer existingContainer = null;
HttpSession session = request.getSession(false);
if ( session != null ){
    existingContainer = getUserContainer(request);
    session.invalidate();
}else{
    existingContainer = new UserContainer();
}

// Create a new session for the user
session = request.getSession(true);
existingContainer.setUserView(userView);
session.setAttribute(IConstants.USER_CONTAINER_KEY, existingContainer);

return mapping.findForward(IConstants.SUCCESS_KEY);
}
```

It's not mandatory that you always use an **ActionForm** to capture the data from an HTML form. Even if you don't declare an **ActionForm** for a form, the data is still available from the request. However, your application will have to manually handle the process of validation and error handling from the Action class.

Declaring ActionForm Properties as Strings

All request parameters that are sent by the browser are Strings. This is true regardless of the type that the value will eventually map to in Java. For example, dates, times, Booleans, and other values are all strings when they are pulled out of the request. They will also be converted into strings when they are written back out to the HTML page. Therefore, it makes sense that all of the **ActionForm** properties where the input may be invalid, should be of type String. The reason for this is to support displaying the data back out in its original form to the user, when there is an error. For example, if a user types in "12Z" for a property expecting to be an Integer, there's no way to store "12Z" into an **int** or **Integer** property. However, you can store it into a String until it can be validated. This same value, which is stored in a String, can be used to render the input field with the value, so the user can see their mistake. This is functionality that even the most inexperienced users have come to expect and look for.

ActionForms are not the Model

Many developers get confused when they learn about the `ActionForm` class. Although it can hold state for an application, the state that it holds should be limited and constrained to the “user input” that is received from the client and should only hold it until it can be validated and transferred to the business tier.

You’ve already seen why it’s important to separate the model from the presentation tier in an application. Business objects can be persisted and should contain the business logic for an application. They should also be reusable. This set of criteria does not match up well when compared against `ActionForms`. For one thing, the `ActionForm` class is most certainly tied to the Struts framework and explicitly to a web container, since it imports `javax.servlet` packages. It would be very difficult to port `ActionForm` classes to a different type of framework, like a Swing application.

`ActionForms` are exclusively designed to capture the HTML data from a client, allow “presentation validation” to occur, and to provide a transport vehicle for the data back to the more persistent business tier. You will be well served to keep the `ActionForms` separate from your business components.

Using ActionErrors

Earlier in the chapter, you saw that the `validate()` method returned an `ActionErrors` object. The `ActionErrors` class encapsulates one or more errors that have been discovered by the application. Each problem discovered is represented by an instance of `org.apache.struts.action.ActionError`.

An `ActionErrors` object has request scope. Once an instance is created and populated by the `validate()` method, it is stored into the request. Later, the JSP page can retrieve the object from the request and use the `ActionError` objects contained within it to display errors messages to the user.

The Struts framework includes a JSP custom tag that makes retrieving and displaying the errors messages very easy. This tag, called `ErrorsTag`, will be discussed in Chapter 8.

An instance of `ActionErrors` can be instantiated in the `validate()` method and populated by adding instances of the `ActionError` class to it. The `LoginForm` from Example 7-2 demonstrated this and is illustrated again here for convenience:

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

    if( getEmail() == null || getEmail().length() < 1 ){
        errors.add("email", new ActionError("security.error.email.required"));
    }

    if( getPassword() == null || getPassword().length() < 1 ){
        errors.add("password", new ActionError("security.error.password.required"));
    }
    return errors;
}
```

The `validate()` method in this fragment checks to make sure that the email and password fields have been set with values other than an empty string. If not, `ActionError` objects are added to the `ActionErrors` instance.

The `ActionError` class contains several useful constructors. Several are listed here:

```
public ActionError(String key);
public ActionError(String key, Object value0);
public ActionError(String key, Object value0, Object value1);
public ActionError(String key, Object[] values);
```

The key argument is a String value that corresponds to a key from one of the application's resource bundles. The custom tag `ErrorsTag` uses this value to lookup the message to display to the user. The remaining arguments are used as parametric replacement values for the message. For example, if you had a bundle message defined like this:

```
| global.error.login.requiredfield=The {0} field is required for login
```

then we could create an instance of an `ActionError` like this:

```
| ActionError error = new ActionError("global.error.login.requiredfield", "Email" );|
```

The message displayed to the user after substituting in the "Email" string would be:

```
| The Email field is required for login
```

If building I18N applications is a requirement for you, you must be careful when using hard-coded string values as in the previous example. The String "Email" can't be easily localized since it's hard-coded into the source code.

In this case, you should also get the localized value from the bundle as well before passing it as an argument in `ActionError` constructor.

When adding instances of the `ActionError` class to the `ActionErrors` object, the first argument in the `add()` method is a property that can be used to retrieve a specific `ActionError` instance. If all of your `ActionError` instances can be treated the same and you have no need to retrieve them individually, you can use the constant `ActionErrors.GLOBAL_ERROR` similar to this:

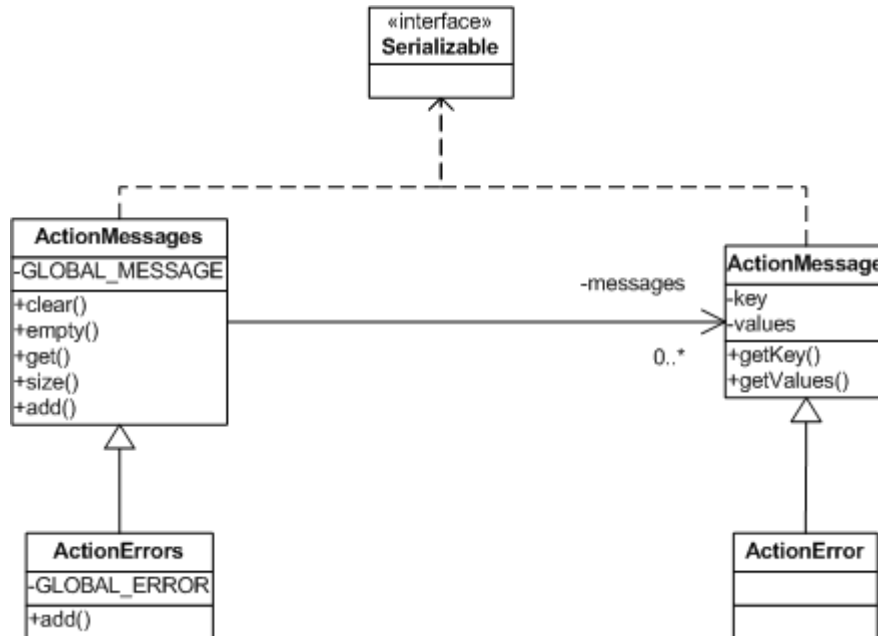
```
errors.add(ActionErrors.GLOBAL_ERROR,  
          new ActionError("security.error.password.required"));
```

The ActionMessage Class

Starting with Struts 1.1, a new message class has been added that can also be used to display messages to a user. The `org.apache.struts.action.ActionMessage` class operates in the same manner that the `ActionError` class does, in fact, it's been added as the super class to the `ActionError` class.

The main reason that the `ActionMessage` class was added to the framework was that the name `ActionError` implies that it shouldn't be used for general purpose informational or warning types of messages, although it is used that way by many developers. A more general-purpose message class made sense.

The `ActionMessage` is used exactly like the `ActionError` class, except that it can represent a less severe message that needs to be displayed to the user. Instances of this class are created the same way and added to an `ActionMessages` object, instead of an `ActionErrors` object. Because `ActionError` is just a specialized message, it extends the `ActionMessage` class. In fact, the `ActionErrors` class also extends the `ActionMessages` class. Figure 7-5 illustrates the relationship between these classes.

Figure 7-5. *ActionErrors are Specialized ActionMessages*

Creating ActionErrors in the Action Class

The `ActionForm` is not the only place that you can create `ActionMessages` or `ActionErrors`. You may also create them in other parts of the framework. If for example, a business operation called from an `Action` raised an exception and you wanted to insert an error message informing the user, you can create `ActionErrors` from the `Action` class itself. The Struts `Action` class includes functionality to help support this.

When the business operation throws the exception, the `Action` class will catch it and take the appropriate steps. This is usually to return to the previous page and display an error message to the user. Returning to the previous state can be accomplished by returning the appropriate `ActionForward`, but the `ActionError` needs to be put in the request before the forward occurs.

In Chapter 10 *Exception Handling*, you'll learn how to take advantage of the new declarative exception handling and thus avoid having to deal with the exceptions in the `Action` class completely.

Example 7-4 illustrates how this is done within the `LoginAction` class.

Example 7-4. Creating ActionErrors from the execute() Method

```
public ActionForward execute( ActionMapping mapping,
```

```
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response )
throws Exception{

    /**
     * Get the user's email and password, which should have already been
     * validated by the ActionForm.
     */
    String email = ((LoginForm)form).getEmail();
    String password = ((LoginForm)form).getPassword();

    // Login through the security service
    IStorefrontService serviceImpl = getStorefrontService();

    UserView userView = null;
    try{
        userView = serviceImpl.authenticate(email, password);
    }catch( InvalidLoginException ex ){
        ActionErrors errors = new ActionErrors();
        ActionError newError = new ActionError( "security.login.failed" );
        errors.add( ActionErrors.GLOBAL_ERROR, newError );
        saveErrors( request, errors );
        // Return back to the previous state
        return mapping.findForward( mapping.getInput() );
    }

    // Authenticate was successful
    UserContainer existingContainer = null;
    HttpSession session = request.getSession(false);
    if ( session != null ){
        existingContainer = getUserContainer(request);
        session.invalidate();
    }else{
        existingContainer = new UserContainer();
    }

    // Create a new session for the user
    session = request.getSession(true);
    existingContainer.setUserView(userView);
    session.setAttribute( IConstants.USER_CONTAINER_KEY, existingContainer);

    return mapping.findForward(IConstants.SUCCESS_KEY);
}
}
```

In Example 7-4, when an `InvalidLoginException` is thrown by the `authenticate()` method, the exception is caught and an `ActionError` is created. The `saveErrors()` method exists in the Struts base `Action` class and stores the `ActionErrors` object into the request. A corresponding `saveMessages()` method also exists for a similar purpose.

Once the `ActionErrors` or `ActionMessages` is stored in the request and control is forwarded to a JSP page, the framework includes several JSP custom tags for printing out the messages to the user. These will be discussed in the next chapter.

Performing Presentation Validation

This chapter has touched on performing your application's input validation in the `validate()` method of the `ActionForm`. You can create whatever presentation validation rules you need in this method. For example, the `LoginForm` from Example 7-2 validated that the email and password fields were entered and they were not empty strings. Although this is trivial example, you have the freedom to validate anything that you would like. A very common validation rule is to ensure that a string value that should be a number is in fact a string representation of a valid number. The validate routine for this rule might look like the one in Example 7-5.

Example 7-5. Performing a Number Validation Rule

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

    String orderQtyStr = getQuantity();

    if( orderQtyStr == null || orderQtyStr.length() < 1 ){
        errors.add( ActionErrors.GLOBAL_ERROR,
                    new ActionError( "order.quantity.required" ));
    }

    // Validate that the qty entered was in fact a number
    try{
        // Integer.parse was not used because it's not really I18N safe
        java.text.Format format = java.text.NumberFormat.getNumberInstance();
        Number orderQty = (Number)format.parseObject( orderQtyStr );
    }catch( Exception ex ){
        // The Quantity entered by the user was not a valid qty
        errors.add( ActionErrors.GLOBAL_ERROR,
                    new ActionError( "order.quantity.invalid" ));
    }
    return errors;
}
```

As you can imagine, checking for required values or validating that data entered fits a certain format or is of a certain type, is done quite often in web applications. Since all data that is retrieved from a request is of type `String`, you must ensure that the data is not going to corrupt your business components.

Although you can perform the validation programmatically, the Struts framework provides an alternative that can be external to the `ActionForm` and `validate()` method. In fact, for most of the standard validation rules, you don't have to write a line of code because the rules are already defined. All you need to do is to declaratively

configure the rules that you need in an extra XML file. The Struts Validator will be covered in Chapter 11.

Using Dynamic ActionForms

Using the `ActionForm` class has many advantages over performing the functionality yourself in the `Action` class or some set of helper utility classes. Since the behavior that the `ActionForm` class provides is needed in nearly every web application, as well as many times in the same application, using the framework to perform the work can really reduce the development time and your frustration level. Having stated the benefits of using `ActionForms`, there are a few very important downsides to using them.

The first and foremost problem with using `ActionForms` is the sheer number of classes that it can add to a project. Even if you share `ActionForm` definitions across many pages, the additional classes make it more difficult to manage a project and provide maintenance. This is why some developers might create a single `ActionForm` and implement the properties for all of the HTML forms within these. The problem with this of course, is that combining the fields into this one class makes it a point of contention on a project that has more than just a few developers.

Another major liability is the requirement to define the properties in the `ActionForm` that need to be captured from the HTML form. If a property is added or removed from the HTML form, the `ActionForm` class may need to be modified and recompiled.

For these reasons, a new type of `ActionForm` was added to the framework, which is dynamic in nature and allows you to avoid having to create concrete `ActionForm` classes for your application. The dynamic `ActionForm` is implemented by the base class `org.apache.struts.action.DynaActionForm`, which extends the `ActionForm` class. There are only three real differences among all `ActionForms` for any application:

- The properties that the `ActionForm` defines
- The `validate()` method
- The `reset()` method

The properties for a `DynaActionForm` are configured in the Struts configuration file, which you'll see how to do in the next section. The `reset()` method is called at exactly the same time during request processing as it is for a standard `ActionForm`. The one difference is that you have a little less control over what you do during the method. However, you can always subclass the `DynaActionForm` to override the reset behavior.

The validation of the presentation data is a little more complicated, because we'll need to wait until we talk about the Struts Validator components before talking about how validation occurs in a dynamic form.

Configuring Dynamic ActionForms

To use the `DynaActionForm` in your Struts application, the first step is to add a `form-bean` element to the configuration file. There are two very important differences between a `form-bean` element for a regular `ActionForm` and one that is dynamic.

First, a `form-bean` element for a dynamic `ActionForm` is required to have an attribute called `dynamic`, which must have a value of "true". This is necessary for the framework to understand that it should handle this `ActionForm` differently. The second difference is that you must include one or more `form-property` elements in order for the dynamic form to have properties. The `DynaActionForm` uses a `java.util.Map` internally to store key/value pairs. The `form-property` elements are loaded into the `Map` and become the properties that get populated by the framework.

The attributes for the `form-bean` and `form-property` are discussed in more detail in Chapter 4.

Example 7-5. A Dynamic ActionForm must be specified in the Struts Configuration File

```
<form-beans>
  <form-bean
    name="loginForm"
    dynamic="true"
    type="org.apache.struts.action.DynaActionForm">

    <!-- Specify the dynamic properties of the form -->
    <form-property
      name="email"
      type="java.lang.String" />
    <form-property
      name="password"
      type="java.lang.String" />

    <!-- You can also set the initial value of a property -->
    <form-property
      initial="false"
      name="rememberMe"
      type="java.lang.Boolean" />
  </form-bean>
</form-beans>
```

The declarative properties are what make the `ActionForm` dynamic. At runtime, the framework creates an instance of the `DynaActionForm` class and makes it possible to set and get the configured property values. To add new properties, you only need to

modify the configuration file. No source code needs to be changed. The power and flexibility that this provides for you is immense.

As Chapter 4 outlined, the `form-beam` element also allows you to specify the initial value for each property. The framework will set the property to that value when the application is started. The initial value is also used when the `reset()` method is called to reset the values back to their original state. If you don't include the `initial` attribute, then properties will be assigned default values based on the Java programming language; numbers to zero (0) and properties of type `Object` will be assigned a null value by the framework.

The `type` attribute expects a fully qualified Java class name. Therefore, you will need to use the wrapper classes for primitives. For example, `java.lang.Boolean` for a boolean property type, `java.lang.Integer` for an int property, and so on.

Performing Validation using Dynamic ActionForms

Since the `DynaActionForm` is used for every dynamic `ActionForm` and you don't provide subclasses of `ActionForm`, there's no way to override the `validate()` method. Fortunately, the framework comes to your aid again with a feature called the Struts Validator.

The Struts Validator was created by David Winterfeldt and is now in the main Struts distribution. The validator is a framework that was intended to work with Struts from the beginning. It supports basic validation rules like checking for required fields, email, date and time fields, and many others. One of the biggest benefits is that it provides many of the validation rules that web applications need to perform. It is also very easy to add your own validator; this will be covered in Chapter 11.

Looking Ahead to JavaServer Faces

JavaServer Faces (JSF) is designed to provide a standard set of JSP tags and Java classes, which will make it easier to build Java server application GUIs. The problem that's being addressed with this technology is when creating web applications; the Servlet and JSP technologies don't provide specific enough APIs for creating the client GUI. Developers have to rely on HTML components and one of several alternatives to facilitate style and behavioral changes to all of the views. The mechanism to manage all of this eventually takes on a life of its own and overshadows the business of the application. Instead of performing maintenance on business operations and logic, more and more time is needed to manage the presentation widgets.

JSF plans to fix this by creating a standard way to define complex HTML forms and other GUI elements. This will enable developers to focus their attention on a single component framework.

The first release of the framework includes the following design goals:

1. Create a standard GUI component framework, which can be leveraged by development tools to make it easier for tool users to both create high quality GUIs and manage the GUI's connections to application behavior.
2. Define a set of simple lightweight Java base classes for GUI components, component state, and input events. These classes will address GUI lifecycle issues, notably managing a component's persistent state for the lifetime of its page.
3. Provide a set of common GUI components, including the standard HTML form input elements. These components will be derived from the simple set of base classes (outlined in #1) that can be used to define new components.
4. Provide a JavaBeans model for dispatching events from client-side GUI controls to server-side application behavior.
5. Define APIs for input validation, including support for client-side validation.
6. Specify a model for internationalization and localization of the GUI.
7. Automatic generation of appropriate output for the target client, taking into account all available client configuration data, such as browser version, etc.
8. Automatic Generation of output containing required hooks for supporting accessibility, as defined by WAI.

What does JSF have to do with Struts?

JSF and Struts will fit together quite well, as developers will eventually be able to supplement or substitute the Struts custom tag libraries with JSF components. The rest of the Struts framework, both model and controller components will remain relatively unaffected by the JSF architecture.

JSF will also include an event model, but the architecture will be designed so that developers can choose to use the GUI components with or without the event mechanism.

Because there is no public specification for JSF currently, it's hard to be specific on exactly how the two will tie into each other. However, Craig McClanahan who is the founder of the Struts framework is also now the specification lead for the JavaServer Faces JSR. This means that we can look closely at Craig's comments on JSF and Struts and know that he is speaking with subject matter expertise.

You can find the JavaServer Faces JSR at <http://jcp.org/jsr/detail/127.jsp>.

[Note: Add a note in here about how the regular ActionForm reset() doesn't do anything, but the dynamic one actually resets everything. This may cause a change upgrading to them and you might need to extend and override the reset().]

See: <http://www.mail-archive.com/struts-user@jakarta.apache.org/msg30062.html>

9

Extending the Struts Framework

One of the biggest advantages of using a framework is the ability to extend and customize it based on the needs of the application. The Struts framework is no exception and it provides several very important extension points for developers to extend. This chapter takes a quick glance at several of those extension points and points out the benefits and a few downsides when extending the framework.

What are Extension Points?

Think of a framework as a house that comes pre-built with a percentage of the structure already complete, and then you have the option of modifying certain characteristics like wallpaper, color, brick or stucco. If the default characteristics already suit your needs, then you have the option of not changing anything. If you already like the color for example, then that's one less thing to worry about. This is similar to how a framework functions and that's a big advantage when building applications. If functionality is present in the framework and it suits the needs of your application, then you don't have to worry about that aspect of the application. This in turn, frees up developers and allows them to focus on the core application, rather than the infrastructure.

This is not a perfect analogy, but it makes the point that a good framework should provide much of the infrastructure, the foundation and the plumbing if you will, and provides flexibility for you to customize certain behavior. The most important aspect of a framework however, is that it should provide extension points throughout.

Framework extension points, also referred to as "hooks", allow you to extend the framework in specific places, to adapt it to meet the application's requirements. Where and how a framework provides these hooks is very important. If not done correctly, or in

the wrong locations, then it becomes very hard for an application to adapt the framework, which makes the framework less useful. The rest of this chapter focuses on where the Struts framework provides these extension points, and how you can take advantage of them, to build out specialized functionality for your application.

General Extension Points

This section discusses some extensions points that affect the overall framework, not necessarily one particular layer. Arguably, the most important of these is the concept of the *plugin* mechanism.

Using the Plugin Mechanism

The Struts framework provides a mechanism to allow components to be plugged-in and loaded dynamically. This feature was added in version 1.1 and is supported through the use of the `org.apache.struts.action.Plugin` interface. Any Java class can function as a plugin, as long as it implements the `Plugin` interface.

The `Plugin` interface contains two methods, as shown in Example 9-1.

```
Example 9-1. The org.apache.struts.action.Plugin Interface
public interface Plugin {
    /**
     * Receive notification that the specified sub-application is being
     * started up.
     */
    public void init(ActionServlet servlet, ApplicationConfig config)
        throws ServletException;

    /**
     * Receive notification that our owning sub-application is being
     * shut down.
     */
    public void destroy();
}
```

During startup of a Struts application, the `ActionServlet` will call the `init()` method for each and every `Plugin` that is configured; the framework supports one or more `Plugins` to be configured for each sub-application. Initialization routines that your plugin needs to perform should be done during the `init()` method. This is a good time to initialize a database connection, or establish a connection to a remote system, for example.

Initializing a database connection can also be done through the use of a `datasource`.

The second method that your plugin must implement is the `destroy()` method. The framework calls this method when the application is being shut down. You should perform any necessary cleanup behavior during this time. For example, this is the perfect time to close database connections, remote sockets, or any other limited resource that the plugin is using.

Let's provide a concrete example of how to use the plugin mechanism of the Struts framework. Suppose that your application needs the ability to communicate with an EJB tier. One of the first things that must be done before that can occur is to get a reference to the Java Naming and Directory Interface (JNDI) service. JNDI enables clients to have access to various naming and directory services, like datasources, JavaMail sessions, and also EJB home factories. Example 9-2 illustrates a simple example of acquiring an `InitialContext` for a JNDI service using the Struts plugin mechanism.

Example 9-2. An example of using the Struts `Plugin`

```
package com.oreilly.struts.storefront.framework.ejb;

import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.Context;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.config.ApplicationConfig;
import org.apache.struts.action.Plugin;
import javax.servlet.ServletException;

public class JNDIConnectorPlugin implements Plugin {
    private String jndiFactoryClass;
    private String jndiURL;
    private Context initCtx = null;

    public JNDIConnectorPlugin() {
    }

    public void init(ActionServlet servlet, ApplicationConfig config)
        throws ServletException{
        // Get the host and port where the JNDI service is running
        jndiFactoryClass = servlet.getInitParameter("jndiFactoryClass");
        jndiURL = servlet.getInitParameter("jndiURL");

        try{
            Hashtable props = new Hashtable();
            // The EJB spec also allows these to be read from jndi.properties file
            props.put( Context.INITIAL_CONTEXT_FACTORY, jndiFactoryClass );
            props.put( Context.PROVIDER_URL, jndiURL );
            initCtx = new InitialContext(props);
        }catch( Exception ex ){
            throw new ServletException( ex );
        }
        // Store the JNDI Context into the ServletContext
        servlet.getServletContext().setAttribute( "Storefront.InitCtx", initCtx );
    }
}
```

```
public void destroy(){
    try{
        if ( initCtx != null ){
            initCtx.close();
            initCtx = null;
            // No need to remove from ServletContext since app is being shut down
        }
    }catch( Exception ex ){
        ex.printStackTrace();
    }
}
```

When the framework calls the `init()` method of the `JNDIConnectorPlugin` class, the plugin creates an `InitialContext` object and stores it into the `ServletContext`. This allows the `JNDI Context` to be used by the entire application, when needed.

This is just a simple example; there are many possibilities that you can use the plugin for. The Validator framework, which we'll discuss in Chapter 11, utilizes the plugin to initialize the validation rules for an application.

Adding the Plugin to the Configuration File

The plugin must be declared in the Struts configuration for the framework to be aware of it and initialize it at startup. It's specified in the configuration file using the `plug-in` element:

```
<plug-in
    className="com.oreilly.struts.storefront.framework.ejb.JNDIConnectorPlugin"/>
```

You can also pass properties to your `Plugin` class by using the `set-property` element. For more information on configuring the `Plugin` element or passing properties to an instance, see *The Struts Configuration DTD* in Chapter 4.

Extending the Struts Configuration Classes

One of the biggest changes to version 1.1 of the Struts framework is the `org.apache.struts.config` package. This package contains all of the classes that are used as in-memory representations of the information stored in the Struts configuration file. They represent everything from `Action` configurations to `Plugin` configurations.

If you look back at Chapter 4, you'll see that most of the configuration elements in the Struts configuration file allow you to supply a fully qualified Java class name for the configuration class through the `className` attribute. This gives you the freedom to customize the configuration element and pass additional information.

For example, suppose that you want to pass an additional parameter to your `Action` classes. By default, the `ActionMapping` class is used, which extends `ActionConfig` from the `config` package. If you wanted to pass an additional parameter called `ssl-required` that controlled whether HTTP or HTTPS was used, you could extend the `ActionMapping` class and configure this extension through the `className` attribute. Through this mechanism, the ability to extend the Struts configuration elements makes the framework extremely extensible and flexible to meet just about any application need.

Controller Extension Points

The next set of possible extension points is within the controller layer. Some of these have been briefly mentioned in previous chapters, but are repeated here for completeness.

Extending the `ActionServlet`

In earlier versions of the Struts framework, it was almost a given that an application needed to extend the `ActionServlet`. Most of the controller functionality, excluding the `Action` class behavior, was present in this class. With Struts 1.1, this is no longer true. However, there are a few good reasons why you might need to extend the `ActionServlet`.

As was pointed out in Chapter 5, the initialization routines that get invoked when a Struts application is first launched, reside in the `ActionServlet`. If you needed to modify the way in which the framework initializes itself, this would be the place to change it. To extend the `ActionServlet`, just create a subclass of the `org.apache.struts.action.ActionServlet` class. You can then override the method or methods that you need to function differently. Once this is done, you need to modify the deployment descriptor, so that the Struts application will use your custom `ActionServlet`:

```
<servlet>
  <servlet-name>storefront</servlet-name>
  <servlet-class>
    com.oreilly.struts.storefront.framework.ExtendedActionServlet
  </servlet-class>
</servlet>
```

Most of the runtime request processing behavior has been moved to the `RequestProcessor` class. If you need to customize the manner in which your Struts application processes a request, it is discussed next.

Extending the RequestProcessor

If you need to override the functionality within the `RequestProcessor` class, you need to let the framework know that it should be using your customized version, rather than the default. The framework can be made aware of your specialized `RequestProcessor` by modifying the configuration file for the Struts application. If your configuration file doesn't already have a `controller` element within it, you'll need to add one. There are several attributes that can be configured within the `controller` element. See chapter 4 for more details on these attributes:

```
<controller
  contentType="text/html; charset=UTF-8"
  debug="3"
  locale="true"
  nocache="true"
  processorClass="com.oreilly.struts.framework.CustomRequestProcessor"/>
```

The `processorClass` attribute allows you to specify the fully qualified Java class name of your specialized `RequestProcessor`. The Struts framework will create an instance of your specialized `RequestProcessor` at startup, and use it to process all of the requests for the application. Since each sub-application can have its own Struts configuration file, you can specify a different `RequestProcessor` for each sub-application.

Using the `processPreprocess()` method

There are many methods that can be overridden within the `RequestProcessor` class. One of the methods within the `RequestProcessor` class that is designed with extension in mind is the `processPreprocess()` method. This method is called for each and every request and by default, does nothing with the request. However, you can utilize this method in various ways to change the default request processing behavior. The method, which is shown here:

```
protected boolean processPreprocess( HttpServletRequest request,
                                     HttpServletResponse response){
    return (true);
}
```

is called early in the request processing stage, before the `ActionForm` is called, and also before the `execute()` method is called on the `Action` object. By default, the `processPreprocess()` method returns true, which tells the `RequestProcessor` to continue processing the request. However, you can override this method and perform some type of conditional logic, and if the request should not be processed any further, you can return false. When false is returned from the `processPreprocess()` method, the `RequestProcessor` will stop processing the request and simply return from the `doPost()` or `doGet()` call. Because of this, it's up to you to programmatically forward or redirect the request somewhere.

Example 9-3 illustrates an example of this approach. This example checks to see if there is an existing user session for the request. If so, it will return true so the request will continue. If there is no session, or if the session was just created, the request is redirected to the login page and false is returned from the method.

Example 9-3. Using the `processPreprocess()` method

```
protected boolean processPreprocess( HttpServletRequest request,
                                    HttpServletResponse response){

    boolean continueProcessing = true;
    HttpSession userSession = request.getSession(false);
    // Check to see if there's a session for the user that wasn't just created
    if ( userSession == null || userSession.isNew() ){
        continueProcessing = false;
        response.sendRedirect( "/login.jsp" );
    }
    // Tell the RequestProcessing to continue processing the request or not
    return continueProcessing;
}
```

The `processPreprocess()` method in Example 9-3 checks to make sure that the user has a valid session. Depending on the servlet container, it may not be enough just to check whether there is a session or not. Some containers will create a new session for the request, if one doesn't already exist. That's the reason that Example 9-3 is checking for both conditions.

The most important thing to note from this example is that even though the `processPreprocess()` is returning false, it's still up to the method to take care of redirecting the request. Returning false just lets the `RequestProcessor` know that it doesn't need to continue, but it doesn't do anything with the request; that's the responsibility of the `processPreprocess()` method.

The manner in which Example 9-3 specifies the path in the `sendRedirect()` method is not the best approach. This redirect would only work if the example was running as the default web application and there was a `login.jsp` page in the root directory. It was done this way to keep the example simple. A better approach would be to retrieve the `ActionForward` for the login page and use the path from it in the `sendRedirect()` method. That way, if the actual page changed for the forward, this method doesn't have to be modified.

There are other ways that you can perform the same logic, without using the `processPreprocess()` method. One alternative is to use a Servlet Filter that's part of the Servlet 2.3 API. Filters allow you to inspect the request before it ever reaches the Struts controller. However, there are two problems to be aware of with filters.

First, because filters are part of the 2.3 API, if you are using a Servlet container that only supports 2.2, then you will not be able to use them. The second problem is that since the filter inspects the request very early in the processing stage, filters don't have easy access to the Struts API. It becomes very hard to lookup `ActionForwards` or anything else that you might normally use in the `processPreprocess()` method. In fact, since the Struts controller hasn't even seen the request at the time the filter inspects it, the controller hasn't had a chance to select the proper sub-application

Extending the Base Action Class

There have been several places in previous chapters where I've mentioned a technique of creating a base `Action` that extends the Struts `Action` class, and then using it as a superclass for other actions. One of the reasons for doing this is that in many applications, there is common logic that must be implemented by most of the `Action` classes. By letting this super `Action` class contain most of this code, the redundancy can be eliminated. Example 9-4 provides a very simple version of a base `Action` that performs this type of role.

Example 9-4. A Base Action Class

```
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import java.util.Locale;
import java.util.Iterator;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import com.oreilly.struts.storefront.framework.util.IConstants;
import com.oreilly.struts.storefront.framework.exceptions.*;
import com.oreilly.struts.storefront.framework.UserContainer;
import com.oreilly.struts.storefront.service.IStorefrontService;

/**
 * An abstract Action class that all store front action classes can extend.
 */
abstract public class StorefrontBaseAction extends Action {
    /**
     * The default execute() method that all actions must implement.
     */
    public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) throws Exception{

        // It just calls a worker method that contains the real execute logic
        return executeAction( mapping, form, request, response, getUserContainer(request));
    }

    /**
     * The actual do work method that must be overridden by the subclasses.
     */
}
```

```
abstract public ActionForward executeAction( ActionMapping mapping,
                                           ActionForm form,
                                           HttpServletRequest request,
                                           HttpServletResponse response,
                                           UserContainer userContainer )

    throws Exception;

// Notice this super Action is also a good place to put common utility methods

public boolean isLoggedIn( HttpServletRequest request ){
    UserContainer container = getUserContainer(request);
    return ( container != null && container.getUserView() != null );
}

/**
 * Retrieve the UserContainer for the user tied to the request.
 */
protected UserContainer getUserContainer(HttpServletRequest request) {
    HttpSession session = request.getSession();
    UserContainer userContainer =
        (UserContainer)session.getAttribute(Constants.USER_CONTAINER_KEY);

    // Create a UserContainer if one doesn't exist already
    if(userContainer == null) {
        userContainer = new UserContainer();
        userContainer.setLocale(request.getLocale());
        session.setAttribute(Constants.USER_CONTAINER_KEY, userContainer);
    }
    return userContainer;
}
}
```

The `StorefrontBaseAction` class shown in Example 9-4 illustrates how you can use a base `Action` to perform repetitive behavior that all of the `Action` classes need not perform.

Suppose for example that you all of your `Action` classes needed to obtain the `UserContainer` for the current user and use the information within in; like user ID or security permissions, for example. One approach is to force all of the `Action` classes to obtain the `UserContainer` on their own, handle the situation where there isn't one, and so on. An alternate, more manageable approach is to put that behavior in a super `Action`, and pass the `UserContainer` to the subclasses.

Since the superclass must implement the `execute()` method in order to be called before the subclasses, we will need to invoke a different method on the subclasses. We've called this additional method `executeAction()`, since that is what its doing. You can call it whatever you like.

As Example 9-4 shows, the `StorefrontBaseAction` implements the `execute()` method but inside the `execute()` method, it gets an instance of a `UserContainer` and passes it as an argument to the `executeAction()` method. Each subclass

implements the abstract `executeAction()` method and has the `UserContainer` passed in, instantiated, and guaranteed not to be null. This is only a trivial example of what you can do. Any behavior that all actions need to perform should be a candidate for being implemented in the super `Action` class. When it comes time to modify the implementation or add additional, only the behavior in the super class and `executeAction()` needs to change.

Extending View Components

There is probably less reason or need to extend any of the components located within the view layer. Typically, views are written exclusively for an application. For example, it's unlikely that a JSP page written for one application will be used within a different application. This is not always the case, but differences between look and feel and content, makes it improbable. The one area within the Struts view layer where extensions are often created is the JSP tag libraries.

Extending Struts Custom Tags

The custom tags provided by the Struts framework can be reused across applications and application domains. Therefore, it makes sense that customization and extensions are more likely with these components than with JSP pages. Since the tag handlers are regular Java classes, specialization is achieved through subclassing.

Although you can extend any tag that you need, generally speaking, the HTML tag library will be the one that you'll most likely need to customize, if any; mainly because the custom tags within this library have the greatest impact on the view content.

Regardless of the tag that you extend, you'll need to create your own tag library to hold your tag extensions.

You could actually modify the Struts tag libraries and include your new tag class, but that would make upgrade to newer versions of the Struts framework that much harder. You're better off creating your own tag library that contains just your application's tags.

Once you've created a `.tld` file for you extensions and registered it with the web application deployment descriptor, you are free to use your tags, just as you would any other one.

Extending the Model Components

Since the Struts framework doesn't provide a great deal of components for the model layer, extensions to these components are better discussed in other Java programming books. There are two classes that should be thrown into this category, although they might not be the best representation of what a model component is; it does fall into that classification since it's holding model state.

UserContainer and ApplicationContainer Classes

We've made mention of the `UserContainer` and `ApplicationContainer` throughout the previous chapters without defining exactly what these are. These two classes have less to do with Struts applications and more to do with web applications in general. The purpose of these classes is to store user and application specific information in instances of these classes, rather than in the `HttpSession` and `ServletContext` objects respectively.

One of the problems with storing data into the `HttpSession` for example, is that the interface to store and retrieve data from the session object is not strongly typed. In other words, the interface for any data is:

```
public void setAttribute( "permissionsKey", permissions );
public Object getAttribute( "permissionsKey" );
```

The client must know some magical key, in order to put and retrieve an object from the storage. Some programmers desire a stronger-typed interface instead. Something like this:

```
userContainer.setPermissions( permissions );
userContainer.getPermissions();
```

Here the client doesn't have to worry about what key the object is being stored under, or in fact, how the data is being stored. It could be an `HttpSession` object, or some other data store. The point is that the client is made aware of this because they are not forced to use the methods of the `HttpSession` directly.

There's nothing really complicated about the `UserContainer` class itself. It's an ordinary JavaBean that contains instance variables, along with public getters and setters for the properties. Example 9-5 illustrates a basic `UserContainer` class.

Example 9-5. A Basic UserContainer Class

```
package com.oreilly.struts.storefront.framework;
import java.util.Locale;
import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;
import com.oreilly.struts.storefront.customer.view.UserView;

/**
```

```
* Used to store information about a specific user. This class is used
* so that the information is not scattered throughout the HttpSession.
* Only this object is stored in the session for the user. This class
* implements the HttpSessionBindingListener interface so that it can
* be notified of session timeout and perform the proper cleanup.
*/
public class UserContainer implements HttpSessionBindingListener {

    // The user's shopping cart
    private ShoppingCart cart = null;
    // Data about the user that is cached
    private UserView userView = null;
    /**
     * The Locale object for the user. Although Struts stores a Locale for
     * each user in the session, the locale is also maintained here.
     */
    private Locale locale;

    /**
     * Default Constructor
     */
    public UserContainer() {
        super();
        initialize();
    }

    public ShoppingCart getCart() {
        return cart;
    }

    public void setCart(ShoppingCart newCart) {
        cart = newCart;
    }

    /**
     * Set the locale for the user.
     */
    public void setLocale(Locale aLocale) {
        locale = aLocale;
    }

    /**
     * Retrieve the locale for the user.
     */
    public Locale getLocale() {
        return locale;
    }

    /**
     * The container calls this method when it is being unbound from the
     * session.
     */
    public void valueUnbound(HttpSessionBindingEvent event) {
        // Perform resource cleanup
        cleanUp();
    }
}
```

```
}

/**
 * The container calls this method when it is being bound to the
 * session.
 */
public void valueBound(HttpSessionBindingEvent event) {
    // Don't need to do anything, but still have to implement the
    // interface method.
}

public UserView getUserView() {
    return userView;
}

public void setUserView(UserView newView) {
    userView = newView;
}

/**
 * Initialize all of the required resources
 */
private void initialize() {
    // Create a new Shopping cart for this user
    cart = new ShoppingCart();
}

/**
 * Clean up any open resources. The shopping cart is left intact
 * intentionally.
 */
public void cleanUp() {
    setUserView( null );
}
}
```

One thing to notice is that the `UserContainer` class in Example 9-5 implements the `HttpSessionBindingListener` interface. The methods of this interface allow the `UserContainer` to be notified when it is bound and unbound from the session. This allows it to perform any cleanup on the object that might be necessary.

The `ApplicationContainer` is used for a similar purpose, but at the application level, not the session. It's useful for storing or caching information that is needed by all users across the application. Things like selection lists, configuration properties, and other non-client specific data that you need to get once and hold onto is a candidate for the `ApplicationContainer` class.

Downsides to Extending the Framework

There are of a few downsides to customizing or extending a framework. This may seem contradictory to the previous material, because it suggested that customization is a forecasted goal of using a framework. However, like other things in software development, there are tradeoffs.

When extending a framework, one of the biggest issues that you might have to face is what to do when newer versions of the framework are made available. Unless careful attention had been paid to backwards compatibility, your application might no longer work correctly using a newer version of the framework. The Struts framework for example, underwent some significant changes to its APIs between 1.0 and 1.1. In particular, the `perform()` method is no longer the preferred method that the controller uses to invoke the `Action`; instead the `execute()` method is used. Fortunately, the developers working on the Struts framework were careful and ensured the functionality was compatible with applications built using earlier versions.

You must also take that same care when building your applications. If for example, you override methods of the Struts framework to achieve specialized behavior, it's not out of the realm of possibility that the method is deprecated or removed in future versions. In fact, there are quite a few comments in the framework source that indicate certain portions of the Struts framework will eventually be retired. Although it's nearly impossible to protect your application from all of potential changes, it's best that you go into it with your eyes wide open. Using a framework, even one that is as good and as complete as Struts currently is, is not a silver bullet. You will still have the same issues of upgrades, whether you build your own framework, or use one provided by another source.

10

Exception Handling

Throwing exceptions is Java's way of informing dependent clients that something abnormal occurred during the processing of a method. The client is notified of the type of problem by an instance of a specific exception being thrown. It's entirely up to the client what course of action to take when an exception occurs. In some cases, the client may choose not take any action, which causes the JVM to continue to search for a handler for the exception.

Handling exceptions within your Struts applications is not much different. When an abnormal condition occurs, an exception is thrown to the calling client to notify it of the abnormality. Where things do get different for web applications, and specifically the Struts framework, is what action is taken on behalf of the client and how these exceptions are reported back to the end user.

This chapter will look at how you can properly use the Java exception handling mechanism within your Struts applications to help make your applications more industrial-strength and allow them to gracefully respond when things don't go as expected. Special attention will be given to the differences between performing the exception handling programmatically and using the new declarative feature added to the new version of Struts.

Java Exception Handling

Before we dive into how best to handle exceptions in the Struts framework, it would be a good idea for you to get a picture in your mind of what actually occurs when a method throws an exception. An understanding of the complications that take place in the JVM when exceptions occur, may enlighten you as to the importance of throwing exceptions

for the right reason as well as the importance of throwing the right exceptions. Because there is additional overhead for the JVM to handle an exception, you should always take care to use exceptions correctly.

Java Exceptions

In Java, exceptions are objects that are created when an abnormal* condition occurs in a running application. When a Java application throws an exception, it throws an object that is a descendant of `java.lang.Throwable`. The `Throwable` class has two direct subclasses, `java.lang.Error` and `java.lang.Exception`. Figure 10-1 shows a partial hierarchy tree for the `Throwable` class.

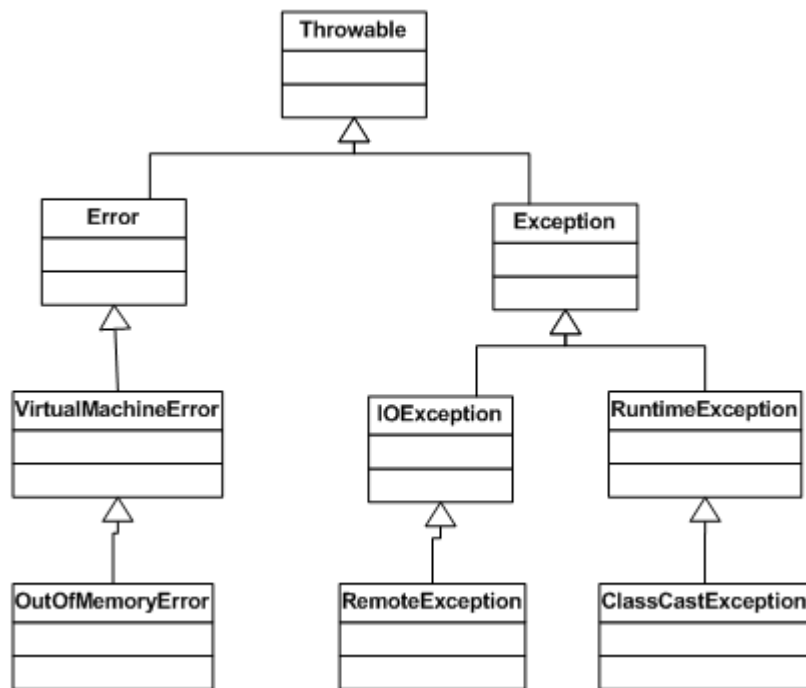


Figure 10-1. A partial class hierarchy for the `Throwable` class

Obviously not all of the descendants of the `Throwable` class can be shown. There are more than a hundred direct and indirect subclasses in the core Java library alone. The members of the `Exception` family tree are normally thrown to indicate an abnormal condition that can usually be handled by the application. Most of the exceptions that your Struts application creates and throws should be subclasses of the `Exception` class. The other branch of `Throwable`, which is the `Error` class and all of its descendants, is

* The abnormal condition is commonly referred to as an “exception condition”.

reserved for more serious problems that occur during an application's life cycle. For example, if there's no more memory available for an application, an `OutOfMemoryError` will occur. Typically, there's nothing that a client can do when this error occurs, therefore clients don't generally worrying about handling the subclasses of `Error`. In most cases, it's the JVM itself that throws instances of `Error` or its subclasses.

The Method Invocation Stack

The JVM uses a *Method Invocation Stack*, also referred to as a call stack, to keep track of the succession of method invocations of each thread. The stack is used to keep local information about each method that has been called, going all of the way back to the original `main()` method of the application. When each new method is invoked, a new stack frame is pushed onto the top of the stack and the method becomes the current executing method. The local state of each method is also saved with each stack frame. Figure 10-2 illustrates an example Java call stack.

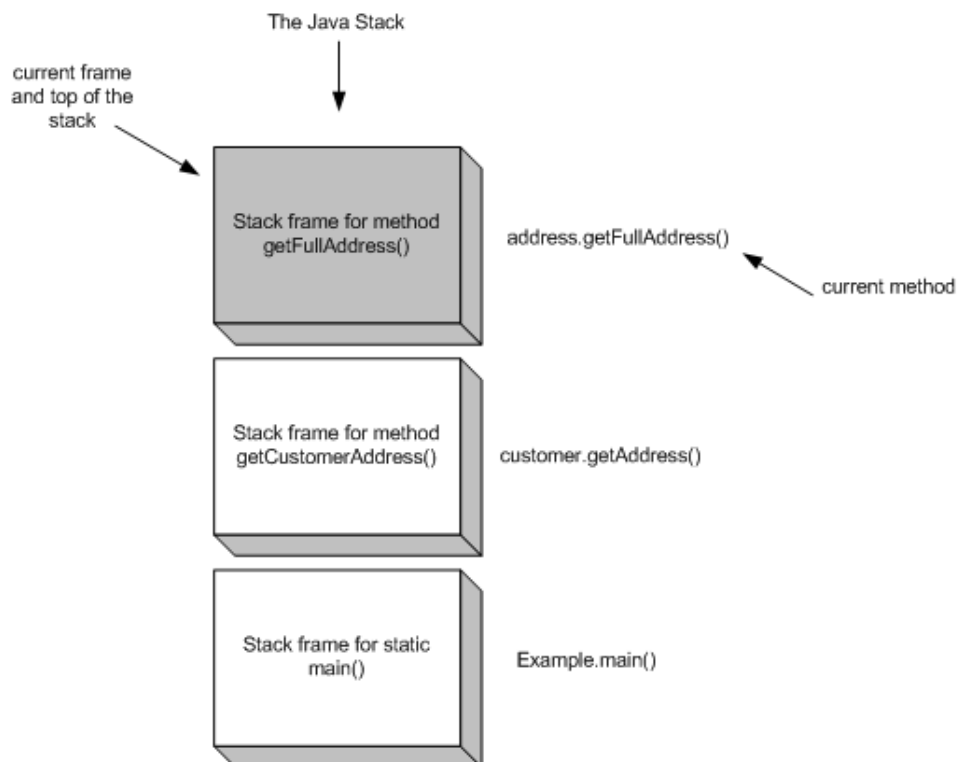


Figure 10-2. An example of a Java method invocation stack (call stack)

When a Java method completes normally, the JVM pops the current method's stack frame from the stack and continues processing in the previous method where it left off.

When an exception condition occurs however, the JVM must find a suitable exception handler for it. It first checks to see if the current method catches the exception or one of its parent exceptions. If so, execution will continue in that catch clause. If the current method doesn't provide a catch clause to handle the exception raised, the JVM will start popping method frames off the call stack until it finds a handler for the exception. Eventually, if it pops its way all the way back to the `main()` method and still doesn't find a handler for the exception, the thread will terminate. If that thread is the main thread and there are no other non-daemon running threads, the application itself may terminate. If the JVM does find an exception handler along the way, that method frame will become the top of the stack and execution will continue on from there.

The importance of understanding how the JVM handles exceptions is to realize that there's plenty going on underneath the hood when exceptions occur in your applications. It can be a lot of work for the JVM to locate an exception handler for a particular exception, especially if the handler is located far down the call stack. It's very important that you provide sufficient exception handlers at the appropriate levels. If you let exceptions go, it's very likely that they can halt your application.

What about the *throws* Clause?

When determining the method signatures for the classes that are part of the application, deciding which exceptions will be thrown from the methods should get as much attention as what the parameters are and what the return type is.

You might have heard of the concept “*Design by Contract*”. The idea behind this principle is that the set of public methods that a class exposes represents a virtual contract between a client and the class itself. The client has certain obligations in the way in which it invokes the method. There may also be requirements on the class itself as part of the contract.

When something abnormal occurs and an exception is thrown from a method in the class, the contract, in a sense, has been broken. The class is informing the client that it can't fulfill its terms of the contract. It's entirely up to the caller to decide how to handle the exception. This is why the throws clause of a method signature is so important. It forces a client to decide what it will do when one of these abnormal conditions occur. However, as you'll see in the next section, all Java exceptions are not equal.

Checked and Unchecked Exceptions

Java exceptions can be separated into two distinct, but important groups, *checked* and *unchecked*. A checked exception signals an abnormal condition that you want the client to have to deal with. All checked exceptions must either be caught and handled within the calling method, or declared in the throws clause of the method. This is why they are called checked. The compiler and the JVM will verify that all of the checked exceptions that can occur in a method are taken care of. The compiler and JVM don't care if

unchecked exceptions are ignored, because these are exceptions that the client probably couldn't do anything about anyway. Unchecked exceptions are usually the result of incorrect logic or programming errors. These are exceptions like `java.lang.ClassCastException`.

The determination of whether an exception is checked or unchecked is simply based on its location in the exception hierarchy. All classes that are descendants of the `java.lang.Exception` class, except for subclasses of `RuntimeException`, are checked exceptions and the compiler will ensure that they are either handled by the method or listed in the throws clause. `RuntimeException` and its descendants are unchecked exceptions and the compiler will not complain about these not being in a throws clause for a method or handled in a try/catch block.

Performance Impact of Exception Handling

In general, wrapping your Java code with try/catch blocks doesn't have a significant performance impact on your applications. Only when exceptions actually occur, is there a negative performance impact to your applications. The impact is due to the lookup that the JVM has to do to locate the proper handler for the exception. If the catch block for the exception is located in the same method, the impact is not so bad. However, the further down the call stack the JVM has to go to find the exception handler, the greater the impact becomes.

This is why you should only use a try/catch block for error conditions that may occur in your application. You should never use exceptions for things such as controlling program flow. The following use of a try/catch is probably fine, but getting very close to what might be considered a waste of exception handling.

```
Double basePrice = null;
String basePriceStr = request.getParameter( "BASE_PRICE_AMOUNT" );

// Use a try/catch to make sure the value is a number
try{
    basePrice = Double.valueOf( basePriceStr );
}catch( NumberFormatException ex ){
    // The value could not be converted to a valid Double, set the default
    basePrice = ApplicationDefaults.DEFAULT_BASE_PRICE;
}
```

The previous code fragment shows a try/catch block determining an error condition and then taking corrective actions. The error condition is an invalid price value and the corrective action is to assign a default value. There are other ways to determine whether a string is a valid double value, but using this approach is fairly popular. Fortunately for this approach, the exception handler is located in the same method and the JVM doesn't incur a large penalty for this occurrence.

Of course, the rules are somewhat subjective and what's a valid reason to one developer may not be to another. You should be aware of the issue and avoid using try/catch blocks for something other than actual error conditions.

System versus Application Exceptions

Exceptions can further be classified into one of two categories, *System Exceptions* and *Application Exceptions*. System exceptions are those exceptions that are more serious in nature. These are typically low-level problems that aren't related to the application logic and ones that end-users are not expected to be able to recover from. In many cases, system exceptions are *unchecked* and your application isn't supposed to catch them because they are either programming errors or they are so severe that nothing can be done about them.

Application exceptions are problems that occur that represent a violation of a business rule or some other condition in the application logic. As an example, you might throw an application exception when a user attempts to login to the application, but their account has been locked. This isn't a catastrophic event, but it is a problem that needs to be reported and dealt with.

Within Struts applications, and web applications in general, there are essentially two approaches that you can take when an exception occurs. If the exception is an application exception, where the end-user may be able to recover from, then you generally want to return control back to the input page and display a user-friendly message to the user informing them of the problem and some action that can be taken to resolve it. Continuing with the locked account example from the previous paragraph, you could throw an `AccountLockedException` back to the action class, which would forward control back to the login page informing the user that the account is locked.

If the thrown exception is a low-level exception like a `RemoteException`, the only real action that the application can take is to display a system error page. There's nothing the user can do to fix the problem. It may be a programming error or some type of network issue, but the point is that you don't want to let the user see the stack trace of the exception, so you should instead forward to a system error page that's more user-friendly to look at and maybe informs the user to notify the system administrator. The exception should also be logged to aid in determining the root cause of the problem.

Later in this chapter, you'll see examples of how to return control back to the input page and show a localized message to the user. You will also see ways of dealing with system errors by forwarding control to a system error page. All of which, will add value to the application and to the user experience.

Using Chained Exceptions

It's often necessary and suitable to catch a particular type of exception and re-throw a different one. This is sometimes necessary because a client might not know or care to handle the original exception. For example, let's say that a client invokes an action on a Struts application to upload an image file to a database. Let's further assume that the action class calls an update method whose signature looks like the following:

```
public void updateImageFile( String imagePath ) throws UploadException;
```

When the method is called with an image to upload and a problem occurs, an `UploadException` will be thrown. However, the underlying problem will be more specific like the file system is full or the database already has the image, depending on the destination of the image upload. The original exception thrown may be `IOException` or `SQLException`, but the user doesn't really know or care about this level of detail. All they care about is the update function failed. Although the end-user doesn't care about the specific exception, the system administrator or the developers who will be assigned the task to debug and fix the problem do care. That's the reason that you don't want to throw away the root cause of the problem, when you create the alternate exception.

Prior to version 1.4, Java didn't provide a built-in mechanism to wrap the original exception with a new one. Developers were left to their own devices to solve the problem. Most homegrown solutions looked something like the exception class in example 10-1.

Example 10-1. An example of Exception class that supports chained exceptions

```
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 * This is the common superclass for all application exceptions. This
 * class and its subclasses support the chained exception facility that allows
 * a root cause Throwable to be wrapped by this class or one of its
 * descendants.
 */
public class BaseException extends Exception {
    protected Throwable rootCause = null;
    protected BaseException( Throwable cause ) {
        this.rootCause = rootCause;
    }

    public void setRootCause(Throwable anException) {
        rootCause = anException;
    }

    public Throwable getRootCause() {
        return rootCause;
    }

    public void printStackTrace() {
```

```
        printStackTrace(System.err);
    }

    public void printStackTrace(PrintStream outStream) {
        printStackTrace(new PrintWriter(outStream));
    }

    public void printStackTrace(PrintWriter writer) {
        super.printStackTrace(writer);

        if ( getRootCause() != null ) {
            getRootCause().printStackTrace(writer);
        }
        writer.flush();
    }
}
```

The exception class in example 10-1 allows you to wrap the original `Throwable` with an instance of this exception class or any of its descendants. The nice thing about this feature is that it allows you to abstract out the ugly details of lower level exceptions, while at the same time keeping those details available so they can be printed out to a log and used by developers. Because these exceptions can be chained together endlessly, this concept is commonly referred to as *exception chaining*.

Exception chaining is an excellent way of not letting lower layer abstractions, like JDBC access, to propagate outward to an end user. The end user doesn't care about the lower layer problem and abstracting the problem to a higher level exception will keep them from seeing too much of the details.

Another benefit of using a higher layer exception class is so that the API of the upper layer is not tied or coupled to the details of the implementation. Suppose that a file system is initially used to store the images and therefore the throws clause of the `updateImageFile` method from before declares that it throws `IOException`. If later the implementation changed to use JDBC instead, the throws clause and the clients that invoked the method would have to be changed to declare or catch `SQLException`. By using a higher level of abstraction, the client only needs to be concerned about the `UploadException`, regardless of the underlying implementation.

Dealing with Multiple Exceptions

A slight variation on the exception chaining idea is the concept of throwing multiple exceptions from a method. For example, let's say a user is filling out a form that has several price fields that must fall between some minimum and maximum values. Let's further assume that the price values can only be validated on the backend and can't be validated in an `ActionForm`.

Unless your application can throw multiple exceptions from a method, the user will only see one exception at a time. This approach will work, but may become very

annoying for the end-user. They have to fix one field and then re-submit only to receive the next error. It would be easier for the user if all of the errors are displayed and could be fixed at the same time. Unfortunately, a Java method can only throw a single instance of `Throwable`.

One of the solutions to do this is to allow an exception class to have a primary exception and also support a collection of other exceptions. Each exception can be treated the same, but the primary exception is used when only a single exception occurs and the client can check the exception collection to see if there are more. Example 10-2 illustrates what the `BaseException` class from example 10-1 would like with this feature added to it.

Example 10-2. An example of an exception class that supports multiple nested exceptions

```
package com.oreilly.struts.framework.exceptions;

import java.util.Collection;
import java.util.ArrayList;
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 * This is the common superclass for all application exceptions. This
 * class and its subclasses support the chained exception facility that allows
 * a root cause Throwable to be wrapped by this class or one of its
 * descendants. This class also supports multiple exceptions via the
 * exceptionList field.
 */
public class BaseException extends Exception{
    protected Throwable rootCause = null;
    private Collection exceptions = new ArrayList();

    public BaseException(){
        super();
    }

    public BaseException( Throwable cause ) {
        this.rootCause = rootCause;
    }

    public Collection getCollection() {
        return exceptions;
    }

    public void addException( BaseException ex ){
        exceptions.add( ex );
    }

    public void setRootCause(Throwable anException) {
        rootCause = anException;
    }

    public Throwable getRootCause() {
        return rootCause;
    }
}
```



```
    }

    public void printStackTrace() {
        printStackTrace(System.err);
    }

    public void printStackTrace(PrintStream outStream) {
        printStackTrace(new PrintWriter(outStream));
    }

    public void printStackTrace(PrintWriter writer) {
        super.printStackTrace(writer);

        if ( getRootCause() != null ) {
            getRootCause().printStackTrace(writer);
        }
        writer.flush();
    }
}
```

Notice in example 10-2 that a `java.util.Collection` has been added to the class. If more than one exception occurs during the processing of a method, the additional exceptions can be added to the collection and returned to the client. If the client wishes only to deal with a single exception, it doesn't have to retrieve the additional ones.

[\[Note: Reviewers, I'm planning to strip out a great deal of the information from the previous section to shorten this chapter, but also because this material can be found in other Java books. If there is a certain section that you feel needs to stay in, make a comment to me and I will try to keep that in. If you think most if not all, needs to be cut, please let me know that as well.\]](#)

Exception Handling provided by Struts

Prior to version 1.1, the Struts framework provided very minimal exception handling for applications. It was basically left up to you to extend the framework with your own exception handling capabilities. This encouraged each and every development group to come at the approach from different directions and made it difficult to discuss common solutions.

Starting with version 1.1, Struts now provides a small, but effective exception-handling framework for your applications. The approach that the Struts designers have taken follows in the footsteps of the EJB and Servlet specifications have taken for handling security, by allowing developers to use either a declarative and/or a programmatic approach.

Declarative versus Programmatic Exception Handling

Declarative exception handling is done by expressing an application's exception handling policy, including which exceptions are thrown and how they are to be handled, in a form that is completely external to the application code. This approach makes it easier to modify the exception handling logic, without major recompilation of the code.

Programmatic exception handling is quite the opposite. It involves writing application-specific code to handle the exceptions, rather than just configuring an external configuration file.

As with other Struts configuration options, the declarative mappings are done in the Struts configuration file. As you saw in chapter 4, you're able to specify the exceptions that may occur and what to do if they do occur, both at a global level and for a specific action mapping. For a more complete discussion of the parameters available for the exception handling elements, refer back to chapter 4.

Example 10-3 shows a partial Struts configuration file that declares three different exceptions that may be thrown from the *login* action.

Example 10-3. A Struts Configuration File that Uses Declarative Security Mappings

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <action-mappings>
    <action
      path="/login"
      type=" com.oreilly.struts.storefront.security.LoginAction"
      name="loginForm"
      scope="request"
      input="/login.jsp">

      <!--The following exceptions can be thrown during the login action -->
      <exception
        key="security.error.changepassword"
        path="/changePassword.jsp"
        type=" com.oreilly.struts.framework.exceptions.ExpiredPasswordException"/>
      <exception
        key=" security.error.loginfailed "
        type=" com.oreilly.struts.framework.exceptions.InvalidLoginException"
        path="/login.jsp"/>
      <exception
        key=" security.error.accountlocked "
        type=" com.oreilly.struts.framework.exceptions.AccountLockedException"
        path="/accountLocked.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

```
</action>
</action-mappings>
</struts-config>
```

The `exception` element that is defined either in the action mapping or in the global exceptions section specifies the path to forward to when one of the specified exceptions occurs during the corresponding action invocation. For example, if during the `login` action, an `ExpiredPasswordException` is thrown, the controller will forward control to the `changePassword.jsp` page. Likewise, if an `AccountLockedException` is thrown, control will forward to the `accountLocked.jsp` page.

When an exception is not programmatically handled in the action class, the `RequestProcessor` gets a chance to see if there is an `exception` element configured for that specific exception type. If there is, then control is forwarded to the resource specified in the `path` attribute of the `exception` element. Example 10-4 shows the `processException()` method from the `RequestProcessor` class.

Example 10-4. The processException method from the Struts RequestProcessor

```
protected ActionForward processException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Exception exception,
                                         ActionForm form,
                                         ActionMapping mapping)
    throws IOException, ServletException {

    // Is there a defined handler for this exception?
    ExceptionConfig config = mapping.findException(exception.getClass());

    if (config == null){
        if (log.isDebugEnabled()){
            log.debug(getInternal().getMessage("unhandledException",exception.getClass()));
        }
        if (exception instanceof IOException){
            throw (IOException) exception;
        }else if (exception instanceof ServletException){
            throw (ServletException) exception;
        }else{
            throw new ServletException(exception);
        }
    }

    // Use the configured exception handling
    try {
        Class handlerClass = Class.forName(config.getHandler());
        ExceptionHandler handler = (ExceptionHandler)handlerClass.newInstance();
        return (handler.execute(exception, config, mapping, form,request, response));
    }catch (Exception e){
        throw new ServletException(e);
    }
}
```

Notice how an `ExceptionConfig` object may be returned from the `findException()` method at the beginning of the `processException()` method. The `ExceptionConfig` object is an in-memory representation of the exception element specified in the configuration file. If the `findException()` method doesn't find an exception element for the specific type of exception that occurred, the exception is thrown back to the client without going through a Struts exception handler. Unless the exception is an `IOException` or one of its subclasses, the exception will be wrapped by a `ServletException` instance and re-thrown.

If there is an `exception` element specified in the action mapping for the specific type of exception that occurs, an `ExceptionConfig` object is returned from the `findException()` method. The `getHandler()` method is then called on the `ExceptionConfig` object and this handler is used to process the exception.

The Struts framework has a default exception handler class that will be used to process the exceptions unless you configure one of your own. The default handler class is `org.apache.struts.action.ExceptionHandler`. The behavior in the `execute()` method of this handler is to create an `ActionError` and store it into the proper scope and return an `ActionForward` object that is associated with the `path` attribute specified in the `exception` element. To summarize, if you declare an `exception` element inside an `action` element, the default exception handler will create and store an `ActionError` into the specified scope and return control to the resource specified in the `path` attribute.

As you saw back in chapter 4, the exception element also allows you to override the exception handler behavior if you want to modify the behavior taken when an exception occurs. You can do this by specifying a fully qualified Java class that extends the `org.apache.struts.action.ExceptionHandler` class and that overrides the `execute()` method. For example, suppose that all of your application exceptions extend the `BaseException` class shown in example 10-5.

Example 10-5. An Exception class that supports a message key and arguments

```
package com.oreilly.struts.framework.exceptions;

import java.util.Collection;
import java.util.ArrayList;
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 * This is the common superclass for all application exceptions. This
 * class and its subclasses support the chained exception facility that allows
 * a root cause Throwable to be wrapped by this class or one of its
 * descendants. This class also supports multiple exceptions via the
 * exceptionList field.
 */
public class BaseException extends Exception{
    protected Throwable rootCause = null;
    private Collection exceptions = new ArrayList();
```

```
private String messageKey = null;
private Object[] messageArgs = null;

public BaseException(){
    super();
}

public BaseException( Throwable cause ) {
    this.rootCause = rootCause;
}

public Collection getCollection() {
    return exceptions;
}

public void addException( BaseException ex ){
    exceptions.add( ex );
}

public void setMessageKey( String key ){
    this.messageKey = key;
}

public String getMessageKey(){
    return messageKey;
}

public void setMessageArgs( Object[] args ){
    this.messageArgs = args;
}

public Object[] getMessageArgs(){
    return messageArgs;
}

public void setRootCause(Throwable anException) {
    rootCause = anException;
}

public Throwable getRootCause() {
    return rootCause;
}

public void printStackTrace() {
    printStackTrace(System.err);
}

public void printStackTrace(PrintStream outStream) {
    printStackTrace(new PrintWriter(outStream));
}

public void printStackTrace(PrintWriter writer) {
    super.printStackTrace(writer);

    if ( getRootCause() != null ) {
```

```
        getRootCause().printStackTrace(writer);
    }
    writer.flush();
}
}
```

The `BaseException` class in example 10-5 contains a `messageKey` that can be used as a key in the Struts resource bundle. This key can be passed into the constructor of the `ActionError` class and the Struts framework will match it to a message in the Struts resource bundle. This class also contains an object array that the creator of the exception can populate. These objects can then be used to substitute into a message from the bundle that contains substitution parameters based on the `MessageFormat` class. A message in the bundle might look this like:

```
| global.error.invalid.price=The price must be between {0} and {1}.
```

When creating an `ActionError` object, you can pass an array of objects as the second parameter and each object will be substituted into the parameters enclosed by the braces. The 0th argument in the array will be inserted into the {0} position, the object at index 1 into the {1} position, and so on. Chapter 12 “Internationalization and Localization” covers this topic in more detail.

Example 10-6 illustrates how to extend the default exception handler class and provide specialized behavior for substituting the arguments from the exception into the `ActionError` constructor.

Example 10-6. A Specialized Exception Handler

```
package com.oreilly.struts.chapter10examples;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ExceptionHandler;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionError;
import org.apache.struts.util.AppException;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.config.ExceptionConfig;

import com.oreilly.struts.framework.exceptions.BaseException;

public class SpecialExceptionHandler extends ExceptionHandler {

    protected ActionForward execute(Exception ex,
                                    ExceptionConfig config,
                                    ActionMapping mapping,
                                    ActionForm formInstance,
                                    HttpServletRequest request,
                                    HttpServletResponse response)
        throws ServletException {
        ActionForward forward = null;
```

```
        ActionError error = null;
        String property = null;

        /* Get the path for the forward either from the exception element
         * or from the input attribute.
         */
        String path = null;
        if (config.getPath() != null) {
            path = config.getPath();
        }else{
            path = mapping.getInput();
        }
        // Construct the forward object
        forward = new ActionForward(path);

        /* Figure out what type of exception has been thrown. The Struts
         * ApplicationException is not being used in this example.
         */
        if( ex instanceof BaseException) {
            // This is the specialized behavior
            BaseException baseException = (BaseException)ex;
            String messageKey = baseException.getMessageKey();
            Object[] exArgs = baseException.getMessageArgs();
            if ( exArgs != null && exArgs.length > 0 ){
                // If there were args provided, use them in the ActionError
                error = new ActionError( messageKey, exArgs );
            }else{
                // Create an ActionError without any arguments
                error = new ActionError( messageKey );
            }
        }else{
            error = new ActionError(config.getKey());
            property = error.getKey();
        }

        // Store the ActionError into the proper scope
        // The storeException method is defined in the parent class
        storeException(request, property, error, forward, config.getScope());

        return forward;
    }
}
```

The specialized behavior that you perform in your handler class is up to you. The behavior shown in example 10-6 is done so that arguments that go with the exception can be inserted into the `ActionError` and used to make the user messages more informative.

For information on how to install a custom exception handler, see “The Struts Configuration DTD” in Chapter 4.

There are plenty of other instances where you might need to override the default behavior. The default exception handler provided by the Struts framework doesn't

support an exception object holding on to multiple exceptions. If your application needs to support this behavior, you are going to need to create your own `ExceptionHandler` class.

In many cases however, the Struts exception handler will be sufficient. Only when you need specialized exception handling that can't be obtained from the Struts exception handler, should you bother to create your own. Figure 10-3 illustrates a sequence diagram for the default exception handling mechanism of Struts.

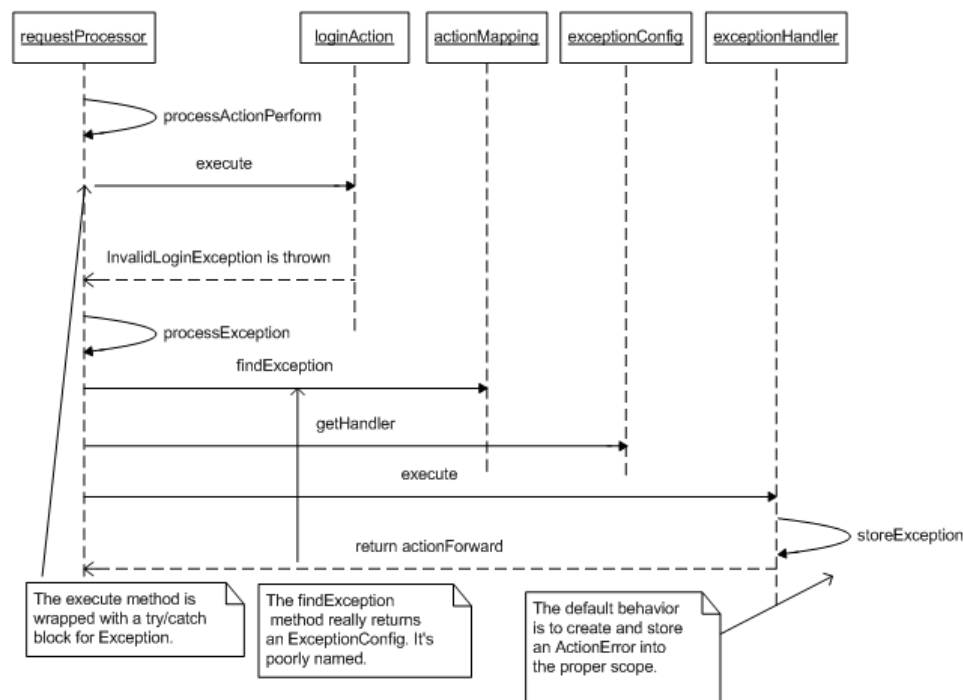


Figure 10-3. A Sequence Diagram for Struts Exception Handling

[\[Note: I will make this figure a little more readable in the final version.\]](#)

Using the declarative exception handling mechanism of Struts does not preclude you from also using a programmatic approach by itself or in conjunction with the declarative option. In fact, they can work quite well together. Your action classes will get the first opportunity to handle any specific exceptions and only if an exception is not caught and handled by the action instance, will it be caught by the `processActionPerform()` method in the `RequestProcessor` class. The `RequestProcessor` can then use the declarative exception handling mechanism if the action instance wasn't interested in handling the exception. The next section discusses how to handle exceptions using a programmatic approach.

Using Programmatic Exception Handling

The alternate approach to the declarative exception handling provided by Struts is to build the application-specific exception handling into the code itself. This means that you will have to extend the framework with behavior specific to your application.

As mentioned earlier in this chapter, there are two basic courses of action when an exception is thrown within an action class. If the exception is an application exception, the course of action is to log the exception, create and store an `ActionError` into the appropriate scope, and forward control to the appropriate `ActionForward`. You'll remember from the discussion of declarative exception handling, this is the same behavior that the Struts default exception handler performs, minus the logging.

In the case of the Storefront application, application exceptions would all be descendants of `BaseException`, therefore it becomes very easy to detect when an application exception occurs because you can simply have a catch block for `BaseException`. If the exception is not an instance of `BaseException`, then it can be assumed that it's a system exception and treated as such. The course of action for system exceptions is normally to log the exception and return an `ActionForward` for the system error page.

At first, you might be tempted to add a try/catch block in your action classes and perform the exception handling like this.

```
try{

    // Perform some work that may cause an application or a system exception

}catch( BaseException ex ){
    // Log the exception

    // Create and store the action error
    ActionErrors errors = new ActionErrors();
    ActionError newError = new ActionError( ex.getErrorCode(), ex.getArgs() );
    errors.add( ActionErrors.GLOBAL_ERROR, newError );
    saveErrors( request, errors );

    // Return an ActionForward for the input resource
    return mapping.getInput();

}catch( Throwable ex ){
    // Log the exception

    // Create and store the action error
    ActionError newError = new ActionError( ex.getErrorCode(), ex.getArgs() );
    ActionErrors errors = new ActionErrors();
    errors.add( ActionErrors.GLOBAL_ERROR, newError );
    saveErrors( request, errors );

    // Return an ActionForward for the system error resource
    return mapping.findForward( IConstants.SYSTEM_FAILURE_PAGE );
}
```

The problem with the approach shown here is that you end up having the same redundant code inside of almost every action class. This is one of the benefits of using the declarative approach. However, if you don't want to use the declarative approach, or you can't because you're using an earlier version of Struts, there's an alternate approach that you can use that doesn't involve so much redundancy.

You've already seen in Chapter 5 how the use of an abstract `BaseAction` class can reduce the redundancy inside the action classes for other issues. You can also push the programmatic exception handling functionally up to the `BaseAction` class and not have to have it in all of your action classes. To do this, you will need to implement the additional `executeAction()` method that was discussed in Chapter 5. Example 10-7 shows the `execute()` method of the `StorefrontBaseAction` class.

Example 10-7. The execute method of the StorefrontBaseAction

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {
    ActionForward forwardPage = null;
    try{
        UserContainer userContainer = getUserContainer( request );

        // Inform the specific action instance to do its thing
        forwardPage = executeAction(mapping, form, request, response, userContainer);
    }catch (BaseException ex){
        // Log the application exception using your logging framework
        // Call the generic exception handler routine
        forwardPage = processExceptions( request, mapping, ex );
    }catch (Throwable ex){
        // Log the system exception using your logging framework

        // Make the exception available to the system error page
        request.setAttribute( Action.EXCEPTION_KEY, ex );

        // treat all other exceptions as a system error
        forwardPage = mapping.findForward( IConstants.SYSTEM_FAILURE_KEY );
    }
    return forwardPage;
}
```

The `execute()` method invokes the `executeAction()` method, which all `Action` subclasses must override, and wraps the invocation with the appropriate try/catch blocks. The `StorefrontBaseAction` is abstract and provides an abstract version of the `executeAction()` method, which is shown here.

```
abstract public ActionForward executeAction( ActionMapping mapping,
                                             ActionForm form,
                                             HttpServletRequest request,
                                             HttpServletResponse response,
                                             UserContainer userContainer )
    throws BaseException;
```

When any application exception occurs, as long as it extends `BaseAction`, it will be caught in the try/catch block inside the `execute()` method. The subclasses don't have to worry about providing a catch block at all, unless they are planning to provide specialized behavior for the exception.

The `execute()` method passed the exception, along with the request and mapping objects, to the `processExceptions()` method. The `processExceptions()` method is shown in example 10-8.

Example 10-8. The `processExceptions` method in the `StorefrontBaseAction`

```
protected ActionForward processExceptions( HttpServletRequest request,
                                           ActionMapping mapping,
                                           BaseException ex ){
    ActionErrors errors = new ActionErrors();
    ActionForward forward = null;

    // Get the locale for the user
    Locale locale = getUserContainer( request ).getLocale();
    if (locale == null){
        // If it hasn't been configured, get the default for the environment
        locale = Locale.getDefault();
    }

    processBaseException(errors, (FieldException) ex, locale);

    // Either return to the input resource or a configured failure forward
    if ( mapping.getInput() != null) {
        forward = new ActionForward( mapping.getInput() );
    }else if (mapping.findForward(IConstants.FAILURE_KEY) != null){
        forward = mapping.findForward( IConstants.FAILURE_KEY );
    }

    // See if this exception contains a collection of sub exceptions
    Collection exceptionCollection = ex.getCollection();
    if (exceptionCollection != null && !exceptionCollection.isEmpty() ){
        int size = exceptionCollection.size();
        Iterator iter = exceptionCollection.iterator();

        while( iter.hasNext() ){
            // All sub exceptions must be BaseExceptions
            BaseException subException = (BaseException)iter.next();
            processBaseException(errors, subException, locale);
        }
    }

    // Tell the Struts framework to save the errors into the request
    saveErrors( request, errors );

    // Return the ActionForward
    return forward;
}
```

The `processExceptions()` method seems to be quite busy, however it's really not that bad. Here are the steps that the method is performing.

1. Obtain the Locale for the user.
2. Call the `processBaseException()` method to process the top-level exception.
3. Determine if there are sub exceptions and if so, process each one.
4. Save all of the `ActionError`'s that were created.
5. Return control back to either the resource identified in the `input` attribute of the action or to a "Failure" `ActionForward` that has been configured for the action.

The `processBaseException()` method is where the `ActionError` objects are created. This method is shown in example 10-9.

Example 10-9. The processBaseException of the StorefrontBaseAction class

```
protected void processBaseException( ActionErrors errors,
                                     BaseException ex,
                                     Locale locale) {

    // Holds the reference to the ActionError to be added
    ActionError newActionError = null;

    // The errorCode is the key to the resource bundle
    String errorCode = ex.getMessageKey();
    /**
     * If there are extra arguments to be used by the MessageFormat object,
     * insert them into the argList. The arguments are context sensitive
     * arguments for the exception, there may be 0 or more.
     */
    Object[] args = ex.getMessageArgs();

    /**
     * In an application that had to support I18N, you might want to
     * format each value in the argument array based on its type and the
     * user locale. For example, if there is a Date object in the array, it
     * would need to be formatted for each locale.
     */

    // Now construct an instance of the ActionError class
    if ( args != null && args.length > 0 ){
        // Use the arguments that were provided in the exception
        newActionError = new ActionError( errorCode, args );
    }else{
        newActionError = new ActionError( errorCode );
    }

    errors.add( ActionErrors.GLOBAL_ERROR, newActionError );
}
```

The `processBaseException()` method is responsible for creating the `ActionError` object using the `messageKey` field to lookup a bundle message and if there are any arguments included, include those in the `ActionError` constructor as well.

As you can see, adding programmatic exception handling to your applications is definitely more work than using the default behavior provided by the Struts framework. It will make maintenance more difficult if you change your exception hierarchy drastically or change how you want to handle certain exceptions. However, if you are using an earlier version of Struts, this may be your only choice. You may have to extend these examples for your own applications, but they do show a well-designed approach that you can build upon.

Within the EJB and Servlet specifications, programmatic security is frowned upon because it's too easy to couple your application to the physical security environment. With exception handling, the possibility of needing to change the exceptions that are thrown based on the target environment, is just not likely. Therefore, there isn't the same stigma associated with programmatic exception handling as there is attached to programmatic security. It is true however, that if you can take advantage of declarative exception handling, your application will be easier to maintain than having the same functionality in your source code. An application will have to be modified over time. New exceptions will be thrown and will need to be caught. The more you can specify declaratively about an application, the easier time you'll have trying to maintain it.

Tying Up the Loose Ends

Before we leave the topic of exception handling there are several special cases that we should discuss. Each one of these is unique and you may or may not have a need for in your applications.

Handling Remote Exceptions

When dealing with remote Java objects, it's possible that they can throw `java.rmi.Remote` exceptions. In fact, every EJB method that is exposed to a remote client must declare that it throws `RemoteException`. Dealing with `RemoteException` is very similar to handling system exceptions, except that it is not a descendant of either `java.lang.Error` or `java.lang.RuntimeException`, as other system exceptions are.

Often, the application will not be able to recover from a `RemoteException` and will have to display the system error page. If you're using EJB and you get a `RemoteException`, you could attempt to recover by acquiring a new remote reference, but there's probably some type of programming or environment error and the end user will not be able to continue. Whether you're using a programmatic or a

declarative approach, you'll likely want to log the exception, create and store an `ActionError` object and then forward to the system error page. You could define the exception handling behavior to forward the user back to the previous page and give them the choice of trying again. If some type of network blip caused the remote exception, it may be possible for the user to continue to use the application.

Exceptions in Custom Tags

JSP custom tags usually throw `JSPException` objects or one of its descendants. Prior to the JavaServer Pages 1.2 Specification, the `JSPException` class didn't support exception chaining. Since Struts was introduced before the 1.2 Specification, there are several places inside of the Struts custom tag libraries that still throw the original exception away when a `JSPException` is created.

However, the tags do usually store the exception in the request scope under the key `Action.EXCEPTION_KEY`, which maps to a literal string of `org.apache.struts.action.Action.EXCEPTION_KEY`. If you need to get access to the root cause, you can probably use this key to retrieve the exception object.

Version 1.2 of the JSP Specification modified the `JSPException` to support exception chaining, however the Struts developers will probably choose to leave the current tags alone for backwards compatibility and only take advantage of the new `JSPException` for future tags. However, in the custom tags that you create, you should utilize the `rootCause` field in the `JSPException` class when you re-throw exceptions as different types.

Internationalized Exception Handling

Chapter 12 covers Internationalization in depth, but it's relevant to say a few words about how exception handling and Internationalization are connected. Often while throwing exceptions in Java, developers will do something like the following.

```
// Detect some problem and throw an exception
throw new Exception( "An exception has occurred." );
```

The problem with hardcoding the string into the exception is that it's only useful for the developers who might read it that are from the same locale. If developers or system administrators from different locales were to try and use the log files where these exceptions are logged, it might be difficult and would probably not provide much help to them. Instead of hardcoding the messages for the exceptions, you may be better served to get the message from a resource bundle. Obviously, exceptions that are thrown from third-party packages are not within your control, just as stack traces are hard to localize. Many organizations don't worry about localizing the exception messages and that may be

fine as long as no one from locales other than your own will ever need to use the information.

A good approach to localizing the exception message is to override the `getLocalizedMessage()` method defined in `java.lang.Throwable`. You can use this method to produce a local-sensitive message that can be shown to the user or inserted into a log file.

Conclusion

The new declarative exception handling is a great new addition to the Struts framework and one that should most certainly save developers time, during initial development and maintenance. Whenever possible, you should make an all out effort to take advantage of the declarative exception features, rather than attempting to write your own. The good news is however, that if you do need to create your own customized exception handling, the freedom and flexibility exists in the framework.

11

Using the Struts Validator

The Struts framework allows input validation to occur inside the `ActionForm`. To perform validation on data passed to a Struts application, developers must code special validation logic inside each `ActionForm` class. Although this approach works, it has some serious limitations. This chapter introduces the Validator framework by David Winterfeldt, which was created specifically to work with the Struts components and to help to overcome some of these limitations.

The Validator provides the ability to declaratively configure validation routines for a Struts application, without requiring you to program special validation logic. The Validator has become so popular and widely used by Struts developers, that it has been added to list of Jakarta projects, as well as the main Struts distribution.

The Need for a Validation Framework

In Chapter 7, you learned how to provide validation logic inside the `ActionForm` class. For each property that you need to validate, a separate piece of validation logic needs to be written to validate it. If an error is detected, you have to manually create an `ActionError` object and add it to the `ActionErrors` collection. Although this solution works, there are a few problems with the approach.

The first problem is that coding validation logic within each `ActionForm` causes redundant validation logic throughout your application. Within a single web application, the type of validation that needs to occur across HTML forms is very similar. The need to validate required fields, dates, times, and numbers for example, typically occurs in many places throughout an application. Most non-trivial applications have multiple HTML

forms that accept user input, which must be validated. Even if you use a single `ActionForm` for your entire application, you might still wind up replicating the validation logic.

The second major problem with this approach is one of maintenance. If you need to modify or enhance the validation that occurs for an `ActionForm`, recompilation of the source code is necessary. This makes it very difficult to customize or configure an application for specific needs.

The Validator framework allows you to move all the validation logic completely outside of the `ActionForm` and declaratively configure it for an application through external XML files. No validation logic is necessary in the `ActionForms`, and this makes your application easier to develop and maintain. The other great benefit of the Validator is that it's very extensible. It provides many standard validation routines out-of-the-box, but if you require additional validation rules, the framework is easy to extend and provides the ability to plug in your own rules; again without needing to modify your application.

Installing and Configuring the Validator

The Validator framework is now a part of the Jakarta Commons project. It's included with the Struts main distribution, but you can also get the latest version from the Commons download page at <http://jakarta.apache.org/commons>. Unless you need the source code or the absolute latest version, the necessary files are included with the Struts 1.1 distribution.

Required Packages

The Validator depends on several other packages to function properly, the most important of which is the Jakarta ORO package. The ORO package contains functionality for regular expressions, performing substitutions, and text splitting, among other utilities. The libraries were originally developed by ORO, Inc. and donated to the Apache Software Foundation. Earlier versions of the Validator framework depended on a different regular expression package called *Regexp*, which is also a Jakarta Project. However, ORO was considered the more complete of the two and the Validator that is included with Struts 1.1, now depends on the ORO package.

Other projects that are required are Commons BeansUtils, Commons Logging, Commons Collections, and Digester. All of the dependent packages for the Validator are included in the Struts 1.1 download. The `commons-validator.jar` and `jakarta-oro.jar` files need to be placed into the `WEB-INF/lib` directory for your web application. The other dependent JAR files must also be present, but should already be there due to Struts framework requirements.

Configuring the Validation Rules

As was mentioned earlier in the chapter, the Validator framework allows the validation rules for an application to be declaratively configured. This means that they are specified external to the application source. There are two important configuration files for the Validator framework.

The `validator-rules.xml` File

The first configuration file is called *validation-rules.xml*. This file contains a global set of validation rules that can be used out-of-the-box by your application. This file is application-neutral and can be used by any Struts application. You should only need to modify this file if you plan to modify or extend the default set of rules.

If you do need to extend the default rules, you might be better served to put your custom rules in a different XML file, so as to keep them separate from the default ones. This will help when it comes time to upgrade to a newer version of the Validator framework.

Each `validator` element describes one unique validation rule. The following fragment from the *validation-rules.xml* file is the definition for the required validation rule for example:

```
<validator
  name="required"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateRequired"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.required">
</validator>
```

The `validator` element also allows a `javascript` sub-element, but it is not shown here for brevity. The JavaScript support in the Validator framework will be discussed later in the chapter.

The `name` attribute assigns a logical name to the validation rule. It is used to reference the rule from the other rules within this file and also the application-specific validation file that is discussed in the next section. The names must be unique.

The `classname` and `method` attributes define the class and method that contains the logic for the validation rule. For example, in the code fragment for the `required` Validator rule, the `validateRequired()` method in the `StrutsValidator` class

will be invoked for the `required` validation rule. The `methodParams` attribute is a comma-delimited list of parameters for the method defined in the `method` attribute.

The `msg` attribute is a key from the resource bundle. The Validator framework will use this value to lookup a message from the Struts resource bundle when a validation error occurs. By default, the Validator framework uses the following values:

```
errors.required={0} is required.
errors.minlength={0} can not be less than {1} characters.
errors.maxlength={0} can not be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be an byte.
errors.short={0} must be an short.
errors.integer={0} must be an integer.
errors.long={0} must be an long.
errors.float={0} must be an float.
errors.double={0} must be an double.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid e-mail address
```

You should add these to your application's resource bundle, or change the key values in the `validation-rules.xml` file if you plan to use alternate messages.

The `validator` element also supports an attribute called `depends`. It is used to specify other validation rules that should be called before the rule specifying it. The is illustrated in the `minLength` validation rule:

```
<validator
  name="minlength"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateMinLength"
  methodParams=" java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest "
  depends="required"
  msg="errors.minlength">
</validator>
```

Before the `minLength` validation rule is called, the required rule will be invoked, as dictated by the `depends` attribute. You can also setup a rule to depend on multiple rules by separating the rules in the depends attribute with a comma:

```
| depends="required,integer"
```

If a rule that is specified in the `depends` attribute fails validation, the next rule will not be called. For example, in the `minLength` validation rule shown previously, the `validateMinLength` method will not be invoked if the `required` validation rule fails. This should stand to reason because there's no sense in checking the length of a value, if there's no value present.

The Validator framework is fairly generic. It contains very basic, atomic rules that can be used by an application. As you'll see later in this chapter, it's this generic ability that allows it to be used with non-Struts applications as well. The `org.apache.commons.validator.GenericValidator` class implements the generic rules as a set of public static methods. Table 11-1 lists the set of validation rules available in the `GenericValidator`.

Table 11-1. Validation Rules in the `GenericValidator` Class

Method Name	Description
<code>isBlankOrNull</code>	Checks if the field isn't null and length of the field is greater than zero not including whitespace.
<code>isByte</code>	Checks if the value can safely be converted to a byte primitive.
<code>isCreditCard</code>	Checks if the field is a valid credit card number.
<code>isDate</code>	Checks if the field is a valid date.
<code>isDouble</code>	Checks if the value can safely be converted to a double primitive.
<code>isEmail</code>	Checks if a field is a valid e-mail address.
<code>isFloat</code>	Checks if the value can safely be converted to a float primitive.
<code>isInRange</code>	Checks if a value is within a min and max range.
<code>isInt</code>	Checks if the value can safely be converted to a int primitive.
<code>isLong</code>	Checks if the value can safely be converted to a long primitive.
<code>isShort</code>	Checks if the value can safely be converted to a short primitive.
<code>matchRegexp</code>	Checks if the value matches the regular expression.
<code>maxLength</code>	Checks if the value's length is less than or equal to the max.
<code>minLength</code>	Checks if the value's length is greater than or equal to the min.

Because the validation rules in the `GenericValidator` are so fine-grained, the Struts developers added a utility class to the Struts framework called `org.apache.struts.util.StrutsValidator`, which defines a set of higher-level methods that are coupled to the Struts framework, but make its easier to use the Validator when using it with Struts. They are listed here without descriptions because the names are similar enough to the ones from Table 11-1.

- `validateByte`
- `validateCreditCard`

- `validateDate`
- `validateDouble`
- `validateEmail`
- `validateFloat`
- `validateInteger`
- `validateLong`
- `validateMask`
- `validateMinLength`
- `validateMaxLength`
- `validateRange`
- `validateRequired`
- `validateShort`

The `StrutsValidator` contains the concrete validation logic that's used by Struts. It's this class and the methods listed above that are declaratively configured in the `validation-rules.xml` file. When one of these methods is invoked and the validation fails, an `ActionError` is created and added to the `ActionErrors` object. These errors are eventually stored into the request and made available to the view components.

The `validation.xml` File

The second configuration file that is required by the Validator framework is the `validation.xml` file. This file is application specific, and describes which validation rules from the `validation-rules.xml` file are used by a particular `ActionForm`. This is what is meant by declaratively configured; you don't have to put code inside of the `ActionForm` class. The validation logic is associated to one or more `ActionForm` classes through this external file.

There are currently no official DTDs for the two Validator configuration files. There are several that have been created by the Struts user community, but they have not been verified or added to the Validator framework. Hopefully by the time this book is published the Validator developers will add a DTD for the configuration files.

The outermost element is the `form-validation` element. It can contain two child elements, `global` and `formset`. The `global` element can be repeated zero or more times, while the `formset` element can be present one or more times.

The `global` element allows you to configure `constant` elements that can be used throughout the rest of the file. This is analogous to how you might define a constant in a Java file and then use it throughout the class. The following fragment shows a `global` fragment that defines two constants:

```
<global>
  <constant>
    <constant-name>phone</constant-name>
    <constant-value>^\(?\d{3})\)?[-| ]?\d{3})[-| ]?\d{4})$</constant-value>
  </constant>
  <constant>
    <constant-name>zip</constant-name>
    <constant-value>^\d{5}\d*$</constant-value>
  </constant>
</global>
```

This fragment includes two constants, `phone` and `zip`, although you can include as many as you need. These constants are available to the elements within the `formset` section. They can be reused many times within the `formset`, referring to them by name. This is best illustrated with a small example. Example 11-1 shows a simple `validation.xml` file.

Example 11-1. A Simple `validation.xml` File

```
<form-validation>
  <global>
    <constant>
      <constant-name>phone</constant-name>
      <constant-value>^\(?\d{3})\)?[-| ]?\d{3})[-| ]?\d{4})$</constant-value>
    </constant>
  </global>
  <formset>
    <form name="checkoutForm">
      <field
        property="postalCode"
        depends="required,mask">
        <arg0 key="registrationForm.firstname.displayname"/>
        <arg1 name="minlength" key="{var:minlength}" resource="false"/>
        <var>
          <var-name>mask</var-name>
          <var-value>${zip}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

Notice in Example 11-1, the constant `phone` that's declared in the `global` section, is used in the `var` element for the `postalCode` validation rule.

The `formset` element can contain two child elements, `constant` and `form`. The `constant` element is the same format as the one in the global section. It can also be

repeated here zero or more times. The `form` element can be repeated one or more times within the `formset` element.

The `formset` element supports two attributes that deal with I18N, `language` and `country`. If you don't have any internationalization requirements for your validation routines and want to use the default locale, then you can leave these attributes off. The section *Internationalizing the Validation* later in the chapter discusses this topic in more detail.

The `form` element corresponds to a particular `ActionForm` definition that you want to provide validation for. It contains a single attribute `name`, which should match one of the name attributes from the `form-beans` section of the Struts configuration file. The `form` element can contain one or more `field` elements. The `field` element corresponds to a specific property in the `ActionForm`. In example 11-1, the sole `field` element for the checkout form corresponds to the `postalCode` property in an `ActionForm` called `checkoutForm` in the `form-beans` section of the Struts configuration file. The `field` element supports several attributes. They are listed in Table 11-2.

Table 11-2. The Attributes of the Validator `field` element.

Attribute	Description
<code>property</code>	The property name in the <code>ActionForm</code> subclass.
<code>depends</code>	Specifies one or more validation rules that are executed for the property.
<code>page</code>	Used to specify a page number in a multi-page wizard-like <code>ActionForm</code> .
<code>indexedListProperty</code>	Specifies a property name in the <code>ActionForm</code> that returns a Collection that can be indexed.

The `msg` element allows you specify an alternate message for a field element. The validation rule can use this value instead of the default message declared with the rule. The value for the `msg` element must be a key from the application resource bundle.

The field element allows up to four additional elements to be included. The elements, named `arg0`, `arg1`, `arg2`, and `arg3`, are used to pass addition values to the message, either from the resource bundle, `var`, or `constant` elements. Example 11-1 included elements for `arg0` and `arg1`.

```
<arg0 key="registrationForm.firstname.displayname"/>
<arg1 name="minlength" key="{var:minlength}" resource="false"/>
```

The `arg0` value uses a key attribute to specify a value from the resource bundle. The `arg1` element on the other hand, is passing the value specified in the `var` element, to the `minLength` validation rule. The `resource` attribute is set to false to indicate that the framework should not attempt to retrieve this value from the resource bundle.

The `var` element is used to pass information to a validation method. In Example 11-1, the substituted value for the `zip` constant is passed into the mask validation rule so that it can be used to check whether or not the property value conforms to the proper zip mask. The `var` element can be repeated zero or more times for a `field` element.

Once you have the two XML resource files configured for your application, you need to place them in the `WEB-INF` directory. They will be referenced within the Struts configuration file, as described in the next section.

Plugging in the Validator

Each Struts application needs to know that the Validator framework is being employed. Using a technique that you learned about in Chapter 9, you can use the `PlugIn` mechanism to hook the Validator framework into a Struts application.

Earlier versions of the Validator used an extra servlet to inform the Struts application that the Validator components were present. The `ValidatorServlet` has been deprecated and should not be used.

The following fragment illustrates how to setup the Validator as a `PlugIn`:

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

There was some confusion in one of the earlier beta releases for the Validator that used multiple `set-property` elements. That is no longer supported and you should use a single `set-property` element that specifies multiple Validator resource files, separated by a comma. Also notice that the property value is the plural `pathnames`.

The Struts framework will call the `init()` method in the `ValidatorPlugIn` class when the application starts up. During this method, the Validator resources from the XML files are loaded into memory, so that they will be available to the application. Before calling the `init()` method however, the `pathnames` property value will be passed to the `ValidatorPlugIn` instance. This is how the `ValidatorPlugIn` finds out which Validator resources to load. For more information on how the `PlugIn` mechanism works, see *Using the Plugin Mechanism* in Chapter 9.

Overview of Regular Expressions

It would obviously be a waste of time to cover everything that you need to know about regular expressions in this book. On the other hand, since the Validator framework relies heavily on regular expressions, it would be beneficial to go over some basics.

[\[Note to Reviewers: I was planning on including a very brief introduction to regular expression here. I wanted your opinion before including it. Chuck\]](#)

Using an `ActionForm` with the Validator

You can't use the standard Struts `ActionForm` class with the Validator. Instead, you will need to use a subclass of the `ActionForm` class that is specifically designed to work with the Validator framework. There are two root subclasses to select from depending on whether or not you are planning to use dynamic `ActionForms`. Figure 11-1 shows the `ActionForm` and its descendants to help you visualize the hierarchy better.

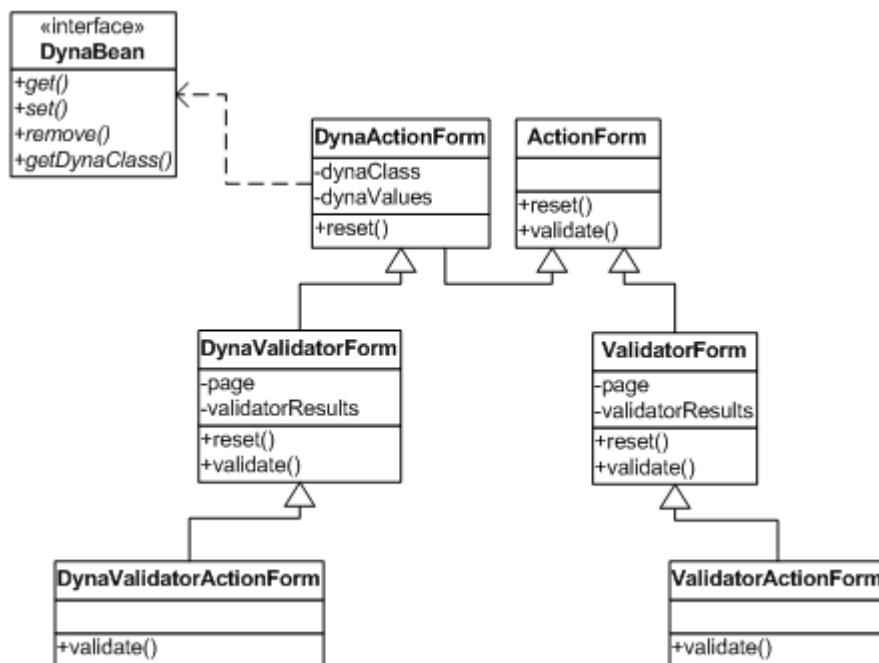


Figure 11-1. The `ActionForm` Class Hierarchy

If you are using dynamic `ActionForms`, then you should use the `DynaValidatorForm` branch of the hierarchy. If you are using standard

`ActionForms`, then you can use the `ValidatorForm` or one of its descendants instead.

Whether you use dynamic `ActionForms` or regular `ActionForms`, the manner in which you configure the Validator is the same. Just be sure that whichever `ActionForm` subclass you choose, you configure the `form-bean` section of the Struts configuration file using the fully qualified class name. See *The form-beans Element* in Chapter 4 for more details.

Dynamic or non-dynamic is only the first decision that you have to make. There is another decision that you must make regarding choosing the proper `ActionForm` subclass. Notice that in both the dynamic and non-dynamic branch of the `ActionForm` hierarchy in Figure 11-1, there are two versions of `ValidatorForm` to choose from. The parent class is just called `ValidatorForm`, or `DynaValidatorForm` for the dynamic branch.

Each one of these has a subclass that contains the name `Action` in its title. The subclass of the `ValidatorForm` is called `ValidatorActionForm` and the one for the dynamic version is called `DynaValidatorActionForm`. The reason for the two different versions is whether you want to associate the validation with the form-bean definition or the action definition. The versions of the classes in Figure 11-1 that contain the word `Action` in their title pass the `path` attribute from the action element into the Validator. Therefore, the Validator will use the action's name to look up the validation rules. If you use the `ValidatorForm` or `DynaValidatorForm`, the name of the `ActionForm` is used to lookup the set of validation rules to use. The only reason for using one or the other is have finer-grained control over which validation rules are executed. For example, supposed that an `ActionForm` contain three different validation rules, but only two of them should get executed for a particular action. You could configure the rules to only perform the subset of validation rules when that action gets invoked. Otherwise, all of the rules would be invoked. In general, `ValidatorForm` or `DynaValidatorForm` will be sufficient for your needs.

It would be helpful to provide a more complete example of using the Validator framework. As with previous chapters, we'll employ the Storefront application to help us understand the Validator better. In particular, we'll look at the HTML form used to capture the shipping information during checkout of the Storefront application. This is shown in Figure 11-2.

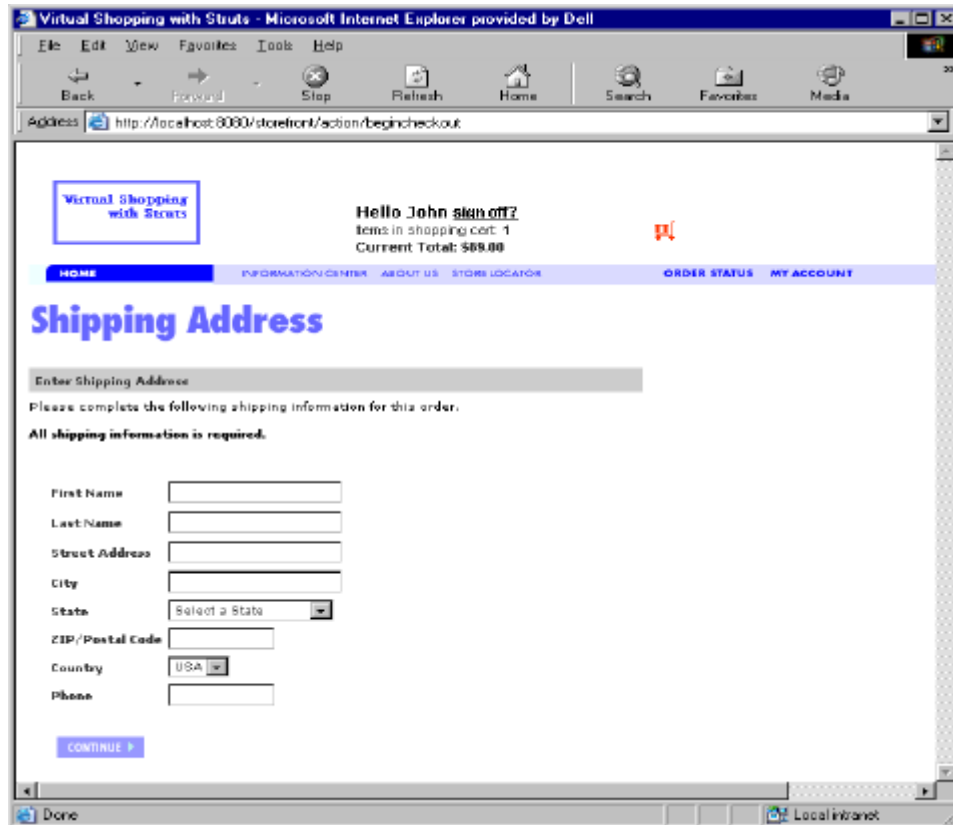


Figure 11-2. Capturing the Shipping Address Information

For this example, we are going to use a dynamic form. Therefore we are going to use the `DynaValidatorForm` class to capture the shipping address details. Because the checkout process will span multiple pages, and we want to capture this information across pages, we will need to configure the form bean to be session scope. We are also going to capture all of the checkout properties in a single `ActionForm` class. Instead of having a `ShippingForm` and a `CreditCardForm`, we will have a single form that captures all of it called `CheckoutForm`.

In our Struts configuration file, we setup the `checkoutForm` as shown here:

```
<form-bean
  name="checkoutForm"
  dynamic="true"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="firstName" type="java.lang.String"/>
  <form-property name="lastName" type="java.lang.String"/>
  <form-property name="address" type="java.lang.String"/>
  <form-property name="city" type="java.lang.String"/>
  <form-property name="state" type="java.lang.String"/>
  <form-property name="postalCode" type="java.lang.String"/>
```

```
<form-property name="country" type="java.lang.String"/>
<form-property name="phone" type="java.lang.String"/>
</form-bean>
```

Notice that the `type` attribute specifies the exact `ActionForm` subclass that we chose and have also set the `dynamic` attribute to true. These are two common setup mistakes that newcomers to the framework make.

The next step is to edit the application-specific validation logic, which is done in the `validation.xml` file. You must declare a validation rule for each property in the form that you need to validate. In some cases, you might need to specify multiple ones. In Figure 11-2 for example, the phone field is required and it must fit a specific format. These are two separate rules that must evaluate to true or the validation for the form fails. The entire `validation.xml` file will not be shown, because it's too large and most of it is redundant. A small section will be shown that will help you understand how things are connected. This is shown in Example 11-2.

Example 11-2. A Sample `validation.xml` for the Checkout Form

```
<formset>
  <constant>
    <constant-name>phone</constant-name>
    <constant-value>^\d{3}\d{3}[-| ]?\d{3}[-| ]?\d{4}$</constant-value>
  </constant>
  <constant>
    <constant-name>zip</constant-name>
    <constant-value>^\d{5}(-\d{4})?$</constant-value>
  </constant>
  <form name="checkoutForm">
    <field
      property="firstName"
      depends="required,mask">
      <arg0 key="label.firstName"/>
      <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z]*$</var-value>
      </var>
    </field>
    <field
      property="postalCode"
      depends="required,mask">
      <arg0 key="registrationForm.firstname.displayname"/>
      <arg1 name="minlength" key="{var:minlength}" resource="false"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${zip}</var-value>
      </var>
    </field>
    <field
      property="phone"
      depends="required,mask">
      <arg0 key="registrationForm.firstname.displayname"/>
      <arg1 name="minlength" key="{var:minlength}" resource="false"/>
```

```
<var>
  <var-name>mask</var-name>
  <var-value>${phone}</var-value>
</var>
</field>
</form>
</formset>
</form-validation>
```

Using the Validator Framework

Not that we have everything configure for the Storefront example, it's time to run the example. The nice thing about using a declarative approach versus a programmatic one is that, once you have everything configured, you're ready to go. The absence of programming makes the declarative approach much simpler. This is especially true for the Validator framework. There's nothing to code, as long as the default validation rules satisfy your requirements.

When we submit the shipping address page with no information in the fields, the validation rules kick in and the result is shown in Figure 11-3.

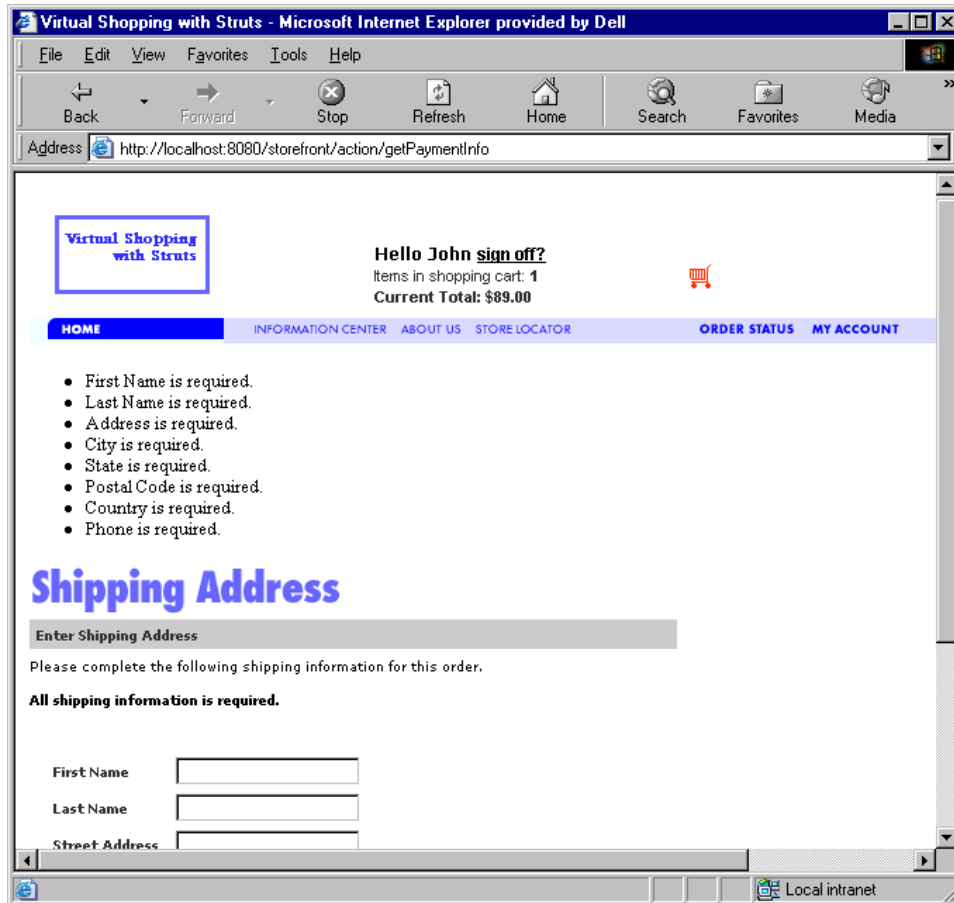


Figure 11-3. The Shipping Address Page using the Validator Framework

Creating your own Validation Rules

The Validator framework is pre-configured with many of the most common rules that you'll likely need for your Struts applications. If your application has validation requirements that are not met by the default ones, you have complete freedom to create your own. There are several steps that you must follow however, to create your own customized rules.

1. Create a Java class that contains the validation methods.
2. Edit the *validation-rules.xml* file or create your own version. If you do create a new validation resource file, be sure to add it to the list of resource files in the Validator PlugIn.
3. Use the new validation rules in the *validation.xml* file for your application.

For each validation method you create, it needs to have the following signature:

```
public static boolean validateXXX( java.lang.Object,  
                                org.apache.commons.validator.ValidatorAction,  
                                org.apache.commons.validator.Field,  
                                org.apache.struts.action.ActionErrors,  
                                javax.servlet.http.HttpServletRequest,  
                                javax.servlet.ServletContext );
```

where **validateXXX** can be whatever you want it to be, as long as it's not a duplicate rule name. Table 11-3 explains what the arguments to the **validateXXX()** method are used for.

*Table 11-3. The **validateXXX()** Method Arguments*

Parameter	Description
Object	The JavaBean that validation is being performed on
ValidatorAction	The current ValidatorAction being performed
Field	The Field object being validated
ActionErrors	The errors objects to add an ActionError to if the validation fails
HttpServletRequest	The current request object
ServletContext	The application's ServletContext

In most cases, the method should be static. However, you can define instance-level methods, but you must ensure that the methods are thread-safe. Example 11-3 illustrates a new validation rule that determines whether a String value is a valid boolean.

Example 11-3. A Validation Rule that Validates a Boolean Value

```
import java.io.Serializable;  
import java.util.Locale;  
import javax.servlet.ServletContext;  
import javax.servlet.http.HttpServletRequest;  
import org.apache.commons.validator.Field;  
import org.apache.commons.validator.GenericTypeValidator;  
import org.apache.commons.validator.GenericValidator;  
import org.apache.commons.validator.ValidatorAction;  
import org.apache.commons.validator.ValidatorUtil;  
import org.apache.struts.action.ActionErrors;  
import org.apache.struts.util.StrutsValidatorUtil;  
  
public class NewValidator implements Serializable {  
    public static boolean validateBoolean( Object bean,  
                                         ValidatorAction va, Field field,  
                                         ActionErrors errors,  
                                         HttpServletRequest request) {  
  
        String value = null;  
        // The boolean is stored as a String value  
        if (field.getProperty() != null && field.getProperty().length() > 0){
```

```
        value = ValidatorUtil.getValueAsString(bean, field.getProperty());
    }
    Boolean result = null;
    try {
        result = Boolean.valueOf(value);
    } catch (Exception e){
        // Do Nothing on Purpose
    }
    // Create an ActionError for the failed validation
    if (result == null) {
        errors.add( field.getKey(),
                    StrutsValidatorUtil.getActionError(request, va, field));
    }
    // Returns true if the value was successful converted, false otherwise
    return (result != null);
    }
}
```

Step 2 is to add this new rule to the *validation-rules.xml* file, or to a new file just to keep your customized rules separate. The *validator* element for the *validateBoolean* rule should look something like:

```
<validator name="boolean"
  classname="NewValidator"
  method="validateBoolean"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.boolean">
```

The final step is to utilize the new validation rule in the *validation.xml* file. This involves creating a *field* element that matches to a boolean property on an *ActionForm*:

```
<field
  property="sendEmailConfirmation"
  depends="required">
  <arg0 key="label.emailconfirmation"/>
</field>
```

The Validator and JSP Custom Tags

There are several JSP custom tags that are included within the Struts tag libraries that can be used with the Validator framework. One of the tags is used to generate dynamic JavaScript based on the validation rules. The other tags are part of the core Struts framework and are used both with and without the Validator.

The tags listed in Table 11-4 are generic and can be used with or without the Validator framework, but come in handy when using the Validator framework.

Table 11-4. JSP Custom Tags that can be used with the Validator

Tag Name	Description
Errors Tag	Displays any validation errors found during processing
ErrorsExist Tag	Determines if there were any validation errors
Messages Tag	Displays any messages found during processing
MessagesExist Tag	Determines if there were any messages during processing

The tags in Table 11-4 allow JSP pages to detect and obtain access to messages or errors that were detected in the Struts application. These tags were discussed in more detail in Chapter 8.

Using JavaScript with the Validator

The Validator framework is also capable of generating JavaScript for your Struts application using the same framework as it does for server-side validation. This is accomplished by using a set of JSP custom tags that are designed specifically for this purpose.

Configuring the `validation-rules.xml` File for JavaScript

The Validator custom tag called `JavascriptValidator` is used to generate client-side validation based on a `javascript` attribute being present within the `validator` element. Before the JSP custom tag can be used, there must be a `javascript` element for the validation rule. The following code fragment illustrates the `required` validation rule that includes a `javascript` element:

```
<validator
  name="required"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateRequired"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.required">
  <javascript><![CDATA[
    function validateRequired(form) {
      var bValid = true;
      var focusField = null;
      var i = 0;
      var fields = new Array();
      oRequired = new required();

      for (x in oRequired) {
        if ((form[oRequired[x][0]].type == 'text' ||
          form[oRequired[x][0]].type == 'textarea' ||
          form[oRequired[x][0]].type == 'select-one' ||
```

```
        form[oRequired[x][0]].type == 'radio' ||
        form[oRequired[x][0]].type == 'password') &&
        form[oRequired[x][0]].value == '') {
            if (i == 0)
                focusField = form[oRequired[x][0]];
            fields[i++] = oRequired[x][1];
            bValid = false;
        }
    }

    if (fields.length > 0) {
        focusField.focus();
        alert(fields.join('\n'));
    }
    return bValid;
}]]>
</javascript>
</validator>
```

When the `JavaScriptValidator` tag is included in the JSP page, the text from the `javascript` element is written to the JSP page to provide client-side validation. When the user submits the form, the client-side validation is executed and any validation rules that fail present messages to the user.

You will need to include the `javascript` tag with the name of the `ActionForm` that it's going to validate against:

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

The `formName` attribute is used to lookup the set of validation rules to include as JavaScript in the page. You will manually have to add an `onsubmit` event handler for the form:

```
<html:form action="getPaymentInfo" onsubmit="return validateCheckoutForm(this);">
```

When the form is submitted, the `validateCheckoutForm()` JavaScript function will be invoked. The validation rules will be executed and if one or more rules fail, the form will not be submitted. The `javascript` tag generates a function that has a name of `validateXXX()`, where `XXX` is the name of the `ActionForm`. So if your `ActionForm` is called `checkoutForm`, then the `javascript` tag will create a JavaScript function called `validateCheckoutForm()` that executes the validation logic. This is why the `onsubmit()` event handler was calling the `validateCheckoutForm()` function.

By default, the `JavaScriptValidator` tag generates both static and dynamic JavaScript functions. If you would like to include a separate file that contains static JavaScript functions to take advantage of browser caching or to better organize your application, you can utilize the `dynamicJavaScript` and `staticJavaScript` attributes. By default, both of these are set to true. You can set the `staticJavaScript` attribute to false in your form and include a

separate JavaScript page with the `dynamicJavascript` set to false and the `staticJavascript` attribute to true. See the documentation for the `JavascriptValidator` tag for more information.

Internationalizing the Validation

The Validator framework utilizes the application resource bundles to generate the error messages, both for client-side and server-side validation. So from an I18N perspective, much of the work to display language-specific messages to the user is included within the frameworks.

It was mentioned earlier that the `formset` element in the `validation.xml` file supports attributes related to internationalization. Those attributes are `language`, `country`, and `variant`. As you know, these attributes correspond to the `java.util.Locale` class. If you don't specify these attributes, the default `Locale` is used.

If your application has I18N validation requirements, you will need to create separate `formset` elements, one for each `Locale` that you need to support, and for each form that you need to validate. For example, if your application had the requirements to support validation for a form called `registrationForm` for both the default locale and the French locale, the `validation.xml` file would contain two `formset` elements; one for the default locale and the other for the French locale. This is shown in the following example fragment:

```
<formset>
  <form name="registrationForm">
    <field
      property="firstName"
      depends="required,mask,minlength">
      <arg0 key="registrationForm.firstname.displayname"/>
      <arg1 name="minlength" key="{var:minlength}" resource="false"/>
      <var>
        <var-name>mask</var-name>
        <var-value>^\w+$</var-value>
      </var>
      <var>
        <var-name>minlength</var-name>
        <var-value>5</var-value>
      </var>
    </field>
  </form>
</formset>
<formset language="fr">
  <form name="registrationForm">
    <field
      property="lastName"
      depends="required,mask">
      <msg name="mask" key="registrationForm.lastname.maskmsg"/>
    </field>
  </form>
</formset>
```

```
<arg0 key="registrationForm.lastname.displayname"/>
<var>
  <var-name>mask</var-name>
  <var-value>^\w+$</var-value>
</var>
</field>
</form>
</formset>
```

[\[Note to Reviewers: I need your opinion on whether the following section adds any value to this chapter. Obviously this book is about the Struts framework, so I'm not sure how beneficial this next section is. My thinking is that everyone building a Struts application has some other business layer that they have to deal with; whether its EJB or just a bunch of Java Objects and JDBC. All business layers that I've written need validation logic, so this is why I mentioned this material. After reading it, any thoughts would be appreciated. Chuck\]](#)

Using the Validator Outside of Struts

Although the Validator was originally designed to work with the Struts framework, it can be used to perform generic validation on any JavaBean. There are several steps that must be performed before the framework can be used outside of Struts. Although the Validator is not dependent on the Struts framework, there has been a considerable amount of work done inside of Struts to make it easier to utilize the Validator. This behavior will need to be replicated for your application, if you plan to use the Validator without Struts.

The package dependencies are exactly the same for Struts and non-Struts applications alike. The ORO, Commons Logging, Commons BeanUtils, Commons Collections, and Digester packages are all required. You will also need an XML parser that conforms to the SAX 2.0 specification. You will not however, need to include the Struts framework.

The first behavior to replicate is the functions for loading and initializing the XML Validator resources. These are the two XML files that are used to configure the rules for the Validator. When the Validator framework is used in conjunction with Struts, the `org.apache.struts.validator.ValidatorPlugIn` class performs this duty. However, since the `ValidatorPlugIn` is dependent on Struts, you will need to create an alternate approach for initializing the appropriate Validator resources. To do this, you can create a simple Java class that performs the same behavior as the `ValidatorPlugIn`, but doesn't have a dependency on the Struts framework. A simple example is provided in Example 11-4.

Example 11-4. A Class that Loads and Initializes the Validator Resources

```
import java.util.*;
import java.io.*;
```

```
import org.apache.commons.validator.ValidatorResources;
import org.apache.commons.validator.ValidatorResourcesInitializer;

public class ValidatorLoader{

    private final static String RESOURCE_DELIM = ",";
    protected ValidatorResources resources = null;
    private String pathnames = null;

    public ValidatorLoader() throws IOException {
        loadPathnames();
        initResources();
    }

    public ValidatorResources getResources(){
        return resources;
    }

    public String getPathnames() {
        return pathnames;
    }

    public void setPathnames(String pathnames) {
        this.pathnames = pathnames;
    }

    protected void loadPathnames(){
        // Set a default just in case
        String paths = "validation-rules.xml,validation.xml";
        InputStream stream = null;

        try{
            // Load some properties file
            stream = this.getClass().getResourceAsStream( "validator.properties" );
            if ( stream != null ){
                Properties props = new Properties();
                props.load( stream );
                // Get the pathnames string from the properties file
                paths = props.getProperty( "validator-pathnames" );
            }
        }catch( IOException ex ){
            ex.printStackTrace();
        }
        setPathnames( paths );
    }

    protected void initResources() throws IOException {
        resources = new ValidatorResources();

        if (getPathnames() != null && getPathnames().length() > 0) {
            StringTokenizer st = new StringTokenizer(getPathnames(), RESOURCE_DELIM);
            while (st.hasMoreTokens()) {
                String validatorRules = st.nextToken();
                validatorRules = validatorRules.trim();
            }
        }
    }
}
```

```
InputStream input = null;
BufferedInputStream bis = null;
input = getClass().getResourceAsStream("validatorRules");

if (input != null){
    bis = new BufferedInputStream(input);

    try {
        // pass in false so resources aren't processed
        // until last file is loaded
        ValidatorResourcesInitializer.initialize(resources, bis, false);
    } catch (Exception ex){
        ex.printStackTrace();
    }
}

// process resources
resources.process();
}
```

The work begin done in the `ValidatorLoader` from Example 11-4 is very similar to what the `ValidatorPlugIn` does. That is, to load and initialize an instance of the `ValidatorResources` class. The object is an in-memory representation of the validation rules for an application. This example uses the `getResourceAsStream()` method to find and load a properties file that contains the list of Validator resource files.

Once you create and initialize an instance of the `ValidatorResources` class, you will need to cache this somewhere. In a Struts application, this is cached in the `ServletContext`. Your application can hang onto this object, or you could even wrap the resource inside of a Singleton.

Modifying the validation-rules.xml File

In the section *Creating your own Validation Rules* of this chapter, you saw how to extend the Validator framework with your own customized rules. You're going to have to do this here as well, except that the method signatures will be different. The method signature in Example 11-3 included parameters that are part of the Servlet and Struts APIs.

[\[Reviewers: I got stuck right here and instead of holding onto this chapter, I thought it best to post it as is. This section will be completed soon and reposted. Chuck\]](#)

12

Internationalization and Struts

Companies can no longer afford to think only about local market places. Since the mid to late 1990s, the business world has been overrun with ideas about a world economy. All you have to do is to look at what's happening in Europe with the Euro. Businesses and even countries are realizing that they just can't think about their traditional markets and at the same time continue to grow revenue and be successful. They must start thinking globally and attempt to bring in these global customers for their products and services.

With the explosion of the World Wide Web (WWW) starting in the mid 1990s, companies that are conducting business on the Internet are finding out that access to their products and services via a web site is an ideal way of attracting new customers from all over the world. One of the key reasons is 24/7 access. Regardless of the time zone the business or the customers are in, the Internet, and more importantly the web, allows a customer to shop and purchase goods and services any time of the day or night. Traditional business hours are irrelevant when it comes to the web. What unlimited access can mean to companies and their revenue is enormous. However, for the software developers that have to build and maintain the applications to support global customers, the task can be daunting.

This chapter focuses on what it takes to make a Struts application available to customers from around the world, regardless of their language or geographical location. Like many other things in software development, planning ahead is the most important thing that you can do to help ensure success. After reading this chapter, you should be able to build Struts applications that make it possible to support a broad reach of customers.

What is Internationalization?

Traditionally, software developers have focused on building applications that solve an immediate business problem. While doing so, it's very easy and sometimes necessary to make assumptions about the user's language or country. In many of the cases, these assumptions are valid and there's never a question as to who the audience will be. However, if you have ever had to reengineer an application because these assumptions weren't correct, then you know how hard it can be to go back and correct the application design after the fact.

Internationalization (I18N), simply stated, is the process of designing your software to support multiple languages and regions ahead of time, so that you don't have to go back and reengineer your applications every time a new language or country needs to be supported. An application that is said to support Internationalization has the following characteristics:

- Additional languages can be supported without requiring code changes.
- Text elements, messages, and images are stored externally to the source code.
- Culturally dependent data such as date and times, decimal values, and currencies are formatted correctly for the user's language and geographic location.
- Has support for non-standard character sets.
- Can quickly be adapted to new language and/or region.

When you internationalize an application, you can't afford to pick and choose which options you want to support. You must implement all of them or the process breaks down. If a user visited your web site for example, and all of the text, images, and buttons were in the correct language, but the numbers and currency were not formatted correctly, this would make for an unpleasant experience for the user.

Ensuring that the application can support multiple languages and regions is only the first step. You still must create localized versions of the application for each specific language and/or regions that you wish to support. Fortunately, here's where the benefits of I18N on the Java platform pay off. For applications that have been properly internationalized, all of the work to support a new language or country is done external to the source code.

A locale is a region (usually geographic, but not necessarily so) that shares customs, culture, and language. Localization (L10N) is the process of adapting your application, which has been properly internationalized, to a specific locale. For applications where I18N support hadn't been planned or built in, this usually means changes to text, images, and messages that are embedded within the source code. After the changes are applied, the source code might need to be recompiled. Imagine doing this time and time again for each new locale that you had to support. Applications that are written for a single locale are commonly referred to as *myopic*.

According to Richard Gillam, from the *Unicode Technology Group*, whose group designed much of the Internationalization support in the Java libraries, “Internationalization is not a feature”. Users will expect that the products they use will work for them, in their native language. Things go wrong and users get unhappy when assumptions that you make are incorrect. Start planning early for I18N support in your applications. Even if it doesn’t look like your going to need it, you’ll be that much ahead and it will not hinder or slow down development as long as you do it right from the start.

Obviously not every application needs support for Internationalization. Some developers and development organizations frown on adding in functionality that isn’t part of the requirements. Even if your application truly has no requirements to support more than a single locale, there are still benefits that you can gain from including some aspects of I18N. For example, by using resource bundles for all of your static text, you can save development and more importantly maintenance time. We’ll see exactly how to accomplish this later in this chapter.

Support for I18N in Java

Java provides a rich set of I18N features in the core library. This section will briefly discuss a few of those core features. The I18N support in the Struts framework relies heavily on these components. Having an understanding of how the Java I18N components cooperate with each other will go a long way in understanding how to internationalize your Struts applications. The topic of Internationalization is too broad to cover in depth in this book. A good source on the topic is the book *Java Internationalization* by Deitsch and Czarnecki published by O’Reilly.

The Locale Class

The `java.util.Locale` class is undeniably the most important of all I18N classes in the Java library. Almost all of the support for internationalization and localization in or around the Java language relies on this class.

The `Locale` class provides Java instances of the locale concept mentioned earlier. A particular instance of the `Locale` represents a unique language and region. When a class in the Java library modifies its functionality during runtime based on a `Locale` object, it’s said to be locale-sensitive. For example, the `java.text.DateFormat` is locale-sensitive since it will format a date differently, depending on a particular `Locale` object.

The `Locale` objects don’t do any of the I18N formatting or parsing work. They are used as identifiers by the locale-sensitive classes. When you acquire an instance of the `DateFormat` class, you can pass in a `Locale` object for the United States (US). The

`DateFormat` class does all of the locale-sensitive parsing and formatting; it only relies on the `Locale` to identify the proper format.

You should be very careful when using the `java.text.Format` class or any of its descendants, including `DateFormat`, `NumberFormat`, and `SimpleDateFormat` because they are not thread-safe. The thread-safety problem exists because an instance of the `Calendar` class is stored as a member variable and accessed during the `parse()` and `format()` method invocations. You will need to use a separate instance for each thread or synchronize access externally. Don't store a single instance somewhere like application scope and allow multiple client threads to access it. You can however store instances in the user's session and use different instances for each user to help ensure thread-safety. The thread-safety problem includes all versions of Java, including 1.4. The API documentation for the `Format` classes has been updated to indicate the known design issue.

When you create a `Locale` object, you typically specify the language and country code. The following code fragment illustrates the creation of two locale objects, one for the US and the other for Great Britain.

```
Locale usLocale = new Locale("en", "US");
Locale gbLocale = new Locale("en", "GB");
```

The first argument in the constructor is the language code. The language code consists of two lower-case letters and must conform to the ISO-639 specification. You can see a complete listing of the available language codes at <http://www.unicode.org/unicode/onlinedat/languages.html>.

The second argument is the country code. It consists of two upper-case letters that must conform to the ISO-3166 specification. The available country codes are available from the same site at <http://www.unicode.org/unicode/onlinedat/countries.html>.

The `Locale` class provides several static convenience constants that allow you to acquire an instance of the most often used locales. For example, if you wanted to get an instance of a Japanese locale, the following two are equivalent:

```
Locale locale1 = Locale.JAPAN;
Locale locale2 = new Locale("ja", "JP");
```

The Default Locale

The JVM will query the operating system when first started, and set a default `Locale` for the environment. You can obtain the information for this default locale by calling the `getDefault()` method on the `Locale` class:

```
Locale defaultLocale = Locale.getDefault();
```

The web container will normally utilize the default locale for its local environment, while using the one passed from the client in the `HttpServletRequest` to display locale-sensitive information back to the end user.

Determining the User Locale

In the last section, you saw how to create `Locale` objects in Java by passing in the language and country code to the `Locale` constructor. Within web applications, including those built using the Struts framework, you rarely ever have to create your own locale instances because the container will do it for you. The `ServletRequest` interface contains two methods that can be called to retrieve the locale preferences of a client:

```
public java.util.Locale getLocale();
public java.util.Enumeration getLocales();
```

Both of these methods utilize the `Accept-Language` header that is part of each client request sent to the servlet container.

Since the web server doesn't keep a long-term open connection with a browser, the client locale preference is sent to the servlet container with each request. Although the user's locale information may be sent with each request, Struts by default, will only retrieve the information once and store it into the user's session. The `Locale`, if stored into the session, is stored with a key of `Action.LOCALE_KEY`, which translates to the string `org.apache.struts.action.LOCALE`.

You can configure whether Struts stores the user's locale into the session or not by setting the locale attribute in the `controller` element within the Struts application configuration file. If you don't provide a value for the `locale` attribute, it defaults to false. See chapter 4 for more information on configuring the locale.

The `getLocale()` method on the `HttpServletRequest` object returns the preferred locale of the client, while the `getLocales()` method returns an `Enumeration` of preferred locales in decreasing order of preference. If a client doesn't have a preferred locale configured, the servlet container will return its default locale. Example 12-1 illustrates how to determine this information using a servlet.

Example 12-1. Determining the User's Locale Information in a Servlet

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Locale;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;

/**
 * Prints out information about a user's preferred locales
 */
public class LocaleServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html";

    /**
     * Initialize the servlet
     */
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    /**
     * Process the HTTP Get request
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head><title>The Example Locale Servlet</title></head>");
        out.println("<body>");

        // Retrieve and print out the user's preferred locale
        Locale preferredLocale = request.getLocale();
        out.println("<p>The user's preferred Locale is " + preferredLocale + "</p>");

        // Retrieve all of the supported locales of the user
        out.println("<p>A list of preferred Locales in decreasing order</p>");
        Enumeration allUserSupportedLocales = request.getLocales();
        out.println("<ul>");
        while( allUserSupportedLocales.hasMoreElements() ){
            Locale supportedLocale = (Locale)allUserSupportedLocales.nextElement();
            out.println("<li>Locale: " + supportedLocale + "</li>");
        }
        out.println("</ul>");

        // Get the container's default locale
        Locale servletContainerLocale = Locale.getDefault();
        out.println("<p>The containers Locale " + servletContainerLocale + "</p>");
        out.println("</body></html>");
    }
}
```

When you execute the servlet from example 12-1, you should see output similar to the browser output in Figure 12-1.

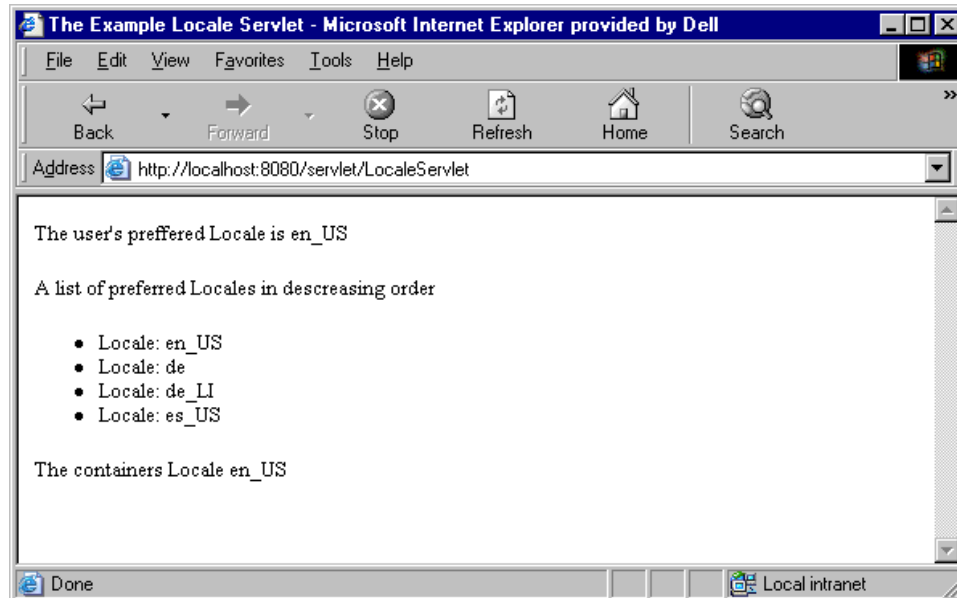


Figure 12-1. The browser output from Example 12-1

The output may be different if you have different locales configured for your system. Most Internet browsers allow you to configure the locales you prefer to support. With Microsoft Internet Explorer for example, you can edit the “languages” from the Tools-Internet Options pull down menu.

Getting the user’s locale within the Struts framework is very easy. There are, in fact, several ways of getting the stored `Locale` for the user, depending on where you are trying to access it from. If you are within an `Action` class for example, you can simply call the `getLocale()` method that’s defined in the Struts base `Action`. You will need to pass the request object to this method, since it will need to use the `HttpSession` to obtain the locale.

Obtaining the user’s locale from anywhere else is also straightforward because you can simply get it directly from the session using the `Action.LOCALE_KEY` mentioned earlier. Remember that the `getLocale()` method on the request object returns the client’s preferred locale, if one has been configured. However, the Struts framework might have already obtained it and placed it into the user’s session for you. You should use the `Locale` object configured in the user’s session, if present.

If your application allows a user to change locales on the fly, then you may have to call the `getLocale()` method on each new request to see if the user has changed locales. An example of doing this was shown in the `CustomRequestProcessor` class in Example 5-4 of Chapter 5.

Java Resource Bundles

The `java.util.ResourceBundle` class provides the ability to group a set of resources together for a given locale. The resources are usually textual elements like field and button labels and status messages, but can also be items such as image names, error messages, and page titles.

The Struts framework does not use the `ResourceBundle` provided by the core language. Instead, it provides similar functionality with the classes within its framework. The `org.apache.struts.util.MessageResources` class and its only concrete subclass, `org.apache.struts.util.PropertyMessageResources`, are used to perform parallel functionality to that of the `ResourceBundle` hierarchy. If you understand the fundamentals of the `ResourceBundle` in the code library, then you basically understand how the version with the Struts framework operates.

In retrospect, the `MessageResources` should have at least been a subclass of the `Java ResourceBundle`.

You'll see an example of creating a resource bundle for a Struts application later in the *"The Struts Resource Bundle"* section of this chapter.

The MessageFormat Class

The Java `ResourceBundle`, as well as the Struts `MessageResources` class, allows for both static and dynamic text. Static text is used for elements like field and button labels where the localized strings are used exactly as they are in the bundle. In other words, the text for the message is known ahead of time. With dynamic text, part of the message may not be known until runtime. To help make the difference clearer, let's look at an example.

Suppose you needed to display a message to the user informing them that the name and phone input fields were required to save a new user record. One approach would be to add entries to the resource bundle like this:

```
error.requiredfield.name=The Name field is required to save.  
error.requiredfield.phone=The Phone field is required to save.  
  
// other resource messages...
```

This approach works fine, but what if there were hundreds of required fields? The resource bundle would become very large and would be difficult to maintain. Notice how the only difference between the two messages is the name of the field that is required.

A much easier and more maintainable approach is to utilize the functionality of the `java.text.MessageFormat` class. This would allow you to do something like this.

```
|error.requiredfield=The {0} field is required to save.
```

```
label.phone=Phone  
label.name=Name
```

The values that are not known until runtime are substituted in the message by a set of braces and an integer value. The integer inside the braces is used as an index into an `Object[]` that is passed in with the `format()` message of the `MessageFormat` class. Example 12-2 provides an example of this.

Example 12-2. Using the MessageFormat Class to Format Messages with Variable Text

```
import java.util.ResourceBundle;  
import java.util.Locale;  
import java.text.MessageFormat;  
  
public class FormatExample {  
  
    public static void main(String[] args) {  
        // Load the resource bundle  
        ResourceBundle bundle = ResourceBundle.getBundle( "ApplicationResources" );  
  
        // Get the message template  
        String requiredFieldMessage = bundle.getString( "error.requiredfield" );  
  
        // Create a string array of size one to hold the arguments  
        String[] messageArgs = new String[1];  
  
        // Get the "Name" field from the bundle and load it in as an argument  
        messageArgs[0] = bundle.getString( "label.name" );  
  
        // Format the message using the message and the arguments  
        String formattedNameMessage =  
            MessageFormat.format( requiredFieldMessage, messageArgs );  
  
        System.out.println( formattedNameMessage );  
  
        // Get the "Phone" field from the bundle and load it in as an argument  
        messageArgs[0] = bundle.getString( "label.phone" );  
  
        // Format the message using the message and the arguments  
        String formattedPhoneMessage =  
            MessageFormat.format( requiredFieldMessage, messageArgs );  
  
        System.out.println( formattedPhoneMessage );  
    }  
}
```

Messages that contain variable data are known as *compound messages*. Using compound messages provides the ability to substitute application specific data into messages from the resource bundle during run-time. It can also reduce the number of messages that your application requires in the resource bundle, which can decrease the amount of time that it takes to translate to other locales..

Using compound messages in your resource bundles can make translation a little harder because the text contains substitution values that are not known until runtime. Human translators must take into account where the variable text goes in the localized message, because the substitution values may need to be in different positions in the message for different languages.

The Struts framework includes the capabilities of the `MessageFormat` class, but encapsulates the functionality behind the components within the framework.

Multilingual Support

Most of us cringe at the thought of supporting user groups that are in one of several possible locales. In many cases however, once an application has been installed and localized, it's like any other single-locale application. The users that access the application are either all from the same locale, or at least similar enough that the language and cultural differences are insignificant.

Multilingual applications on the other hand take internationalization to the next level by allowing users from different locales to access the same application. This means that the application has to be flexible enough to detect the user's locale and format everything based on that locale. Believe it or not, this is much harder to do than just localize an application. The issues dealing with currency alone can make you go screaming out of the room.

The discussion of building multilingual applications is so large that it can't be covered satisfactorily in a book of this type. For the remainder of this chapter, we'll stick with just the everyday internationalization problems that you'll face and not focus on multilingual support.

Internationalizing your Struts Applications

The internationalization support provided by the Struts framework focuses almost exclusively on the presentation of text and images for the application. Functionality such as accepting input from non-traditional languages is not covered within the Struts framework.

As you've already seen, depending on your Struts configuration settings, the framework can determine the preferred locale for a user and store it into the user's session. Once the user's locale has been determined, Struts can use this locale to lookup text and other resources from the resource bundle. The resource bundles are essential components in the Struts framework.

The Struts Resource Bundle

As you saw in Chapter 4, each one of your Struts applications can be configured with one or more resource bundles. The information within each bundle is available to actions, action forms, JSP pages, and custom tags alike.

Creating a Struts Resource Bundle

You need to create a resource bundle that follows the conventions of the `PropertyResourceBundle` class from the Java core library. That is, you need to create a text file that has an extension “.properties” and which adheres to the guidelines discussed in the JavaDoc for the `java.util.Properties` class. The most important of these guidelines is that the format of the messages within this file is:

```
| key=value |
```

Example 12-3 illustrates a properties file called *StorefrontResources.properties* that can be loaded by the Struts framework.

Example 12-3. A Simple Struts Resource Bundle

```
| error.requiredfield=The {0} field is required to save.  
| label.phone=Phone  
| label.name=Name |
```

You must be sure to name the message resource file with the extension “.properties” or the Struts framework will not be able to load it.

Notice that the keys used in examples 12-3 are separated by a period ‘.’ character. We could have also used other characters in the keys or we could have used a single-word key like *labelPhone=Phone*. Using name spacing in your keys is a great way to organize the localized text to make maintenance easier and to prevent name collisions. This is similar to how Java classes utilize package names.

You have to be careful however when using characters other than the period character as a separator. The colon “:” character for example can also be used to separate the key and the value instead of the equal sign “=” and will cause problems if you use it within your key. If you don’t want to use the period character in your keys, you can also safely use the underscore or the hyphen characters.

Resource Bundle Naming Guidelines

The naming of the resource bundle is critical to it working properly. All resource bundles have a base name that is selected by you. In example 12-3, the name *StorefrontResources* was used as a base name. If you needed to provide an additional localized resource bundle for the “French” language and the country “Canada”,

you would create a properties file called `StorefrontResouces_fr_CA.properties` with the appropriate localized text.

When the Struts framework searches for a message from one of the bundles, it looks for one that that is the closest match, based on the base name and the locale. If no locale is provided, it will use the default locale for the JVM. Only when it fails to find a resource bundle with a specific language and country code as part of the name will it default to the base resource bundle. The base resource bundle is the one without any language or country code in its name.

You should always provide a base resource bundle with the set of resource bundles for an application. If a user visits your site with a locale that you don't support, the application will select the base bundle for the user.

The Resource Bundle and the Classpath

The resource bundle needs to be placed in a location where it can be found and loaded. This means that the same class loader that loads the web application must also be able to locate and load the resource bundle. For web applications, this is generally the `WEB-INF/classes` directory.

If you provide a package name for the resource bundle, it must reside in the correct package as well. For example, if you named your resource bundle `com.oreilly.struts.StorefrontResources.properties`, then it must be placed into the `WEB-INF/classes/com/oreilly/struts` directory.

Accessing the Resource Bundle

The resource bundles for a Struts application are loaded at startup and each bundle is represented in memory by an instance of `org.apache.struts.util.MessageResources` class, or actually its concrete subclass `PropertyMessageResources`.

Each `MessageResources` instance is stored in the `ServletContext` when the application is initialized and can be accessed from just about any component within the servlet container. However, it's more typical that you'll use a combination of the custom tags and the `ActionMessage` or `ActionError` classes to access the resources, rather than calling methods on the `MessageResources` class directly.

For example, to create an instance of an `ActionError` object using one of the messages from the default resource bundle, you can do something like this:

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

```
if(getEmail() == null || getEmail().length() < 1) {
    errors.add("email", new ActionError("security.error.email.required"));
}
if(getPassword() == null || getPassword().length() < 1) {
    errors.add("password", new ActionError("security.error.password.required"));
}
return errors;
}
```

The String argument passed into the constructor of the `ActionError` class must be a key in the resource bundle. The `ActionMessage` and `ActionError` classes have several constructors, most of which allow you to pass substitution arguments in, for when your messages are compound messages, as explained earlier in this chapter.

The Bean Tag Library `MessageTag` Class

The Struts framework contains several custom tags that can be used in conjunction with the `MessageResources` for an application. One of the most important however, is the `Message` tag that is part of the Bean tag library.

This custom tag retrieves a message string from one of the bundles for an application. It supports optional parametric replacement if the JSP page requires it. All you need to do is to provide the key from the bundle and which application bundle, and the tag will write out the information to the JSP. For example, the following JSP fragment uses the `MessageTag` to write out the title of the HTML page:

```
<head>
<html:base/>
<title><bean:message key="global.title"/></title>
<script language=javascript src="include/scripts.js"></script>
<link
    rel="stylesheet"
    href="../stylesheets/format_win_nav_main.css" type="text/css"/>
</head>
```

This is one tag that you will find yourself using quite often within your Struts applications.

Setting the Character Set

Supporting character sets other than the typical US default *ISO-8859-1* is a little tricky. There are several steps that you must perform before your environment will be prepared to support them. You will need to configure the application server and/or servlet container to support the character-encoding scheme that you want to use. For example, for Unicode you would tell the container to interpret input as UTF-8. Check with the vendor's documentation, as each one will be configured differently. You can also setup a Servlet filter to do this, but this requires a container that supports the 2.3 Servlet API.

The `contentType` property within the `controller` element in the Struts configuration file needs to be set correctly. Set it to `"text/html; charset=UTF-8"` for html Unicode support. You can also specify this in the JSP pages by putting the following line at the top of the JSP page:

```
| <%@ page contentType="text/html; charset=utf-8" %>
```

and also this line in the HTML head section:

```
| <head>
|   <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8">
| </head>
```

Another option is to set the content type and encoding scheme in the response object like this:

```
| response.setContentType("text/html; charset=UTF-8");
```

But the problem is that you might have to do this in several different places, making maintenance and customization more difficult.

The last step regarding the content type and encoding scheme is that you may have to tell the browser to always send URLs as UTF-8. There's usually a checkbox or option to do this. In IE 5.5, it's in the advanced options section.

There are two final steps to accomplish to ensure your application can fully support Unicode. The first is to make sure your database is configured for Unicode. This is usually the default, but check to be safe. And second, if you're using the JRE rather than the SDK, you'll want to use the I18N version and not the US version.

Exception Handling and Internationalization

Exception handling was covered in detail back in chapter 10 and as you saw, there are also I18N issues that need to be considered when building an exception-handling framework for your application.

Unless you intend on localizing the exception messages that get thrown, you need to isolate the exception messages and be sure that they are never shown to the end user. The one thing that is probably more frustrating for an end user than getting an exception message or stack trace printed out on the screen, is getting one that is not in your native language.

As the chapter on exception handling pointed out, exceptions should be caught and localized messages should be displayed to the user. This can be accomplished by using the Struts message resource bundle and the `ActionError` class. You should never display a Java exception to the end user. Even when there's a system failure that can't be recovered from, you should still have a system error page that is localized for the user.

13

Struts and Enterprise JavaBeans (EJB)

As you've seen so far, you can use Struts to build both the controller and the view components of an MVC-based application. Struts isn't a framework for business logic or data access, so it doesn't play a role in the model component. First of all, this means that business logic (other than presentation validation) would be out of place in an action or form class. It also means that choosing to use Struts in an application shouldn't place any constraints on the design of the model. The separation of responsibilities in a layered architecture means that your Struts classes shouldn't care about how your model is implemented. Likewise, the model shouldn't care (or even know) that the controller and view are built using Struts.

Up to this point, the example applications presented in this book have used the web tier to provide everything needed from the middle tier, including the business logic and database access. With this approach, the application model has consisted simply of regular Java classes deployed in the web container. These classes have had complete responsibility for servicing requests from the action classes that depend on the application model. This architecture is common among web applications and it works well as long as the requirements in areas such as security, scalability, and transaction complexity stay within the limits of what a web container can do. Trying to do everything within the web tier can prove to be a challenge when these requirements, which aren't necessarily as high a priority in the design of a web container as they are in other container types, become too stringent.

The alternative to building the model into the web tier is to use a true application tier, such as a J2EE application server. With this approach the web tier provides the controller and view, and the application tier supplies the business data and its associated rules. Such

a design is appropriate when the scalability and transactional needs of an enterprise application require a more robust container. This situation is what we want to consider in this chapter. While it's true that the development of your Struts classes can be independent of the model implementation, this won't happen without some effort on your part. This chapter covers some of the issues you need to consider when developing an interface between your Struts actions and an application tier. In particular, the focus here is on interfacing to a model built using Enterprise JavaBeans.

To EJB or Not to EJB?

As the J2EE architecture initially grew in significance for enterprise application development, it was often assumed that an application that didn't include EJB wasn't a J2EE application at all. The rush was on to build enterprise-strength systems backed by an EJB container and all it had to offer. For carefully designed applications that required the type of infrastructure inherent with EJB, the technology provided a standards-based approach that led to many successful deployments. However, the focus on the EJB portion of J2EE also led to its use in applications that could have been better served by more lightweight approaches. This, along with some developers' disappointments with the pace of EJB's evolution, has led to somewhat of a backlash against using EJB at all. Developers fall on both sides of this issue, with some in the Struts community being quite vocal in their criticism of EJB. The more moderate opinion is that EJB offers value where its strengths are truly needed, but it can be costly both in performance and complexity otherwise.

This chapter won't attempt to argue the pros and cons of EJB. That topic falls outside the scope of this book. Instead, we'll simply presume that you've been asked to build a Struts-based web tier that needs to interface with an EJB application tier. With that as a starting point, our only concerns will be to identify the key issues to consider and come up with an effective approach for addressing them.

Implementing the Storefront Service Using EJB

Even though this chapter is specific to EJB, the intent is still to keep the focus on Struts. With that in mind, the discussion of EJB implementation details will be kept to a minimum. Even though EJB is a complex topic, the nature of several design patterns geared toward the interaction between EJBs and their clients makes this an easier task than you might first think. After all, an overriding goal of this chapter is to demonstrate how to design an application so that your Struts classes aren't impacted by the choice to use an EJB implementation of the model. You already have a head start on some of the central issues here after seeing how the model component of a web application can be hidden behind a service interface. In particular, you've seen through the Storefront example how easy it is to swap a debug model with a full implementation that accesses a database when this design approach is followed.

Throughout this chapter, the Storefront example will be used to illustrate how an EJB application tier can be used with a Struts application. If it weren't for the remote nature of accessing an EJB from a client application, this implementation choice wouldn't make any difference to you. However, the distributed aspects of EJB must be taken into account when your web tier classes access this type of model. What you'll see in the remainder of this chapter are some recommendations on how to best implement the code needed to interface with the application tier. Key to this discussion is an approach for isolating the code that handles what's unique about EJB so that your action classes aren't affected.

A Quick EJB Overview

The Enterprise JavaBeans Specification defines entity, session, and message-driven as the three EJB types. Each type of bean has a different purpose within an EJB application. Entity beans provide transactional access to persistent data and are most often used to represent rows stored in one or more related tables in a database. For example, you might implement `CustomerBean`, `ItemBean`, and `OrderBean` entity classes, among others, to support the Storefront application. These entity beans would incorporate the functionality provided by the corresponding business object classes in the example application. When entity beans are used, they take on the primary role of supplying the application model. They are expected to provide both the required data persistence operations and the business logic that governs the data they represent. Because the model should be reusable across applications in an enterprise, entity beans need to be independent of the type of clients that ultimately access the application tier.

Session beans are often described as extensions of the client applications they serve. They can implement business logic themselves, but more often their role is to coordinate the work of other classes (entity beans in particular). They are more closely tied to their clients, so the precaution to keep entity beans reusable doesn't apply to session beans to

the same extent. The application tier is primarily considered to be the application model, but session beans are also referred to as controllers because of the coordination they do. This is especially true when session bean methods are used to implement transactions that touch multiple business objects.

Session beans can be implemented as either stateless or stateful. A stateless session bean maintains no state specific to its client, so a single instance can be efficiently shared among many clients by the EJB container. A stateful bean instance, on the other hand, is assigned to a specific client so that state can be maintained across multiple calls. Holding state in the application tier can simplify the client application logic, but it makes it more difficult to scale to a large number of clients. Typical web applications maintain their client state in the user session, and possibly the database, instead of making use of stateful session beans. For this reason, we'll focus on stateless beans for our examples here.

The EJB 2.0 Specification added message-driven beans as the third bean type so that EJB could be integrated with the Java Message Service (JMS). Message-driven beans differ from the other two types in that they respond asynchronously to requests instead of being called directly by a client application. The container invokes a message-driven bean whenever a JMS message that meets the selection criteria of the bean is received. In a more complex version of the Storefront example, a message-driven bean could be used to respond to a notification that a delivery problem has resulted in an item becoming backordered. The bean could be responsible for emailing any customers that had orders already in place for the item. Message-driven beans have no direct interaction with client applications, so they are even less dependent on them than entity beans.

The Session Façade

The first step in designing an interface to the application tier is to identify the entry points that are exposed to a client application. Message-driven beans aren't called directly, so they don't come into play here. However, a typical EJB application can include a number of session and entity beans. As pointed out already, standard practice is to insulate entity beans from the details of the client so that they can be reused in other applications. If your Struts actions were to interact with entity beans directly, the web tier would quickly become coupled to the object model implemented by the application tier. This tight coupling combined with the distributed nature of EJB would lead to a number of issues:

- Changes in the EJB object model would require corresponding changes in the web tier
- Action classes would often be required to execute multiple remote calls to the application server to satisfy the business logic needs of a request

- The business logic and transaction management code needed to orchestrate multiple calls would have to exist in the web tier

To avoid these issues, the interface exposed to the clients of the application tier is limited to the session beans in almost all cases. You'll see this approach referred to as either *Session Wraps Entity* or a *Session Façade*. There are quite a few advantages to this design. When a client application makes a single call to a session bean method to perform a required operation, the session bean can easily execute the request as a single transaction and the implementation details can be hidden. This session bean method might need to access a number of entity beans, or even other session beans, to satisfy the request. No matter what the flow of control might be on the application server, the complexity is hidden from the client. Because session beans become the only clients of the entity beans when using a Session Façade, there's little chance of the entities becoming tied to any particular type of external client.

Even though the discussion here assumes that the business objects are implemented as entity beans, this doesn't have to be the case in an EJB application. The same concerns and advantages that support using a Session Façade apply when other implementations are used as well. Just as some Java developers don't like EJB, not all EJB developers like entity beans. Because the Session Façade hides the object model from the client, entity beans could be replaced with another approach, such as Java Data Objects (JDO) or regular data access objects, without impacting the interface exposed to the client.

The Business Interface

When using a Session Façade, you must first define the details of this interface between the web and application tiers. You might have some question here over which side should be the primary driver of this contract between the two tiers. Early in the development of the Storefront example, the *IStoreFrontService* interface was introduced to define the application tier functionality required to support the presentation needs of the application. In particular, the presentation layer relies on the supporting service to authenticate users and provide product descriptions needed to build the online catalog. Taking a user-centered view of the application, it's easy to see how the service layer requirements can be driven by the web tier. After all, if a certain view is to be built, the model functionality has to exist to support it. However, one of the main reasons to implement an enterprise's business logic within EJBs is to allow those business rules and the supporting data to be used to across multiple applications. This includes providing support for clients other than those associated with web applications. This isn't a problem because the division of responsibility between session and entity beans helps protect the reusability of business logic.

Treating session beans as extensions of the clients they serve means that there's nothing wrong with defining a Session Façade that's specific to a particular client application. As

long as your entity and message-driven beans remain independent of the client, it's reasonable to implement a session bean interface in response to the requirements set forth by a specific client. If multiple client views of the model are required, multiple façades can be implemented to support them.

The façade presented by a session bean can support either local or remote clients. Local clients of a session or entity bean are restricted to being other EJBs deployed within the same container. These clients are tightly coupled to the beans they access, but they offer performance advantages when calls need to be made between EJBs. This is because these method calls use pass-by-reference semantics instead of being treated as remote calls between distributed components. Session beans are often local clients of entity beans, but it's less common for them to have local clients of their own to support. Web tier components obviously aren't EJBs running within the application tier, so we only care about remote clients for our purposes here. What we need to do then is define the remote interface for our Session Façade.

Every session bean that supports remote clients must have a corresponding remote interface that extends the `javax.ejb.EJBObject` interface. It's this interface that determines the business methods that are exposed by the session bean. It might seem strange, but you'll almost never see a method explicitly declared in a remote interface. This is because of an EJB design pattern known as the *Business Interface*.

When you think of a class and an interface that it's associated with, you would normally expect that the class would explicitly implement that interface. This isn't true with remote (or local) interfaces in EJB. Instead, the container creates an intermediate object (often referred to as an *EJBObject*) to implement the remote interface. This object intercepts calls made by clients of the bean and then delegates them to the implementation class after performing any operations such as security or transaction management that might be required. Instead of the Java compiler verifying that the bean class implements each of the business methods declared by the remote interface, that responsibility falls to the deployment tools provided by the container. Even a bean class that compiles without any errors will fail at deployment if there's a mismatch between it and its remote interface.

If you declared a session bean to implement its remote interface you'd be guaranteed that the compiler would catch any problems with its business method declarations. The problem is that you'd also have to provide dummy implementations of the non-business methods declared by `javax.ejb.EJBObject`. These methods would never be called (they're called only on the intermediate object created by the container), but they would have to be there to satisfy the compiler. Instead of taking this approach, most EJB developers create a second interface, known as the Business Interface, that declares the business methods that need to be exposed. By declaring the remote interface to extend this interface and the bean class to implement it, the required methods are exposed and the compiler can verify that the bean implements them. This pattern provides us a convenient starting point for defining our client access mechanism.

The use of a Business Interface also prevents a programmer from accidentally passing or returning a *this* reference to an instance of a bean class that has been declared to implement its remote interface. This topic is beyond the scope of this book, but the short explanation is that the EJB container can only manage bean instances properly when they're referred to using only their remote (or local) interfaces. A bean reference can't be returned in place of its remote interface if the bean class implements only its Business Interface.

Returning to the `IStorefrontService` interface that must eventually be satisfied by our implementation, recall that it contains methods related to both user authentication and the product catalog. Even using a Session Faade, you would likely separate these responsibilities into separate session beans. This would reduce the complexity of the session beans involved and simplify their maintenance. However, given that EJB design isn't our focus here, our first simplification will be to assume that our faade will consist of a single Storefront session bean. You probably wouldn't do this in a real application, but once you know how to interface with a single session bean, applying the same technique to multiples session beans is straightforward. With this decision made, a suitable Business Interface for the Storefront session bean is shown in Example 13-1.

Example 13-1. The Business Interface for the Storefront session bean

```
package com.oreilly.struts.storefront.service;

import java.rmi.RemoteException;
import java.util.List;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.exceptions.*;

/**
 * The Business Interface for the Storefront Session Bean
 */
public interface IStorefront {

    public UserView authenticate( String email, String password )
        throws InvalidLoginException, ExpiredPasswordException,
            AccountLockedException, DatastoreException, RemoteException;

    public List getFeaturedItems() throws DatastoreException, RemoteException;

    public ItemDetailView getItemDetailView( String itemId )
        throws DatastoreException, RemoteException;
}
```

The first thing to notice about the `IStorefront` interface is that its methods don't exactly match those declared by `IStorefrontService`. First of all, our Business Interface doesn't include the `logout()` and `destroy()` methods found in the service interface. The reason for this is that those methods represent web tier functionality and not true business logic that needs to move to the application tier. Besides the missing

methods, you can also see that every method in `IStorefront` is declared to throw `RemoteException`, which is not a part of the declarations in `IStorefrontService`. All business methods exposed to a remote client of an EJB must be declared to throw `RemoteException`. This is the one aspect of a remote interface that can't be hidden by the Business Interface. Without this restriction, this interface could be made to look very much like our service interface. Once we cover how our example session bean will be implemented, we'll get to how these mismatches between the interfaces can be handled.

It's also important to notice that our Business Interface is referencing the view classes already created to support the service interface. The same Data Transfer Object (DTO) pattern introduced in Chapter 7 applies to an EJB-based model. Instead of exposing the actual business object implementation classes, or many fine-grained methods to access their properties, simple JavaBean classes can be used to communicate the state of the model with the client.

With our Business Interface defined, Example 13-2 shows the trivial remote interface declaration we'll need to eventually deploy our session bean.

```
Example 13-2. The remote interface for the Storefront session bean
package com.oreilly.struts.storefront.service;

import javax.ejb.EJBObject;

public interface Storefront extends EJBObject, IStorefront {
    /**
     * The remote interface for the Storefront session bean. All methods are
     * declared in the IStorefront Business Interface.
     */
}
```

Stateless Session Bean Implementation

Without getting into anything too elaborate, we next want to come up with an implementation of our Session Faade. We'll make a few decisions here to simplify things, but the result will be all we need to illustrate how to interface the web and application tiers. It will also be good enough for you to deploy in an EJB container and test your Struts interface.

If you were given the task of implementing the application tier of the Storefront application using EJB, you would likely produce a design consisting of both session and entity beans. You could represent the model components using entity beans and you'd likely have a number of session beans to provide the functionality for security, catalog, and order operations. The session beans would provide the workflow functionality required from process business objects and the entity beans would serve as the corresponding entity business objects.

We've already made the decision to use only a single session bean for the example. The Session Façade makes our next simplification easy as well. Because we've isolated the interface between our two tiers into a façade, any division of responsibilities between session and entity beans is of no concern to us as Struts developers. The web tier sees only session bean methods and DTO classes, so nothing else about the implementation will affect the web components. Given that, we'll implement our façade using a single stateless session bean that does not require any other EJBs.

If you're starting with an EJB implementation that includes entity beans, you might want to use XDoclet (available from <http://www.sourceforge.net/projects/xdoclet>) to automatically generate Struts action forms from these beans. For more complex EJB implementations than what we're looking at here, XDoclet also provides an automated means of generating the various interfaces and deployment descriptors required for a bean. This code generation is performed based on special Javadoc tags that you include in your EJB implementation classes.

Because entity beans are not being used, we can make use of the same ORM approach and entity business object classes already used by the `StorefrontServiceImpl` class. In fact, our implementation will look very much like that class with the exception of the callback methods required by the `javax.ejb.SessionBean` interface. This is shown in Example 13-3.

Example 13-3. The Storefront session bean

```
package com.oreilly.struts.storefront.service;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import org.odmg.*;
import ojb.odmg.*;
import com.oreilly.struts.storefront.businessobjects.CustomerBO;
import com.oreilly.struts.storefront.businessobjects.ItemBO;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.catalog.view.ItemSummaryView;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.exceptions.AccountLockedException;
import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;
import com.oreilly.struts.storefront.framework.exceptions.ExpiredPasswordException;
import com.oreilly.struts.storefront.framework.exceptions.InvalidLoginException;

/**
 * This is a simple Session Bean implementation of the Storefront service
 */
```

```
public class StorefrontBean implements SessionBean, IStorefront {
    private SessionContext ctx;
    private Implementation odmng = null;
    private Database db = null;

    public UserView authenticate( String email, String password )
        throws InvalidLoginException, ExpiredPasswordException,
        AccountLockedException, DatastoreException {

        // Start a transaction
        Transaction tx = odmng.newTransaction();
        tx.begin();

        // Query the database for a user that matches the credentials
        List results = null;
        try{
            OQLQuery query = odmng.newOQLQuery();
            // Set the OQL select statement
            String queryStr = "select customer from " + CustomerBO.class.getName();
            queryStr += " where email = $1 and password = $2";
            query.create(queryStr);

            // Bind the input parameters
            query.bind( email );
            query.bind( password );

            // Retrieve the results and commit the transaction
            results = (List)query.execute();
            tx.commit();
        }catch( Exception ex ){
            // Rollback the transaction
            tx.abort();
            ex.printStackTrace();
            throw DatastoreException.datastoreError(ex);
        }

        // If no results were found, must be an invalid login attempt
        if ( results.isEmpty() ){
            throw new InvalidLoginException();
        }

        // Should only be a single customer that matches the parameters
        CustomerBO customer = (CustomerBO)results.get(0);

        // Make sure the account is not locked
        String accountStatusCode = customer.getAccountStatus();
        if ( accountStatusCode != null && accountStatusCode.equals( "L" ) ){
            throw new AccountLockedException();
        }

        // Populate the Value Object from the Customer business object
        UserView userView = new UserView();
        userView.setId( customer.getId().toString() );
        userView.setFirstName( customer.getFirstName() );
    }
}
```

```
        userView.setLastName( customer.getLastName() );
        userView.setEmailAddress( customer.getEmail() );
        userView.setCreditStatus( customer.getCreditStatus() );

        return userView;
    }

    public List getFeaturedItems() throws DatastoreException {
        // Start a transaction
        Transaction tx = odmg.newTransaction();
        tx.begin();

        List results = null;
        try{
            OQLQuery query = odmg.newOQLQuery();
            // Set the OQL select statement
            query.create( "select featuredItems from " + ItemBO.class.getName() );
            results = (List)query.execute();
            tx.commit();
        }catch( Exception ex ){
            // Rollback the transaction
            tx.abort();
            ex.printStackTrace();
            throw DatastoreException.datastoreError(ex);
        }
        int size = results.size();
        List items = new ArrayList();
        for( int i = 0; i < size; i++ ){
            ItemBO itemBO = (ItemBO)results.get(i);
            ItemSummaryView newView = new ItemSummaryView();
            newView.setId( itemBO.getId().toString() );
            newView.setName( itemBO.getDisplayLabel() );
            newView.setUnitPrice( itemBO.getBasePrice() );
            newView.setSmallImageURL( itemBO.getSmallImageURL() );
            newView.setProductFeature( itemBO.getFeature1() );
            items.add( newView );
        }
        return items;
    }

    public ItemDetailView getItemDetailView( String itemId )
    throws DatastoreException {
        // Start a transaction
        Transaction tx = odmg.newTransaction();
        tx.begin();

        List results = null;
        try{
            OQLQuery query = odmg.newOQLQuery();

            // Set the OQL select statement
            String queryStr = "select item from " + ItemBO.class.getName();
            queryStr += " where id = $1";
            query.create(queryStr);
            query.bind(itemId);
```

```
        // Execute the transaction
        results = (List)query.execute();
        tx.commit();
    }catch( Exception ex ){
        // Rollback the transaction
        tx.abort();
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
    }

    //
    if (results.isEmpty() ){
        throw DatastoreException.objectNotFound();
    }

    ItemBO itemBO = (ItemBO)results.get(0);

    // Build a ValueObject for the Item
    ItemDetailView view = new ItemDetailView();
    view.setId( itemBO.getId().toString() );
    view.setDescription( itemBO.getDescription() );
    view.setLargeImageUrl( itemBO.getLargeImageUrl() );
    view.setName( itemBO.getDisplayLabel() );
    view.setProductFeature( itemBO.getFeature1() );
    view.setUnitPrice( itemBO.getBasePrice() );
    view.setTimeCreated( new Timestamp(System.currentTimeMillis()) );
    view.setModelNumber( itemBO.getModelNumber() );
    return view;
}

/**
 * Opens the database and prepares it for transactions
 */
private void init() throws DatastoreException {
    // get odmng facade instance
    odmng = OJB.getInstance();
    db = odmng.newDatabase();
    //open database
    try{
        db.open("repository.xml", Database.OPEN_READ_WRITE);
    }catch( ODMGException ex ){
        throw DatastoreException.datastoreError(ex);
    }
}

public void ejbCreate() throws CreateException {
    try {
        init();
    }catch ( DatastoreException e ) {
        throw new CreateException(e.getMessage());
    }
}

public void ejbRemove() {
```



```
        try {
            db.close();
        } catch ( ODMGException e ) {}
    }

    public void setSessionContext( SessionContext assignedContext ) {
        ctx = assignedContext;
    }

    public void ejbActivate() {
        // nothing to do for a stateless bean
    }

    public void ejbPassivate() {
        // nothing to do for a stateless bean
    }
}
```

In our `StorefrontBean` class, the business method implementations are unchanged from the `StorefrontServiceImpl` versions. Only the management of the database connection needed to be modified. Whenever the EJB container creates a new instance of this bean, the `ejbCreate()` callback method is invoked and a database connection is established. This connection is closed in the corresponding `ejbRemove()` method that is called prior to the instance being destroyed. The container never passivates stateless session beans, so do-nothing implementations are supplied for the `ejbPassivate()` and `ejbActivate()` methods of the `SessionBean` interface.

A more correct EJB approach would be to open the database connection using a `javax.sql.DataSource` connection factory obtained from a JNDI lookup. This allows the container to manage connection polling and transaction enlistment for you automatically. Again, this doesn't affect our interface so we can continue on using this simple implementation.

Our session bean now has a remote interface and an implementation class. That leaves the home interface, which is always simple in the case of a stateless session bean. All we need is a single create method as shown in Example 13-4.

Example 13-4. The home interface for the Storefront session bean

```
package com.oreilly.struts.storefront.service;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

/**
 * The home interface for the Storefront Session Bean
 */
public interface StorefrontHome extends EJBHome {
```

```
    public Storefront create() throws CreateException, RemoteException;
}
```

JBoss Deployment

We need to select an EJB container and create the required XML deployment descriptors before we can deploy and use our session bean. The open-source JBoss application server fits our requirements perfectly here. This full-featured J2EE implementation complete with EJB 2.0 support is a favorite among open-source developers. You can download the software for free from <http://www.jboss.org>.

With our minimal implementation, we don't need anything complicated as far as deployment information for our session bean. Example 13-5 shows the standard *ejb-jar.xml* descriptor for our bean. For the most part, this file simply identifies the home and remote interfaces and the implementation class. It also declares that all of the business methods should be executed within a transaction.

*Example 13-5. The **ejb-jar.xml** deployment descriptor for the Storefront session bean*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar >

    <description>
        Generic deployment information for the Storefront Session Bean
    </description>
    <display-name>Storefront Session Bean</display-name>

    <enterprise-beans>
        <session >
            <ejb-name>Storefront</ejb-name>
            <home>com.oreilly.struts.storefront.service.StorefrontHome</home>
            <remote>com.oreilly.struts.storefront.service.Storefront</remote>
            <ejb-class>
                com.oreilly.struts.storefront.service.StorefrontBean
            </ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>

    <assembly-descriptor >
        <container-transaction >
            <method >
                <ejb-name>Storefront</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

```
        </container-transaction>
    </assembly-descriptor>

</ejb-jar>
```

In addition to the *ejb-jar.xml* file, most containers require one or more vendor-specific descriptors as part of a bean's deployment information. In this case, all we need to do is associate a JNDI name with our bean. Example 13-6 shows how this is done with JBoss. The fully-qualified name of the bean's remote interface was chosen as the JNDI name. It's also common to see the home interface name used.

Example 13-6. The JBoss deployment descriptor for the Storefront session bean

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
    "http://www.jboss.org/j2ee/dtd/jboss.dtd">

<jboss>

    <enterprise-beans>
        <session>
            <ejb-name>Storefront</ejb-name>
            <jndi-name>com.oreilly.struts.storefront.service.Storefront</jndi-name>
        </session>
    </enterprise-beans>

</jboss>
```

Deployment of an EJB requires packaging it into a JAR. The deployment JAR file for our session bean needs to include the following files:

- The home and remote interface class files
- The bean implementation class file
- The two deployment descriptors (these files must be placed in a *META-INF* directory)
- The *OJB.properties* file and the various repository XML files used by the ORM framework
- The business object and DTO class files referenced by the Storefront bean

Once you've created this JAR file, you can deploy the bean by copying the file to the *server/default/deploy* directory underneath your JBoss installation. You can place the JAR files for your JDBC driver and the OJB classes in the *server/default/lib* directory. At this point, you can start JBoss and verify that you have everything in place to execute the application tier.

Interfacing Struts to EJB

It's now time to turn our attention back to the client side of our Session Façade. In this section, we'll first cover how to satisfy the requirements of our service interface with our session bean implementation. We'll then look at how to better manage the JNDI lookups and home and remote interface management inherent with being a remote client to an EJB.

Using a Business Delegate

As you saw when we defined the Business Interface for the Storefront session bean, we still have some work to do to match it up to the Storefront service interface. Our Business Interface doesn't include all the methods of `IStorefrontService` and the methods that are declared include `RemoteException` in their `throws` clauses. The way we'll address these differences is to go back to the Business Delegate pattern introduced in Chapter 6. Remember that the purpose of this pattern is to hide the business service implementation from the client application.

We'll start out with a fairly straightforward Business Delegate implementation and then cover some specific ways to improve it. An initial implementation is shown in example 13-7.

Example 13-7.A Business Delegate for the Storefront session bean

```
package com.oreilly.struts.storefront.service;

import java.rmi.RemoteException;
import java.util.Hashtable;
import java.util.List;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.exceptions.*;

/**
 * This class is a Business Delegate that supports the implementation of the
 * IStorefrontService interface using the Storefront session bean.
 */
public class StorefrontEJBDelegate implements IStorefrontService {

    private IStorefront storefront;

    public StorefrontEJBDelegate() {
        init();
    }
}
```

```
private void init() {
    try {
        Hashtable props = new Hashtable();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.PROVIDER_URL, "localhost");

        InitialContext ic = new InitialContext(props);
        Object home = ic.lookup(
            "com.oreilly.struts.storefront.service.Storefront");

        StorefrontHome sfHome = (StorefrontHome)
            PortableRemoteObject.narrow(home, StorefrontHome.class);
        storefront = sfHome.create();
    }
    catch (NamingException e) {
        throw new RuntimeException(e.getMessage());
    }
    catch (CreateException e) {
        throw new RuntimeException(e.getMessage());
    }
    catch (RemoteException e) {
        throw new RuntimeException(e.getMessage());
    }
}

public UserView authenticate( String email, String password )
throws InvalidLoginException, ExpiredPasswordException,
    AccountLockedException, DatastoreException {
    try {
        return storefront.authenticate(email, password);
    }
    catch (RemoteException e) {
        throw DatastoreException.datastoreError(e);
    }
}

public List getFeaturedItems() throws DatastoreException {
    try {
        return storefront.getFeaturedItems();
    }
    catch (RemoteException e) {
        throw DatastoreException.datastoreError(e);
    }
}

public ItemDetailView getItemDetailView( String itemId )
throws DatastoreException {
    try {
        return storefront.getItemDetailView(itemId);
    }
    catch (RemoteException e) {
        throw DatastoreException.datastoreError(e);
    }
}
```

```
}

public void logout( String email ) {
    // Do nothing for this example
}

public void destroy() {
    // Do nothing for this example
}
}
```

When an instance of the `StorefrontEJBDelegate` class is created, its `init()` method is called to obtain a remote reference to the Storefront session bean. This method performs the required JNDI lookup using the naming service implementation provided by JBoss. As written, the delegate assumes that the naming service is running on the local machine. We'll look later at how to externalize the details of the JNDI lookup that must be performed by a delegate. Once a remote reference is obtained, the delegate holds it as part of its state. This field is declared to be of the Business Interface type because we only need it for accessing business methods.

Other than what is required to obtain a remote reference, most of the code in our delegate does nothing more than relay business method calls to the session bean implementation. The `logout()` and `destroy()` methods have no counterpart in the application tier, so they would be addressed by web tier functionality if we had anything we needed to do in them for the example.

Exception Handling

The exception handling found in this implementation of the `StorefrontEJBDelegate` class is worth noting. Besides hiding the details of JNDI lookups, a Business Delegate used with a session bean should also hide the EJB-specific exceptions that come with being a remote client. In the business methods of the delegate, any `RemoteException` that gets thrown from a session bean call is caught and reported to the client using a `DatastoreException`. Hiding the remote nature of the model implementation addresses the mismatch in declared exceptions found between our Business Interface and the `IStorefrontService` declarations.

If the inclusion of `RemoteException` in our Business Interface had been the only difference between this and the service interface, it might have been tempting to simply add this exception to `IStorefrontService` and continue forward. However, this would have unnecessarily cluttered the contract for whatever service implementation might be used with implementation details.

The only reason our delegate uses a `DatastoreException` to respond to a `RemoteException` is to leave the service interface unaffected by the implementation approach. If this constraint were relaxed, a better approach would be to declare an exception class whose sole purpose is to report exceptions from a delegate in a generic

fashion. If we were to declare an application exception named `ServiceDelegateException`, we could throw that when a `RemoteException` occurs. This new exception would be a more accurate indication of the type of error that occurred when compared to using `DatastoreException`. The best part is that adding this new exception to our `IStorefrontService` declarations still wouldn't expose the fact that the implementation was based on EJB. Instead of throwing a `RuntimeException` to report a failure in obtaining a remote reference, the `init()` method could also be updated to make use of `ServiceDelegateException`.

Swapping the Implementation

All that's left to do is to swap the current Storefront service implementation with the delegate we have created. The framework put into place with the `StorefrontServiceFactory` in Chapter 6 makes this easy to do. We only need to change the class specified for our service implementation in the `web.xml` file to the following:

```
<init-param>
  <param-name>storefront-service-class</param-name>
  <param-value>
    com.oreilly.struts.storefront.service.StorefrontEJBDelegate
  </param-value>
</init-param>
```

With this change made, an action will be creating a delegate instance whenever it calls the `getStorefrontService()` method implemented in the `StorefrontBaseAction`. This method should only be called once during a request to avoid the unnecessary overhead of creating additional remote references. Even taking care to use the same delegate throughout a request leaves us with an implementation that isn't very efficient. The next section will cover some ways to improve our use of JNDI and home interfaces.

Don't forget that you'll need to copy the JBoss client JARs to the `lib` directory for your web application before using your delegate. You'll also need the home and remote interface class files for the Storefront session bean in the `classes` directory.

Managing EJB Home and Remote References

Implementing a Business Delegate clearly isolates and minimizes the dependencies between the web and application tiers. We were able to implement our Storefront session bean using a Business Interface that isn't tied to any particular client type. We were also able to leave our Struts action classes untouched when switching to this implementation of our model. We do have a couple of problems to address though to turn this into a solution you would want to use in a real application. Most importantly, we need to improve how we're obtaining our home interface references. We should also get rid of the hard-coded parameters used by our JNDI lookup.

Performing a JNDI lookup to obtain a home interface reference is an expensive (meaning slow) operation. We couldn't do much about this overhead if we actually needed a new home reference for each request, but that's not the case. An EJB home is a factory object that is valid throughout the lifetime of the client application. There is no state in this object that prevents it from being used across requests or client threads. Our delegate would be significantly improved if the home reference it needed were cached within the web tier after being requested the first time.

As with any design problem, there is more than one technique we should consider for caching our home reference. We're basically talking about application scope data in the web tier, so modifying the delegate to store the reference in the `ServletContext` after doing the required JNDI lookup is a potential solution. This would prevent any additional lookups, but it would require us to make the `ServletContext` available to our delegate through its constructor. This one change would ripple out to our service factory as well because it currently instantiates an `IStorefrontService` implementation using its no-argument constructor. It would be preferable to choose a solution without such a strong tie to HTTP constructs. A more flexible approach is to apply the `EJBHomeFactory` pattern as a way to cache the references we need.

Implementing an `EJBHomeFactory`

The `EJBHomeFactory` pattern is defined in the *EJB Design Patterns* book written by Floyd Marinescu. Implementing this pattern allows you to create and cache any EJB home reference needed by your application. Because it has no dependency on the `ServletContext`, you can reuse this technique in non-web applications. Example 13-8 shows the implementation of this pattern that we'll use for the Storefront application.

Example 13-8. `EJBHomeFactory` implementation

```
package com.oreilly.struts.storefront.framework.ejb;

import java.io.InputStream;
import java.io.IOException;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

/**
 * This class implements the EJBHomeFactory pattern. It performs JNDI
 * lookups to locate EJB homes and caches the results for subsequent calls.
 */
public class EJBHomeFactory {
    private Map homes;
    private static EJBHomeFactory singleton;
    private Context ctx;

    private EJBHomeFactory() throws NamingException {
        homes = Collections.synchronizedMap(new HashMap());
        try {
```



```
// load the properties file from the classpath root
InputStream inputStream = getClass().getResourceAsStream(
    "/jndi.properties" );
if ( inputStream != null ) {
    Properties jndiParams = new Properties();
    jndiParams.load( inputStream );

    Hashtable props = new Hashtable();
    props.put(Context.INITIAL_CONTEXT_FACTORY,
        jndiParams.get(Context.INITIAL_CONTEXT_FACTORY));
    props.put(Context.PROVIDER_URL, jndiParams.get(Context.PROVIDER_URL));
    ctx = new InitialContext(props);
}
else {
    // use default provider
    ctx = new InitialContext();
}
} catch( IOException ex ){
    // use default provider
    ctx = new InitialContext();
}
}

/**
 * Get the Singleton instance of the class
 */
public static EJBHomeFactory getInstance() throws NamingException {
    if (singleton == null) {
        singleton = new EJBHomeFactory();
    }
    return singleton;
}

/**
 * Specify the JNDI name and class for the desired home interface
 */
public EJBHome lookupHome(String jndiName, Class homeClass)
    throws NamingException {
    EJBHome home = (EJBHome)homes.get(homeClass);
    if (home == null) {
        home = (EJBHome)PortableRemoteObject.narrow(ctx.lookup(
            jndiName), homeClass);
        // cache the home for repeated use
        homes.put(homeClass, home);
    }
    return home;
}
}
```

Notice that our **EJBHomeFactory** class accepts the JNDI name and class for the home interface it is requested to locate. If we needed to access more than one session bean in the application tier, we would simply implement a delegate class for each session bean and use our factory to locate the corresponding home interface. Besides the performance

improvements the caching of homes gives us, all the ugly narrowing and exception handling that goes along with looking up these references is kept in one place.

The `EJBHomeFactory` constructor also takes care of externalizing the provider and factory parameters we need to access the naming service. A standard approach for doing this is to use a *jndi.properties* file that includes entries like the following:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost
```

If you call the no-argument constructor for the `InitialContext` class, the classpath is searched for a *jndi.properties* file. If this file is found, its entries are used to initialize the naming context. In our example, the factory class explicitly loads this file from the classpath. Otherwise, classloader priorities within the web server could prevent these settings from being picked up before the default values defined for the server.

The `JNDIConnectorPlugin` example in Chapter 9 demonstrated how the naming service parameters could be read from the *web.xml* file and stored in the `ServletContext`. We're not using that approach here because we want to keep our factory independent of the web tier.

Using an `EJBHomeFactory` in a Business Delegate

Our Business Delegate can be simplified now that we have a standard approach for locating the home interface. We can change the implementation of the `init()` method to the following:

```
private void init() {
    try {
        StorefrontHome sfHome = (StorefrontHome)EJBHomeFactory.getInstance().
            lookupHome("com.oreilly.struts.storefront.service.Storefront",
                StorefrontHome.class);
        storefront = sfHome.create();
    }
    catch (NamingException e) {
        throw new RuntimeException(e.getMessage());
    }
    catch (CreateException e) {
        throw new RuntimeException(e.getMessage());
    }
    catch (RemoteException e) {
        throw new RuntimeException(e.getMessage());
    }
}
```

What about the Remote References?

In our implementation, a remote reference to the session bean is created for each request. This isn't a problem from a performance standpoint because the overhead attached to creating a remote reference pales in comparison to that associated with the home. However, this doesn't mean that you can't cache remote references if you want. In fact, if

you're interfacing with a stateful session bean, you can't keep creating new remote references across requests or you won't be calling the same bean instance each time.

You can avoid creating a new remote reference per request by caching a Business Delegate instance in the session. Unlike homes, you can't cache a remote reference as application scope data. Even for stateless session beans, a remote reference holds information tied to the client thread that created it. If you cache the Business Delegate, there are some important changes to make. This is because the user session could be serialized and restored by the web container and a remote reference isn't required to be serializable. For a stateless session bean, you need to be able to create a new remote reference if necessary. For a stateful session bean, you need to hold an EJB handle in your delegate instead of a remote reference so that you can always maintain a way to access the same bean.

Using Dynamic Proxies

In this section we'll look at one last example of something you might want to implement within your Business Delegate. Our current implementation works well for the Storefront application, but delegates tend to become cluttered with redundant-looking methods if they have to support an interface with more than a few methods. We only have three methods declared for our overly simple Storefront model, but even they follow a somewhat monotonous pattern. With the exception of the `logout()` and `destroy()` methods, each business method in the delegate is implemented by calling the method with the same name on the session bean and catching `RemoteException` to replace it with a `DatastoreException`. A dynamic proxy offers a way to get rid of this redundancy.

If you ever find yourself performing the same additional steps as part of delegating a set of method calls to another object, you should consider introducing a dynamic proxy. This concept is a little difficult to grasp if you've never worked with one before, but its use can do away with a lot of repetitive code. Basically, a dynamic proxy is an object created at runtime using reflection that implements one or more interfaces you specify. The implementation of the interface methods consists of calling the `invoke()` method of an object you also specify. This `invoke()` method is declared by the `java.lang.reflect.InvocationHandler` interface, which must be explicitly implemented by the object used to construct the proxy. The proxy passes parameters to the `invoke()` method that identify the interface method that was called and the arguments that were passed to it. This idea is always easiest to explain by example, so Example 13-9 shows a replacement for our Business Delegate that can be used with a dynamic proxy.

Example 13-9 Dynamic proxy implementation of the Storefront service

```
package com.oreilly.struts.storefront.service;

import java.lang.reflect.*;
import java.rmi.RemoteException;
```

```
import java.util.*;
import javax.ejb.CreateException;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import com.oreilly.struts.storefront.catalog.view.ItemDetailView;
import com.oreilly.struts.storefront.customer.view.UserView;
import com.oreilly.struts.storefront.framework.ejb.EJBHomeFactory;
import com.oreilly.struts.storefront.framework.exceptions.*;

/**
 * This class is a dynamic proxy implementation of the IStorefrontService
 * interface. It implements two of the IStorefrontService methods itself and
 * delegates the others to the methods declared by the IStorefront Business
 * Interface with the same name.
 */
public class DynamicStorefrontEJBDelegate implements InvocationHandler {

    private IStorefront storefront;
    private Map storefrontMethodMap;

    public DynamicStorefrontEJBDelegate() {
        init();
    }

    private void init() {
        try {
            // get the remote reference to the session bean
            StorefrontHome sfHome = (StorefrontHome)EJBHomeFactory.getInstance().
                lookupHome("com.oreilly.struts.storefront.service.Storefront",
                    StorefrontHome.class);
            storefront = sfHome.create();

            // store the Business Interface methods for later lookups
            storefrontMethodMap = new HashMap();
            Method[] storefrontMethods = IStorefront.class.getMethods();
            for (int i=0; i<storefrontMethods.length; i++) {
                storefrontMethodMap.put(storefrontMethods[i].getName(),
                    storefrontMethods[i]);
            }
        }
        catch (NamingException e) {
            throw new RuntimeException(e.getMessage());
        }
        catch (CreateException e) {
            throw new RuntimeException(e.getMessage());
        }
        catch (RemoteException e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    public void logout(String email) {
        // Do nothing for this example
    }
}
```

```
public void destroy() {
    // Do nothing for this example
}

public Object invoke(Object proxy, Method method, Object[] args )
throws Throwable{
    try {
        // check for the two methods implemented by this class
        if (method.getName().equals("logout")) {
            logout((String)args[0]);
            return null;
        }
        else if (method.getName().equals("destroy")) {
            destroy();
            return null;
        }
        else {
            // this method should match a method implemented by the
            // session bean that has the same name and argument list
            Method storefrontMethod = (Method)storefrontMethodMap.get(
                method.getName());
            if (storefrontMethod != null) {
                // call the method on the remote interface
                return storefrontMethod.invoke( storefront, args );
            }
            else {
                throw new NoSuchMethodException("The Storefront does not implement "
                    + method.getName());
            }
        }
    } catch( InvocationTargetException ex ) {
        if (ex.getTargetException() instanceof RemoteException) {
            // RemoteException isn't declared by the IStorefrontService method that was
            // called, so we have to catch it and throw something that is
            throw DatastoreException.datastoreError(ex.getTargetException());
        }
        else {
            throw ex.getTargetException();
        }
    }
}
```

The intent behind `DynamicStorefrontEJBDelegate` is for a dynamic proxy that is created as an implementation of the `IStorefrontService` interface to delegate all those calls to the `invoke()` method declared here. Notice that this delegate class is not declared to implement `IStorefrontService`. In fact, the only business methods from `IStorefrontService` that appear in this class are the `logout()` and `destroy()` methods that aren't implemented by our session bean.

To use the dynamic proxy-based delegate, we need to modify our approach for obtaining an implementation of the service interface from the factory. Rather than devising

something elegant for a small part of this example, we'll ignore that here and just hard-code the new approach we need. This is shown in the following version of the `createService()` method of `StorefrontServiceFactory`:

```
public IStorefrontService createService(){
    Class[] serviceInterface = new Class[] { IStorefrontService.class };
    IStorefrontService proxy = (IStorefrontService)Proxy.newProxyInstance(
        Thread.currentThread().getContextClassLoader(), serviceInterface,
        new DynamicStorefrontEJBDelegate() );
    return proxy;
}
```

When an action class asks for an implementation of the service interface, the factory now creates a dynamic proxy that implements this interface using an instance of `DynamicStorefrontEJBDelegate`. When the action makes a call on the service interface, the call goes to the proxy and is transformed into a call on the delegate's `invoke()` method. The `invoke()` method checks the name of the method that was called and either calls the `logout()` or `destroy()` method implemented in the class or delegates it to the session bean method with the same name. This sequence of calls is illustrated in Figure 13-1. The trapping and replacement of `RemoteException` when our Business Delegate calls a session bean method are now handled in a single place. No matter how many methods are exposed by the session bean Business Interface, this single `invoke()` method can handle them all without modification.

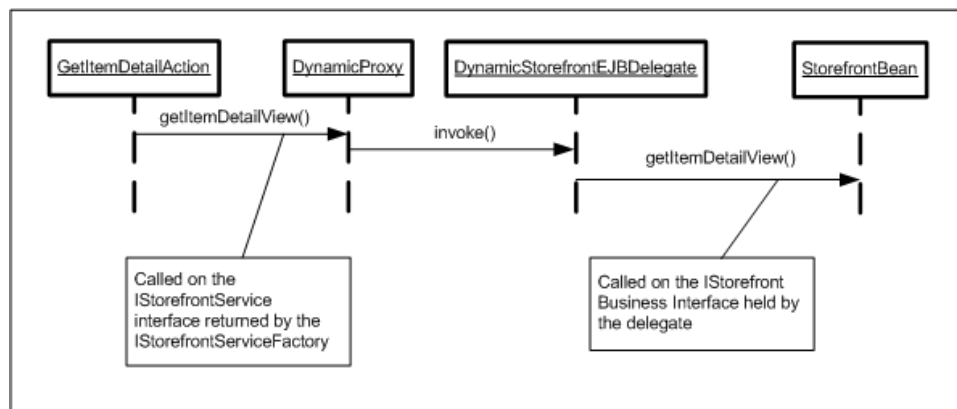


Figure 13-1. Sequence diagram for retrieving item detail through a dynamic proxy

In conclusion, not all web applications require the sophistication and complexity of an EJB container. The well-documented benefits of using an EJB server versus the added development and management complications must be seriously evaluated for each project. Don't just assume that you have to utilize EJB for the model portion of a Struts application. However, if you do decide that EJB is appropriate, as you've seen in this chapter, it doesn't have to impact the rest of the code. There are approaches that you can take to limit the coupling of your application to EJB, and make it much easier use.

14

Using Tiles

Up to this point, not much has been said on how to organize and assemble the content and layout of JSP pages for an application. In many ways, that is outside the scope of the topic of Struts. There are many excellent books that provide strategies for organizing web the content and layout of pages.

In the Storefront application, we have used two different approaches to assembling web pages. The first approach, sometimes referred to as a straight JSP-based approach, is probably the most familiar to web designers. The JSP pages contain presentation logic, along with HTML layout tags; there's no separation of the two. This approach is typically used for smaller, less complicated web applications.

The second approach utilizes the JSP include directive. It's used by developers in larger web applications or after developers realize how redundant the first approach can be. If you have spent any amount of time maintaining web applications, you know how frustrating it can be to update a site's look and feel. Using JSP's include directive allows for some reuse, which reduces total development and maintenance cost.

A third approach, which is introduced in this chapter, describes a far better way to reduce the amount of redundant code a web application contains, and at the same time, allows you to separate the content from the layout better than the previously mentioned approaches.

Understanding Templates

Traditional GUI toolkits like VisualWorks Smalltalk or Java Swing, all provide some type of a layout manager that dictates how content should be displayed within the frame

or window. With typical web sites, the layout can undergo many changes, both small and large, over its lifetime. Using a layout and layout manager, helps to encapsulate the physical areas of the pages within an application so that they can be altered with minimal impact to the rest of the application. Unfortunately, the JSP technology does not provide any direct support for layouts or layout managers. There is why the template-based approach was invented.

The concept of templates is not a new one. It has been around for many years in one form or another. Without templates, web designers must resort to a straight JSP-based approach or at best, make use of the JSP include directive. A hybrid of these two approaches is shown in Example 14-1.

Example 14-1. The `index.jsp` Page from the Storefront Application

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html:html>
  <head>
    <title><bean:message key="global.title"/></title>
    <html:base/>
    <script language="javascript" src="include/scripts.js"></script>
    <link rel="stylesheet" href="stylesheets/format_win_nav_main.css" type="text/css">
  </head>

  <body topmargin="0" leftmargin="0" bgcolor="#FFFFFF">

    <!-- Header Information -->
    <%@ include file="include/head.inc"%>

    <!-- Menu Bar -->
    <%@ include file="include/menubar.inc"%>

    <!-- Include the special offer -->
    <%@ include file="include/mainoffer.inc"%>

    <!-- Featured Items Header Row -->
    <table width="645" cellpadding="0" cellspacing="0" border="0">
      <tr>
        <td width="21">
          <html:img height="1" alt="" page="/images/spacer.gif" width="1" border="0"/>
        </td>
        <td width="534">
          <html:img page="/images/week_picks.gif" altKey="label.featuredproducts"/>
        </td>
        <td width="1" bgcolor="#9E9EFF">
          <html:img height="1" alt="" page="/images/spacer.gif" width="1" border="0"/>
        </td>
        <td width="1" bgcolor="#9E9EFF">
          <html:img height="1" alt="" page="/images/spacer.gif" width="1" border="0"/>
        </td>
        <td width="90" bgcolor="#9E9EFF" align="right">
```

```
<html:img height="1" alt="" page="/images/spacer.gif" width="90" border="0"/>
</td>
</tr>
<tr>
<td>
<html:img height="1" alt="" page="/images/spacer.gif" width="21" border="0"/>
</td>
<td colspan="4" bgcolor="#9E9EFF">
<html:img height="1" alt="" page="/images/spacer.gif" width="1" border="0"/>
</td>
</tr>
</table>

<html:img height="10" alt="" page="/images/spacer.gif" width="1" border="0"/><br>

<!-- Include the featured items -->
<% include file="include/featureditems.inc"%>

<!-- Include the copyright statement -->
<% include file="include/copyright.inc"%>
</body>
</html:html>
```

Example 14-1 shows the current `index.jsp` page for the Storefront application. Although the main page uses the JSP include directive, we can still see that the layout is mixed with content in the page. For example, notice that the JSP page in Example 14-1 specifies explicitly that the `head.inc` comes first, then the `menubar.inc`, next the `mainoffer.inc` include file, right down to the `copyright.inc` include at the bottom of the page. For every page that we wanted to have this particular layout, we would need to contain the same statements, in the same order. If a customer wanted the menu along the left side, instead of across the top for example, every page would have to be changed.

The Storefront application uses the JSP include mechanism rather than a straight JSP approach. Although the include mechanism is a step in the right direction, because it does reduce redundancy (imagine if we included the copyright content for every page), it's still less attractive than a template-based approach.

Static versus Dynamic Content

With JSP, there are two different ways to include content: static and dynamic. The `include` directive shown here:

```
<%@ include file="include/copyright.inc"%>
```

includes the source of the target page at translation/compile time. Therefore, it's not possible to include runtime content using the include directive. The JSP include directive treats a resource as a static object, the context of the resource is included literally in the page.

In direct contrast, the include action shown here:

```
<jsp:include page="include/copyright.inc"/>
```

handles the resource as a dynamic object. The request is sent to the resource and the result of the processing is included. Templates utilize a dynamic approach so that runtime expressions can be evaluated and included.

What is a Template?

A template is a JSP page that utilizes a JSP custom tag library in order to describe the layout of a page. The template acts as a definition for what the pages of an application will look like, without actually specifying the content. The content is inserted into the template page at runtime. All of the pages within the site may reuse the same template, but can also use different ones.

The purpose of a template is to get a consistent look and feel within an application. So it makes sense that most of the pages utilize the same template. However, it's not uncommon to have a different look and feel for a few pages within an application and therefore require a different template.

Example 14-2 illustrates a template for the Storefront application.

Example 14-2. A Basic Template for the Storefront Application

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<html:html>
<head>
<title><bean:message key="global.title"/></title>
<html:base/>
```

```
</head>
<body topmargin="0" leftmargin="0" bgcolor="#FFFFFF">

  <!-- Header Page Information -->
  <tiles:insert attribute="header" />

  <!-- Menu Bar -->
  <tiles:insert attribute="menubar"/>

  <!-- Main Body Information -->
  <tiles:insert attribute="body-content"/>

  <!-- Copyright Information -->
  <tiles:insert attribute="copyright"/>
</body>
</html:html>
```

Interestingly enough, there are not many new concepts being introduced in the template file in Example 14-2. The first thing that you should notice is that we are using Struts custom tag libraries. The fact that we are using a Tiles tag library as well as the HTML and Bean libraries shouldn't be too shocking; it's just like any other tag library. Don't worry too much about this for now. It's enough to understand that they can be used together just like any other set of tag libraries. We'll talk in detail about the Tiles tag library later in the chapter.

The rest of the page is a mixture of HTML layout tags, along with Tiles and Struts tags. You should notice that there's no content included, only `insert` tags where content will be inserted during runtime. You should already be familiar with the Struts tags shown here, so we won't say anything about them; they should be quite obvious in this example anyway. The `insert` tag is performing a role similar to that of the JSP include directive. It's basically saying that somewhere there's a variable called `header`, for instance. Pass the attribute value of "header" to the `insert` tag and insert the content that is produced right here. The same thing for the `menubar`, `body-content`, and `copyright` inserts as well. We'll explain very shortly how the "real" content is substituted in for these attributes during runtime.

Notice that this layout is very similar to the one shown in Example 14-1. The only difference is that instead of explicitly including the `mainoffer` and `featureditem` includes as Example 14-1 does, the template file includes a `body-content` section. By doing this, we can reuse the template for any page that has this generic format. All you have to do is to figure out how to supply the page-specific body-content and we can reuse this template over and over again. The benefit is that the layout of the various pages that use this template is controlled by this one file. If we need to modify the layout of the site, this is the only file that has to change. That's the real power of using a template-based approach.

The last piece of the puzzle is how the `header`, `menubar`, `body-content`, and `copyright` sections, are put together to form the output that the user eventually sees. The important point to remember is that the JSP page shown in 14-2 is the template. You

still require JSP pages that utilize the template and supply page-specific content that is used by the template. For example, rewriting the `index.jsp` page from Example 14-1 using the template from Example 14-2, it would look like the one in Example 14-3.

Example 14-3. The `index.jsp` for the Storefront Application using a Template

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:insert page="/layouts/storefrontDefaultLayout.jsp" flush="true">
  <tiles:put name="header" value="/common/header.jsp" />
  <tiles:put name="menubar" value="/common/menubar.jsp" />
  <tiles:put name="body-content" value="/index-body.jsp" />
  <tiles:put name="copyright" value="/common/copyright.jsp" />
</tiles:insert>
```

Let's explain what's going on in Example 14-3. The first thing to notice is that it's including the Tiles tag library at the top. Every page (also called a Tile, as we'll learn later) that needs to use the Tiles tag libraries, must include the library:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
```

There are two tags from the Tiles library being used in Example 14-3: `insert` and `put`. You saw the insert tag already in Example 14-2. It's performing slightly different function in Example 14-3, however. There are two attributes being supplied to the tag: `page` and `flush`. The `page` attribute is information to the tag that this JSP page is using a particular template (or Layout in the Tiles world). Layouts will be discussed later in the chapter. We are calling the template from Example 14-2 `storefrontDefaultLayout.jsp`. The flush attribute informs the controller to flush the page output stream before inserting.

The complete set of Tiles tags and their associated attributes will be discussed later.

The `put` tag in Example 14-3 answers a question that we asked in the previous section. That is, how does the page-specific content get supplied to the template? As you can see, the attributes for the `put` tag in this example are `name` and `value`. If you compare the values of the different name attributes, you'll see that they match up to the ones that the template file in Example 14-2 is expecting. When the `index.jsp` page from Example 14-3 is executed, the template file is processed and dynamically passed the `header.jsp`, `menubar.jsp`, `index-tile.jsp`, and `copyright.jsp` files from the `put` tags:

```
<tiles:insert page="/layouts/storefrontDefaultLayout.jsp" flush="true">
  <tiles:put name="header" value="/common/header.jsp" />
  <tiles:put name="menubar" value="/common/menubar.jsp" />
  <tiles:put name="body-content" value="/index-body.jsp" />
  <tiles:put name="copyright" value="/common/copyright.jsp" />
</tiles:insert>
```

At runtime, the values of the `put` tags are dynamically substituted into the template file and processed. The resulting output is what gets displayed to the client.

To wrap up the discussion of templates, we'll provide another page that uses the same template from example 14-2, but which supplies a different **body-content**. Example 14-4 shows the **itemdetail.jsp** page.

*Example 14-4. The **itemdetail.jsp** page for the Storefront Application*

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:insert page="../layouts/storefrontDefaultLayout.jsp" flush="true">
  <tiles:put name="header" value="../common/header.jsp" />
  <tiles:put name="menubar" value="../common/menubar.jsp" />
  <tiles:put name="body-content" value="../catalog/itemdetail-body.jsp" />
  <tiles:put name="copyright" value="../common/copyright.jsp" />
</tiles:insert>
```

The only difference between the **index.jsp** page in Example 14-3 and the **itemdetail.jsp** page in Example 14-4 is the different content supplied by the **body-content** **put** tag.

If you are still confused about the value of using templates, notice that the **index.jsp** and **itemdetail.jsp** pages in Example 14-3 and 14-4 respectively, do not specify anything about how the layout of the content. They both reference the **storefrontDefaultLayout.jsp** file, which has sole responsibility for displaying the content in a prescribed format. If we wanted to change the layout of the site, we would only have to modify the **storefrontDefaultLayout.jsp** file. This is the true power and benefit of templates.

Installing and Configuring Tiles

Before you can utilize the Tiles framework, you must ensure that it's installed and properly configured within your web container. The Tiles framework has no dependencies on a specific container. You will need to obtain the required files and ensure they are placed into their proper directories within the web application.

Downloading Tiles

The Tiles framework is now included with the Struts distribution. It had been previously included in the **contrib** folder, but it now part of the core distribution. You can also find the latest source and binary distribution, as well other useful information on the subject, at <http://www.lifl.fr/~dumoulin/tiles/index.html>.

Installing the Required JARs and Misc Files

The first required step is to install the necessary files. For Tiles, the following files must be located in the *WEB-INF/lib* directory:

- `tiles.jar`
- `commons-digester.jar`
- `commons-beanutils.jar`
- `commons-collections.jar`
- `commons-logging.jar`

Most of these binaries should already be present if you are using Struts. You will also need to install the Tiles tag library descriptor file `tiles.tld` in the *WEB-INF* directory for the application.

Don't add the `tiles.jar` to the classpath of your servlet container in an attempt to avoid placing it in the *WEB-INF/lib* directory of each individual web app. Doing so might cause `ClassNotFoundException` exceptions to be thrown.

You should go ahead and put the `tiles-config.dtd` in the *WEB-INF* directory. This DTD is used to validate Tiles definition files. We haven't introduced what these are yet, but we will.

Adding the Tiles Tag Library

Like any other JSP tag library, you must include it within the web application deployment descriptor to use it. Add the following `taglib` element to the *web.xml* file:

```
<taglib>
  <taglib-uri>/WEB-INF/tiles.tld</taglib-uri>
  <taglib-location>/WEB-INF/tiles.tld</taglib-location>
</taglib>
```

There should already be several `taglib` elements present if you are using any of the standard Struts tag libraries. For each page that needs to use the Tiles tag library, it needs to include the following line at the top of the page:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
```

Configuring Tiles to Work with Struts

The Tiles framework can be used with or without the Struts. Depending on how you use it, there are several options for configuring it for a web application. Since this book is about Struts, we'll focus on how to bind it to a Struts application.

With earlier versions of the Tiles framework, you had to configure a special `ActionServlet` called `ActionComponentServlet` in the `web.xml` file. You were also forced to configure a special `RequestProcessor` in the Struts `controller` element. This is no longer true and a `Tiles Plugin` is now available and will take care of all of the initialization.

To configure Tiles to work with a Struts application, you must add a `plug-in` element in the Struts configuration file for the `TilesPlugin`:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
  <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
  <set-property property="definitions-debug" value="2" />
  <set-property property="definitions-parser-details" value="2" />
  <set-property property="definitions-parser-validate" value="true" />
</plug-in>
```

Within the `plug-in` element, you can specify one or more `set-property` elements to pass additional parameters to the `Plugin` class. The `definitions-config` initialization parameter specifies the XML file or files containing Tile definitions. If multiple filenames are used, they must be comma-separated.

The `definitions-debug` specifies the debug level. The allowed values are:

- 0—No debug information is written out
- 1—Partial debug information is provided
- 2—Full debug information is provided

The default value is 0.

The `definitions-parser-details` parameter indicates the required level of debug information while the definition files are being parsed. This value is passed to the commons Digester. The allowed values are the same as those for the `definitions-debug`. The default value is 0.

The `definitions-parser-validate` parameter specifies whether the parser should validate the Tiles configuration file. The allowed values are true and false. The default is true.

There is an additional parameter not shown called `definitions-factory-class`. You can create a custom definitions factory and supply the class name here. The default is `org.apache.struts.tiles.xmlDefinition.I18NfactorySet`.

Overview of Tiles

The Tiles framework provides a templating mechanism that allows you to separate the responsibility of layout from those of content. Like the templates described earlier in the chapter, you have the ability to establish a layout and dynamically insert the contents of your pages into that layout at runtime. This offers a very powerful mechanism if you need to customize your site based on such things as internationalization, user preferences, or just look and feel changes that occur in every web application, sooner or later. The Tiles framework provides the following features:

- Template Capabilities
- Dynamic Page Construction and Loading
- Screen Definitions
- Support for Tile and Layout Reuse
- Support for Internationalization
- Support for Multi-Channels

There has been a Template tag library within the Struts framework for quite some time. These tags allow you to utilize a very basic templating approach to assembling your JSP pages in a web application. Although these tags are helpful in separating the content for a web application from its prescribed layout, the Tiles framework goes much further and actually provides a superset of the template tag library behavior, as well as many other features.

The Tiles framework was previously called Components, but was changed because the term is so overused. The Tiles documentation and source code still makes reference to the term Components in some places.

Cedric Dumoulin created the Tiles framework to extend the concept of templates and provide developers more flexibility and freedom when creating web applications built with JSP technology. The content for the web applications is still driven by JSP pages and JavaBeans that are stored in the request and session. However, the layout is specified within a separate JSP page, or as well see later in an external XML file.

What is a Tile?

A Tile is an area or region within a web page. The region can be the entire web page, or the page can be broken up into several regions. Figure 14-1 illustrates an example from the Storefront application.

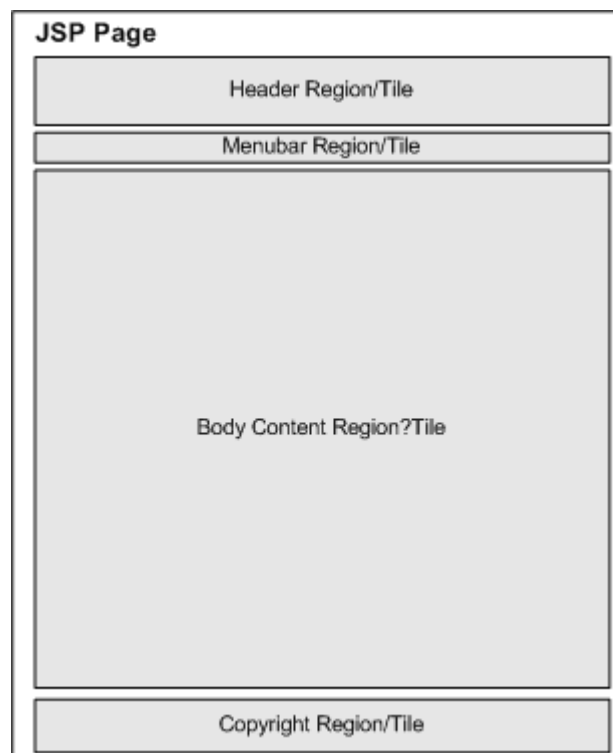


Figure 14-1. The Regions of the Storefront Application

A JSP page is typically made of several regions or Tiles. There's nothing really that special about the page, other than the fact that it's designed to be used with the Tiles framework and makes use of the Tiles tag libraries. The name Tile is more conceptual than it is physical. The point of a Tile is that it is a JSP component that is reusable. This is true for the layouts, as well as the JSP body content. Conceptually different from the way that most JSPs are created, tiles are designed to be reused within an application and possibly across different applications. Other than that, there's nothing that complicated about a Tile. In fact, most of the examples you have seen so far can be classified as tiles, including Examples 14-2, 14-3, and 14.4.

Using a Layout Tile

A layout from the Tiles world is what we have been referring to as a template. A layout serves the exact same purpose as a template, to assemble a group of tiles together in order

to specify the format of a page. The example in 14-2 is in fact, a Tiles layout. The syntax between Tiles and a template library, like the one included with Struts, is almost identical.

Tiles provides a superset of the functionality included with the tags as defined in Struts by David Geary. The Tiles framework takes the concept of templates even further by providing additional functionality.

Layouts are also considered tiles. Many pages, as well as other applications, can reuse layouts. It's very common to build a library of layouts that are used by many different projects. The Tiles framework comes with several pre-built layout tiles that you can reuse or modify as needed. The layouts that are included are:

- **Classic Layout**—Renders a header, left menu, body and footer
- **Columns Layout**—Renders a list of tiles in multi-columns. Each column renders its tiles vertically stacked
- **Center Layout**—Renders a header, left tile, right tile, body and footer
- **Menu Layout**—Renders a menu with links
- **Tabs Layout**—Renders several tiles in a tabs-like fashion
- **Vertical Box Layout**—Renders a list of tiles in a vertical column

Since one of the main goals of Tiles is reuse, you can reuse these layouts within your application, with little or no modifications necessary. You also have the freedom to customize the layouts anyway that you need.

Planning your Layout

It's very important that you plan your layout requirements ahead of time. Trying to decide how to your site is going to look, after it's already built, is definitely not the right approach. This decision is typically made by a human factors team, product management, or possibly by the web developers themselves. In any case, you need to develop the layout (or layouts if you application requires more than one), well ahead of any actual development.

You've already seen the default layout for the Storefront application in Example 14-2. This layout will be used throughout this chapter

Passing Parameters to the Layout

The layout tile that was shown in Example 14-2 is generic. It doesn't know anything about the `itemdetail.jsp` content for example, or any of the other pages for that

matter. This is intentional because it allows us to reuse this layout for many pages. Instead of hardcoding the content within the layout page, the content is supplied or “passed” as parameters to the layout page at runtime? Let’s look at the `signin` tile for the Storefront application. It’s shown in Example 14-5.

Example 14-5. The `signin.jsp` Tile for the Storefront Application

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:insert page="../layouts/storefrontDefaultLayout.jsp" flush="true">
  <tiles:put name="header" value="../common/header.jsp" />
  <tiles:put name="menubar" value="../common/menubar.jsp" />
  <tiles:put name="body-content" value="../security/signin-body.jsp" />
  <tiles:put name="copyright" value="../common/copyright.jsp" />
</tiles:insert>
```

The purpose of the `put` tag in Example 14-5 is to supply the layout tile, specified in the enclosing `insert` tag, with content. The values of the name attributes in Example 14-5, as well in the other Tiles like Example 14-3 and 14-4, must match ones that the layout tile is expecting.

The `insert` tag can optionally include an `ignore` attribute that will cause the tag to not write anything out when it can’t find a value for an expected attribute. By default, a runtime exception is thrown when an attribute has not been supplied.

The Tiles Tag Library

This section introduces the JSP custom tags that are used by the Tiles framework. Table 14-1 lists the tags available to the framework. The tags are very similar to the ones provided in any template-based framework, except that Tiles contains more functionality.

Table 14-1. The Tags Within the Tiles Tag Library

Name	Description
<code>insert</code>	Insert a tiles component.
<code>definition</code>	Create a Tile Component Definition.
<code>put</code>	Put an attribute into a Tile context.
<code>putList</code>	Declare a list that will be passed as an attribute.
<code>add</code>	Add an element to the surrounding list.
<code>get</code>	Gets the content from request scope that was put there by a <code>put</code> tag.

<code>getAsString</code>	Render the value of the specified tile/component/template attribute to the current JspWriter.
<code>useAttribute</code>	Use an attribute value inside a page.
<code>importAttribute</code>	Import a Tile's attribute in specified context.
<code>initComponentDefinitions</code>	Initialize a Tile definitions factory.

The insert Tag

The `insert` tag is responsible for inserting content in a page. In a layout tile, the `insert` tag prescribes where the content will go using attribute values. In a regular non-layout tile, the insert tag is used to retrieve a layout and allow content to be passed to the layout using `put` tags. Table 14-2 lists the attributes for the `insert` tag.

Table 14-2. The Attributes for the `insert` Tag

Attribute Name	Description
<code>template</code>	A string representing the URI of a tile or template. The <code>template</code> , <code>page</code> , and <code>component</code> attributes have exactly the same behavior.
<code>component</code>	A string representing the URI of a tile or template. The <code>template</code> , <code>page</code> , and <code>component</code> attributes have exactly the same behavior.
<code>page</code>	A string representing the URI of a tile or template. The <code>template</code> , <code>page</code> , and <code>component</code> attributes have exactly the same behavior.
<code>definition</code>	Name of the definition to insert. Definitions are defined in a centralized file. For now, only definition from factory can be inserted with this attribute. To insert a definition defined with the definition tag, use <code>beanName=""</code> .
<code>attribute</code>	Name of an attribute in the current tile/component context. Value of this attribute is passed to the <code>name</code> attribute.
<code>name</code>	Name of an entity to insert. Search is done in this order: <code>definition</code> , <code>attribute</code> , <code>page</code> .

<code>beanName</code>	<p>Name of the bean used as a value. Bean is retrieved from specified context, if any. Otherwise, method <code>pageContext.findAttribute()</code> is used. If the <code>beanProperty</code> attribute is also specified, retrieve value from the corresponding bean property.</p> <p>If the bean (or bean property value) is an instance of one of Attribute class (Direct, Instance, ...), insertion is done according to the class type. Otherwise, the <code>toString()</code> method is called on the bean, and returned String is passed to the name attribute.</p>
<code>beanProperty</code>	<p>The name of the bean property. If specified, the value is retrieved from this property.</p>
<code>beanScope</code>	<p>The scope into which the bean is searched. If not specified, method <code>pageContext.findAttribute()</code> is used. Scope can be any JSP scope, component, or template. In these two later cases, bean is searched in tile/component/template context.</p>
<code>flush</code>	<p>True or false. If true, current page output stream is flushed before tile insertion.</p>
<code>ignore</code>	<p>If this attribute is set to true, and the attribute specified by the name does not exist, simply return without writing anything. The default value is false, which will cause a runtime exception to be thrown.</p>
<code>role</code>	<p>If the user is in the specified role, the tag is taken into account; otherwise, the tag is skipped and the content is not written out.</p>
<code>controllerUrl</code>	<p>Url of a controller called immediately before page is inserted. Url usually denote a Struts action. Controller (action) is used to prepare data to be render by the inserted Tile. Only one of <code>controllerUrl</code> or <code>controllerClass</code> should be used.</p>
<code>controllerClass</code>	<p>Class type of a controller called immediately before page is inserted. Controller is used to prepare data to be rendered by the inserted Tile. Only one of <code>controllerUrl</code> or <code>controllerClass</code> should be used. The Class must implement or extend one of the following :</p> <ul style="list-style-type: none"><code>org.apache.struts.tiles.Controller</code><code>org.apache.struts.tiles.ControllerSupport</code><code>org.apache.struts.action.Action</code>.

There have been several examples of the `insert` tag earlier in the chapter.

The definition Tag

The definition tag is used to create a tile (template) definition as a bean. The newly created bean will be saved under the specified `id`, in the requested `scope`. The `definition` tag has the same syntax as the `insert` tag. The new definition can extend a definition described in the definition factory (XML file), and may overload any previously defined parameters. Table 14-3 lists the attributes supported by the `definition` tag.

Table 14-3. The Attributes for the `definition` Tag

Attribute Name	Description
<code>id</code>	Specifies the name under which the newly created definition bean will be saved. This attribute is required.
<code>scope</code>	Specifies the variable scope into which the newly defined bean will be created. If not specified, the bean will be created in page scope.
<code>template</code>	A string representing the URI of a tile/component/template (a JSP page).
<code>page</code>	URL of the template / component to insert. Same as "template".
<code>role</code>	Role to check before inserting this definition. If role is not defined for current user, definition is not inserted. Checking is done at insert time, not during definition process.
<code>extends</code>	Name of a parent definition that is used to initialize this new definition. Parent definition is searched in definitions factory.

The following fragment illustrates how to use the `definition` tag in a JSP page:

```
<tiles:definition
  id="storefront.default"
  page="/layouts/storefrontDefaultLayout.jsp"
  scope="request">
  <tiles:put name="header" value="/common/header.jsp" />
  <tiles:put name="menubar" value="/common/menubar.jsp" />
  <tiles:put name="copyright" value="/common/copyright.jsp" />
</tiles:definition>
```

A complete example is shown later in the *Declaring Definitions in a JSP Page* section of this chapter.

The put Tag

The `put` tag is used to pass attributes to a tile component. This tag can only be used inside the `insert` or `definition` tags. The value (or content) of the `put` tag is

specified using `value` attribute, or using the tag body. It is also possible to specify the type of the value :

- `String`— The content is literally translated
- `page` or `template`— Content is included from specified URL. Name is used as an URL
- `definition`— Content come from specified definition (from factory). Name is used as definition name

If the `type` is used, it is taken into account by the `get` or `insert` tags inside the inserted tile. If the `type` attribute is not specified, the content is 'untyped', unless it comes from a typed bean. Table 14-4 lists the attributes for the `put` tag.

Setting the `direct="true"` is equivalent to setting `type = "string"`.

Table 14-4. The Attributes for the `put` Tag

Attribute Name	Description
<code>name</code>	Name of the attribute.
<code>value</code>	The attribute value. Could be a String or an Object. Value can come from a direct assignement (<code>value="aValue"</code>) or from a bean. One of <code>value</code> , <code>content</code> , or <code>beanName</code> must be present.
<code>content</code>	Content that's put into tile scope. This attribute is equivalent to the <code>value</code> attribute. Attribute added for compatibility with JSP Template.
<code>direct</code>	Determines how the content is handled: true means content is printed <i>directly</i> ; false, which is the default, means content is <i>included</i> . This is another way to specify content type. If 'direct=true' content is "string", if 'direct=false', content is "page". Attribute added for compatibility with JSP Template.
<code>type</code>	Specify content type. Valid values are "string", "page", "template" or "definition".
<code>beanName</code>	Name of the bean used to retrieve the value from. The bean is retrieved from the specified context, if any. Otherwise, the method <code>pageContext.findAttribute()</code> is used. If <code>beanProperty</code> is specified, retrieve value from the corresponding bean property.
<code>beanProperty</code>	The property name in the bean. If specified, the value is retrieve from this property.

<code>beanScope</code>	The scope used to search for the bean. If not specified, the method <code>pageContext.findAttribute()</code> is used. Scope can be any JSP scope, 'tile', 'component', or 'template'. In these three later cases, bean is search in tile/component/template context.
<code>role</code>	If the user is in the specified role, the tag is taken into account; otherwise, the tag is skipped and the content is not written out.

The `putList` Tag

The `putList` tag creates a list that will be passed as an attribute to a tile. The List elements are added using the `add` tag. This tag can only be used inside the `insert` or `definition` tag. Table 14-5 lists the attributes for the `putList` tag.

Table 14-5. The Attributes for the `putList` Tag

Attribute Name	Description
<code>name</code>	Name of the List. This attribute is required.

The `add` Tag

The `add` tag adds an element to the surrounding list. This tag can only be used inside `putList` tag. The value can come from a direct assignement (value="aValue") or from a bean. One of `value` or `beanName` must be specified.

Table 14-6. The Attributes for the `add` Tag

Attribute Name	Description
<code>value</code>	The value to be added. Can be a String or Object.
<code>content</code>	The value to be added. Can be a String or Object. Synonym to value. Attribute added for compatibility with JSP Template.
<code>direct</code>	Determines how the content is handled: true means content is printed <i>directly</i> ; false, which is the default, means content is included.
<code>type</code>	Specify content type: <code>string</code> , <code>page</code> , <code>template</code> or <code>definition</code> . If the <code>type</code> attribute is not specified, content is 'untyped', unless it comes from a typed bean.
<code>beanName</code>	Name of the bean used to retrieve the value. Bean is retrieved from specified context, if any. Otherwise, the method <code>pageContext.findAttribute()</code> is used. If <code>beanProperty</code> is specified, retrieve the value from the corresponding bean property.

<code>beanProperty</code>	The bean property name. If specified, value is retrieve from this property.
<code>beanScope</code>	The scope used to search for the bean. If not specified, method <code>pageContext.findAttribute()</code> is used. The <code>scope</code> can be any JSP scope, 'component', or 'template'. In these two later cases, bean is search in tile/component/template context.
<code>role</code>	If the user is in the specified role, the tag is taken into account; otherwise, the tag is ignored (skipped). The role isn't taken into account if <code>add</code> tag is used in a definition.

The `get` Tag

The `get` tag retrieves content from the tile context and includes it in the page. Table 14-7 lists the attributes for the `get` tag.

Table 14-7. The Attributes for the `get` Tag

Attribute Name	Description
<code>name</code>	The name of the content to get from the tile scope. This attribute is required.
<code>ignore</code>	If this attribute is set to true and the attribute specified by the <code>name</code> does not exist, simply return without writing anything. The default value is false, which will cause a runtime exception to be thrown.
<code>flush</code>	True or false. If true, the current page output stream is flushed before insertion.
<code>role</code>	If the user is in the specified role, the tag is taken into account; otherwise, the tag is ignored.

The `getAsString` Tag

The `getAsString` tag retrieves the value of the specified tile attribute property and renders it to the current `JspWriter` as a `String`. The usual `toString()` conversion is applied on the value. If the named value is not found, a `JSPException` will be thrown. Table 14-8 lists the attributes for the `getAsString` tag.

Table 14-8. The Attributes for the `getAsString` Tag

Attribute Name	Description
<code>name</code>	The attribute name. This attribute is required.

<code>ignore</code>	If this attribute is set to true and the attribute specified by the name does not exist, simply return without writing anything. The default value is false, which will cause a runtime exception to be thrown.
<code>role</code>	If the user is in the specified role, the tag is taken into account; otherwise, the tag is ignored.

The `useAttribute` Tag

The `useAttribute` tag declares a Java variable and an attribute in the specified scope, using tile attribute value. The variable and attribute will have the name specified by `id`, or the original name if not specified. Table 14-9 lists the attributes for the `useAttribute` tag.

Table 14-9. The Attributes for the `useAttribute` Tag

Attribute Name	Description
<code>id</code>	Declared attribute and variable name.
<code>classname</code>	The Class of the declared variable.
<code>scope</code>	The scope of the declared attribute. Defaults to 'page'.
<code>name</code>	Tile's attribute name. This attribute is required.
<code>ignore</code>	If this attribute is set to true, and the attribute specified by the name does not exist, simply return without error. The default value is false, which will cause a runtime exception to be thrown.

The `importAttribute` Tag

The `importAttribute` tag imports the attribute from the tile to requested scope. The `name` and `scope` attributes are optional. If not specified, all tile attributes are imported in page scope. Once imported, an attribute can be used as any other beans from the JSP context. Table 14-10 lists the attributes for the `importAttribute` tag.

Table 14-10. The Attributes for the `importAttribute` Tag

Attribute Name	Description
<code>name</code>	Tile's attribute name. If not specified, all attributes are imported.
<code>scope</code>	Scope into which attribute is imported. Default to page.

`ignore`

If this attribute is set to true, and the attribute specified by the name does not exist, simply return without error. The default value is false, which will cause a runtime exception to be thrown.

The `initComponentDefinitions` Tag

The `initComponentDefinitions` tag initializes the definitions factory. Table 14-11 lists the attributes for the tag.

Table 14-11. Attributes for the `initComponentDefinitions` Tag

Attribute Name	Description
<code>file</code>	The definition file name. This attribute is required.
<code>classname</code>	If specified, the <code>classname</code> attribute of the factory to create and initialize.

Using Definitions

The tiles that we have shown so far add value to an application because they organize the layout of a page in a single resource, the layout JSP page. This can save development time and more importantly, the time it takes to change the layout for an application. However, there is a problem with the approach used in the Storefront application shown earlier. In each of the non-layout tiles, there is redundant code that specifies what content to use for the `header`, `menubar`, and `copyright` content. The same attributes are being passed in every page. This may not always be the case, but in general, these values will be constant throughout an application. For instance, the same copyright page is typically shown on every page.

It's redundant to have to specify these in every tile. It would be nice to declare what these attributes are in one place and the tiles could include just the page-specific attributes where needed. Fortunately, using Tile Definitions solves this. A definition allows you to statically specify the attributes that are used by a template. This allows you to specify only the page-specific attributes in your tile. Definitions add the following capabilities:

- Screen definitions
- Centralize declaration of page description
- Avoid repetitive declaration of nearly the same pages (by using definitions inheritance)
- Avoid creation of intermediate components used to pass parameters
- Specify the name of a definition as `forward` in the Struts configuration file

- Specify the name of a definition as component parameters
- Overload definition attributes
- Use a different copy of a component, depending on the local (i18n)
- Use a different copy of a component, depending on a key (This might be used to show different layouts depending on the client type)

Definitions can be declared in a JSP page or in an XML file. Regardless of which approach you choose, you should strive to keep the definitions in a central place. If you are planning on using a JSP page to specify your definitions, put all of them for an application in a single JSP page. Don't spread your definitions throughout your site. This will only make maintenance more difficult.

Declaring Definitions in a JSP Page

As was previously mentioned, there are two locations you can specify definitions: a JSP page or an XML file. We'll discuss the JSP page approach first.

Create a JSP page and declare all of your definitions in that file. For the Storefront application, we've created a file called `storefront-defs.jsp` and put the default definition in it as Example 14-6 shows.

Example 14-6. Declaring Tiles Definitions in a JSP Page

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:definition
  id="storefront.default"
  page="/layouts/storefrontDefaultLayout.jsp"
  scope="request">
  <tiles:put name="header" value="/common/header.jsp" />
  <tiles:put name="menubar" value="/common/menubar.jsp" />
  <tiles:put name="copyright" value="/common/copyright.jsp" />
</tiles:definition>
```

The syntax for the definition tag was described earlier in the *The Tiles Tag Library* section of this chapter.

The definition in Example 14-6 is using the same layout tile that was used earlier. The common files that were spread through the various tiles, are now located in the definition file. This makes changing the value for these much easier. For instance, if we wanted to specify a different copyright page, the only place to change that would be in the definition file. We would not have to modify every JSP page.

The `definition` tag syntax looks very similar to the syntax for the insert tags shown earlier. We just need to provide an id attribute and switch the `path` attribute to the `page`

attribute. Also, the default scope for the `definition` tag is `page`. It was set to request scope here to give it a little broader scope.

To take advantage of the definition, the tile components need to be able to access the definition. Since we have given the definition request scope, we need a way to include the definition in the various JSP pages, since it will only exist for the lifetime of a request. Fortunately, we already know how to include a JSP page in another page using the JSP include directive. Example 14-7 shows what the `index.jsp` page looks like using the JSP definition file.

Example 14-7. The `signin.jsp` Page Using a Tile Definition

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<%@include file="../common/storefront-defs.jsp" %>

<tiles:insert beanName="storefront.default" beanScope="request">
  <tiles:put name="body-content" value="../security/signin-body.jsp"/>
</tiles:insert>
```

With this approach, the tile components only have to insert the page-specific content. Compare example 14-7 to Example 14-5. Notice that the `signin.jsp` using the definition only needs to provide the page-specific content, the `sign-body.jsp` file.

Declaring Definitions in a Configuration File

You also have the option of declaring definitions in a centralized XML file. Whether you use the JSP page or the XML alternative really depends on your requirements. With the XML approach, you won't need to use the include directive shown earlier.

Creating a Definition Configuration File

To use this approach, create an XML file that follows the syntax of the `tiles-config.dtd`. The definitions XML file should be placed in the `WEB-INF` directory as with the other application meta-information. The DTD should also be placed in the `WEB-INF` directory. Example 14-8 shows an example of a definition XML file.

Example 14-8. The Storefront XML Definitions File

```
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config.dtd">

<tiles-definitions>
  <definition name="storefront.default" path="/layouts/storefrontDefaultLayout.jsp">
    <put name="header" value="/common/header.jsp" />
    <put name="menubar" value="/common/menubar.jsp" />
    <put name="copyright" value="/common/copyright.jsp" />
  </definition>
</tiles-definitions>
```

You'll notice that there's not much difference between the definition format specified in the JSP file in Example 14-6 and the one from the XML file in Example 14-8. The XML file uses a slightly different syntax, but is very similar.

The two formats are just similar enough to cause problems. Notice that in the JSP definition, you use the `put` tag:

```
<tiles:put
  name="body-content"
  value="../security/signin-body.jsp" />
```

but in the XML definition, you use a `put` element:

```
<put
  name="header"
  value="/common/header.jsp" />
```

Make sure that you don't get these two confused, as this can be an annoying bug to track down.

Each definition should have a unique name. JSP tags and pages use the name to retrieve the definition. It can't be used as a URL however. It's only a logical name for the definition.

Extending Tile Definitions

One of the most powerful features of Tile definitions is the ability to create new definitions by extending existing ones. All attributes and properties of the parent definition are inherited and you have the ability to override any attribute or property. To extend a definition, just add the `extends` attribute. Example 14-9 shows an example of a definition named `storefront.customer` extending the `storefront.default` definition.

Example 14-9. Definitions can extend other Definitions

```
<tiles-definitions>
  <definition name="storefront.default" path="/layouts/storefrontDefaultLayout.jsp">
    <put name="header" value="/common/header.jsp" />
    <put name="menubar" value="/common/menubar.jsp" />
    <put name="copyright" value="/common/copyright.jsp" />
  </definition>
</tiles-definitions>

<tiles-definitions>
  <definition name="storefront.custom" extends="storefront.default">
    <put name="copyright" value="/common/new-copyright.jsp" />
  </definition>
</tiles-definitions>
```

In Example 14-9, all of the attributes in the `storefront.default` definition are inherited. However, the `storefront.customer` definition overrides the value for the

copyright attribute with an alternate copyright page. This is a very powerful feature. If you have multiple child definitions all extending a root definition, changing a value in the root definition changes it for all children. You could change the layout in the root definition and have it changed for all child definitions.

Using Definitions as Forwards in Struts

Tiles definitions can also be used as Struts forwards, instead of actual URLs. To use definitions in this manner, you first create the definitions:

```
<tiles-definitions>
  <definition name="storefront.default"
    path="/layouts/storefrontDefaultLayout.jsp">
    <put name="header" value="/common/header.jsp" />
    <put name="menubar" value="/common/menubar.jsp" />
    <put name="copyright" value="/common/copyright.jsp" />
  </definition>

  <definition name="storefront.superuser.main" extends="storefront.default">
    <put name="header" value="/common/super_header.jsp" />
    <put name="menubar" value="/common/super_menubar.jsp" />
    <put name="copyright" value="/common/copyright.jsp" />
  </definition>
</tiles-definitions>
```

The fragment shows two definitions, the standard default definition and a second one that defines the layout for a “super” user. A “super” user might be someone that frequents the site and places many orders. The “super” user might be given more options on the menu bar to facilitate faster ordering.

In the Struts configuration file, we need to define the forwards that use these definitions:

```
<global-forwards>
  <forward name="Super_Success" path="storefront.superuser.main" />
</global-forwards>
```

You can then use the `Super_Success` forward to send the user to the `storefront.superuser.main` definition just as you would for any other forward.

Internationalization Support with Tiles

Although the Struts framework provides certain I18N capabilities and that functionality can be used with Tiles, Tiles also provides the ability to select a particular Tile based on a `Locale`. To support this feature in your application, you need to create a different Tiles definition file for each locale that you need to support. For example, if you needed to support a set of definitions for the US locale and a separate set for the German locale, you would need to create two separate definition files:

- *tiles-tutorial-defs_en.xml*
- *tiles-tutorial-defs_de.xml*

The suffix naming conventions follow the ones set by the `java.util.ResourceBundle`, which is also used by the resource bundle for Struts. When a request for a definition is made, the correct definition is determined by the included locale.

Similar to regular Java resource bundles, you should always provide a base definition that is used when no locale is provided, or when one is used that is not supported. The Tiles base definition file contains no language or country suffix appended to the name.

Once you have the locale-specific definition files created and placed in the *WEB-INF* directory, the only other necessary step is to ensure that a `Locale` is stored in the user's `HttpSession`. The Tiles framework depends on the same `Locale` instance that is used by Struts to determine which definition file to use. You will need to ensure that the Struts framework is storing the user's `Locale` in the session. This is accomplished by setting the `locale` attribute to "true" in the controller element. See *The Struts Configuration DTD* in Chapter 4 for more information on the `controller` element.

That's all there is to it. Note that you should still rely on the Struts resource bundle for locale-sensitive resources like text, messages, and images. The I18N support in Tiles should be used for differences in layout based on the locale. Struts and Tiles work very well together to provide complete I18N support.

16

Addressing Performance

Achieving good performance with an application is absolutely critical to its acceptance by the user community. Not many users are willing to appreciate a well thought-out design and excellent programming standards, if the application is sluggish and slow to respond. Even the most useful and desired application will quickly become the bane of a user community if it's slow to respond to the user's requests.

For web-based applications, organizations must test and measure how fast the various web components are, the number of simultaneous hits the web site can handle, and how scalable the software and hardware resources are. They must also understand how the performance of the application will be affected during peak loads.

This chapter explores the performance implications of using the Struts framework and its associated technologies to build web applications, and how certain design and programming decisions will affect the overall performance of the application. A distinction will be made between performance, load, and stress testing, and what steps are necessary to carry out each.

What is Good Performance?

It's likely that every developer has had the unfortunate opportunity to build a slow application. Obviously, developers don't set out to create a slow application, and there probably isn't a user group asking, "Could you please make the application run very slow for us?" Too often, bad performance isn't discovered until the application is finished and installed into a production environment. So why does it happen?

The simple truth is because not enough attention is given to performance matters during design and construction. This is not to say that performance should be the primary focus during design or even construction. If you focus on performance too exclusively or too soon, it may affect the design and code negatively. On the other hand, if you wait too long, you may find yourself with upset users complaining about the poor performance and you'll be wondering what went wrong.

You may have heard of the axiom "Test soon, Test often". This is a good principle to follow to help ensure that you are not surprised at the end of construction with a poor performing application. The sooner you can detect a performance problem, the more likely that you'll get a chance to fix it before the application goes into production. There's a saying, "Don't leave any broken windows". This statement means that when you detect a problem, fix it and don't let it linger. It's like a building that gets a broken window that's not immediately fixed. If people are lead to believe that one broken window is acceptable, they will eventually decide that it's all right to have many broken windows. Before long, the building is in shambles, and the tenants have all moved out. If you find obvious performance problems during early tests, fix them.

So how do you measure the performance of a web application? What's considered acceptable or too slow? The answers to these questions are strictly related to the non-functional* requirements of the application. There are very tangible and quantitative measurements that can be taken. These measurements help to determine if the application is not able to meet the minimum requirements set out in the non-functional requirements.

The problem is that each application is different and therefore has different non-functional requirements. One application might have to have an average response time of 3.0 seconds and support 50 concurrent users, while another might have to support 500 simultaneous users. Unlike functional testing, where it's easy to see when the application fails to meet the design specifications, performance testing is a little more nebulous.

According to Alberto Savoia, who is the Directory of Software Research at Sun's Microsystems Laboratories, there are four behavioral laws that make web page performance critical to an organization's success.

1. The Law of Stickiness—This law says that web users are sticky, but not loyal. If they find a web site that serves their needs, they tend to continue to use the site. If the web site begins to respond slow and cause the users to wait, they will move to another site that fulfills their same needs. The point is to strive to keep the performance of the application strong in order to keep the users coming back

* The non-functional requirements are part of the analysis work that should be done for any non-trivial application. These requirements describe the broader issues of the application, such as availability, disaster recovery, package dependencies, and almost always, performance criteria.

2. The Law of User Perspective—This law states that you should always measure the performance of your application from a user's point of view, not from yours. The point here is that your environment may have a 100MB network with an otherwise light load on it, but the user may be using a modem with a much smaller bandwidth capability. When testing, always know what the user's environment and network capability will be and test accordingly.
3. The Law of Responsibility—This law states that the user doesn't care what or who is at fault for poor web site performance, they will always blame the application. The problem might be their ISP or other non-application issues, but most users will not be able to isolate the problem to that level and will instead blame the application or site. You must be aware of all of factors that impact the performance of your application.
4. The Law of Expectations—This last law states that user's satisfaction is based on their expectations, and their expectation is set by their personal experiences with other similar web sites. When measuring the performance of your application, don't rely just on arbitrarily numbers to indicate what's slow or fast, compare your results with those of your competitors.

These simple common-sense laws explain the human behavior aspects of web site performance. In general however, slow is slow and fast is fast. There are generalities that can be made across applications and business domains. However, before we discuss how to detect whether performance problems exist with an application, a distinction needs to be made between the types of performance testing that should be conducted.

Performance versus Load Testing

There are many different types of software testing; functional, unit, integration, white box, black box, regression, and so on. Performance and load testing are among the most important, but usually get the least amount of attention. There are generally two reasons why this might be. The first reason is that developers typically wait to the very end of the development cycle to start testing the performance of the application. The end of the cycle is when you have the least amount of time due to deadlines. This reduces the amount of time available to spend on such an important part of application. It is true however, that it's not always practical to conduct performance testing during every phase. Early phases tend to focus on the architecturally significant pieces and there may not be enough of the application built to test the performance of. You should however, gather some preliminary performance measurements as early as possible, so as not to be surprised.

Another reason that performance and load testing don't get much attention is it's honestly hard to do. It's true that there are many tools on the market, both free and commercial, but it's quite another thing to utilize these tools to detect problems. These tools must be able to simulate many simultaneous users of a system. But this involves understanding

what virtual uses are, different threading models and how this affects performance and load, and then you must be able to look at the results and determine if they are acceptable or not. All of this can be very overwhelming to the average developer. This is part of what keeps developers from conducting the tests; they just don't understand the necessary steps or how or where to get started. Many organizations house a separate team that is solely responsible for performance testing. Smaller groups might utilize the qual

Although performance, load, and stress testing are related, they are not the same thing and they are not carried out in exactly the same manner. Performance testing involves executing the functional behavior of the application and essentially timing how long it takes for each result to complete. The amount of time that a single task takes to finish is known as its response time. If you execute the method many times and then take the average, this is its average response time. The average response time for the signin action in the storefront application for example, is roughly 250 milliseconds. This is for a single user using the system. You should always conduct the initial performance testing using a single user, in order to get a baseline. If there are performance bottlenecks for a single user of the system, you can bet these problems will have an impact when multiple users start logging in. In general, the faster the response time, the faster the application is. This end-to-end time can also be thought of as the transaction time for the function being tested.

Based on the response time, you are able to come up with a rough throughput time. Throughput defines the number of transactions that can occur in a set amount of time. The theoretical throughput that is calculated based on a single user, will probably differ with real loads. Due to multi-processing and other hardware and software features, applications can achieve a higher throughput by adding more hardware and software resources. This enables the application to process more transactions per time period, which increases the throughput numbers.

Load testing is analogous to volume testing. This type of testing is performed to see how the application will react to a higher user load on the system. During this type of testing, you can adjust the hardware and software configurations to determine which configuration gives the best throughput for a given user load. Load testing is usually the harder of the two to conduct, because you are constantly going back and forth adjusting configuration systems to see what gives you the higher throughput. No application can sustain an infinite user load. The idea is to try and maximize the number of concurrent users with an acceptable average response time.

Throughput is usually measured in transactions per second (tps), but it can also be measured per minute, per hour, and so on. Armed with response times and throughput numbers, you're then able to make intelligent decisions about how the application should be configured to ensure the best performance and scalability for the users. It's also a good idea to share these numbers with the network engineers so they'll understand how much network bandwidth the application might require.

Stress testing is the next logical extension. Stress testing is essentially load testing using peak loads. When conducting stress testing, you are really trying to stress the application to its limits to see how it reacts, how efficient the memory is used, and what other types of problems will surface.

What are the Benefits of Stress Testing?

Stressing your application under a heavy simulated load offers many benefits. Chiefly among them are:

- Identify bottlenecks in the application under a large user load before they occur in the production environment.
- Control risks and costs by being able to predict scalability and performance limits.
- Increase uptime and availability of the application through proper resource planning.
- Avoid missing go-live dates due to unexpected performance and scalability problems.

Both performance and stress testing should be performed on an application to get the complete picture. They can point to parts of the application that might become a bottleneck, under normal loads and especially as the number of users climbs.

Performance and Stress Testing Tools

There is an abundant supply of performance and stress testing tools available on the market today. Some are very inexpensive and others cost more than you can probably imagine. While commercial testing products tend to offer more plentiful features, there's really no correlation between cost and quality when it comes to these types of tools. For most applications, you don't need to "spend an arm and a leg" on a tool, when there are free or cheaper ones available that will do the job. The best advice is to start out with one of the free or inexpensive ones and see if it meets the project's needs. If not, then take a look at one of the more commercial products to see what it has to offer.

Many of the performance testing tools include functionality to test for a single user and also for multiples ones. For that reason, Table 19-1 lists several of the available performance and stress testing tools.

Table 19-1. Available Performance and Stress Testing Tools

Company	Product	URL
Apache Group	JMeter	http://jakarta.apache.org/jmeter
Mercury Interactive	LoadRunner	http://www-svca.mercuryinteractive.com
Rational	SiteLoad	http://www.rational.com

RadView	WebLoad	http://www.radview.com
Empirix	e-Test Suite	http://www.empirix.com
Seque Software Inc.	SilkPerformer	http://www.seque.com
Microsoft	WAS	http://homer.rte.microsoft.com

Testing the Storefront Application

The Storefront application is a fictitious shopping cart application that you might see on the Internet or might have actually built before. A normal application of this type would be connecting to a database with tens of thousands of records, possibly even hundreds of thousands.

By default, the Storefront application uses a debug implementation and doesn't connect to a database. This was done so you didn't have to have a database installed just run the example application.

There's no real point in going through the entire exercise of testing the Storefront application; the numbers wouldn't mean anything anyway. It would however be helpful to show how to get started and what normal steps must be taken in order to get performance numbers out of an application. The following are general steps that should be followed:

1. Understand the performance goals.
2. Establish the performance baselines for the application.
3. Run tests to collect performance data.
4. Analyze the data to detect where the problems are
5. Make necessary changes to increase performance.
6. Repeat steps 3 through 5 as necessary to reach the performance goals.

For this exercise, we are going to use a scaled-down version of the *Mercury Interactive Load Runner* product. The product is called *Astra Load Test* and is a light version of the Load Runner product. It is a commercial product that is feature rich. You can download a demo version that will support up to 10 virtual users. It's available for download at <http://www-svca.mercuryinteractive.com/products/downloads.html>.

Understand the Performance Goals

Before you begin any testing scenario, it's important to understand what the performance goals of an application are. The performance goals are normally specified in the non-functional requirements for an application and are usually specified using the following units:

- Average Transaction Response Time
- Transactions Per Second (tps)
- Hits/Second

It's not absolutely critical that you know what the performance numbers need to be before starting to test the application, but it can help to have a set of expectations for the application. Sooner or later, someone is going to ask you, "So how does the application perform?" To be able to say, "it's good" or "it stinks", you're going to have to make it relative to some set of goals.

Establishing a Performance Baseline

Once you're ready to get started testing, the first thing that you should do is establish a baseline. A baseline is a snapshot of your application's performance before anything has been done to it. It's always a good idea to get a performance baseline before you start changing code to better the performance. Otherwise, how do you know if you've made it better, or in fact worse?

Taking a Baseline

Most performance testing tools allow you to record the interaction sequence between a browser and the web application. Although most tools allow you to also manually create the testing scripts, using the automatic recording aspects of the tool is very convenient and alleviates many of the tedious chores. Figure 19-1 illustrates the record screen of the Astra LoadTest software.

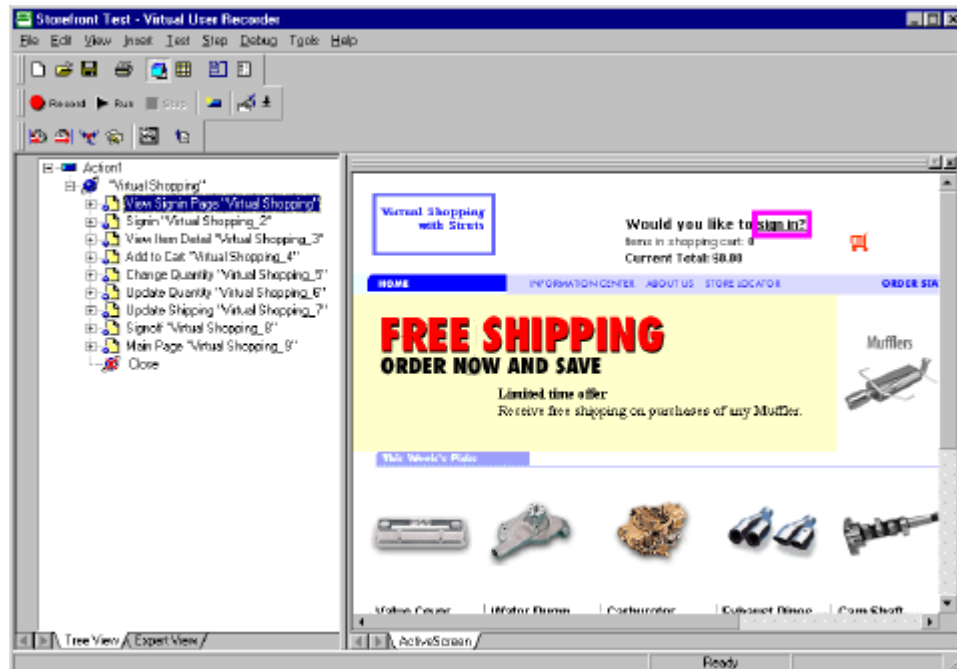


Figure 19-1. The Recording Screen of the Astra LoadTest Application

With Astra, as with most other web testing tools, each interaction with the web application can be recorded as a separate transaction. In Figure 19-1, each element in the tree view on the left-hand side of the screen represents a separate transaction that can be played back and the performance metrics recorded for it.

Once you start recording, all interaction between the client and the server is recorded, including request parameters and headers. You can then play back this recording and modify the different parameters like the number of users executing the recording.

Once you have the necessary test scripts, you can establish the baseline. The baseline measurement is normally taken with a single user using the application. Depending on whether you are conducting performance tests or are concentrating more on load testing, the number of virtual users can vary. It's typically a good idea to start off with one user and scale upwards. If the application is slow with a single user, it's a good chance that it'll be slow with multiple users. Figure 19-2 shows the testing script from Figure 19-1 running against the Storefront application with a single user.

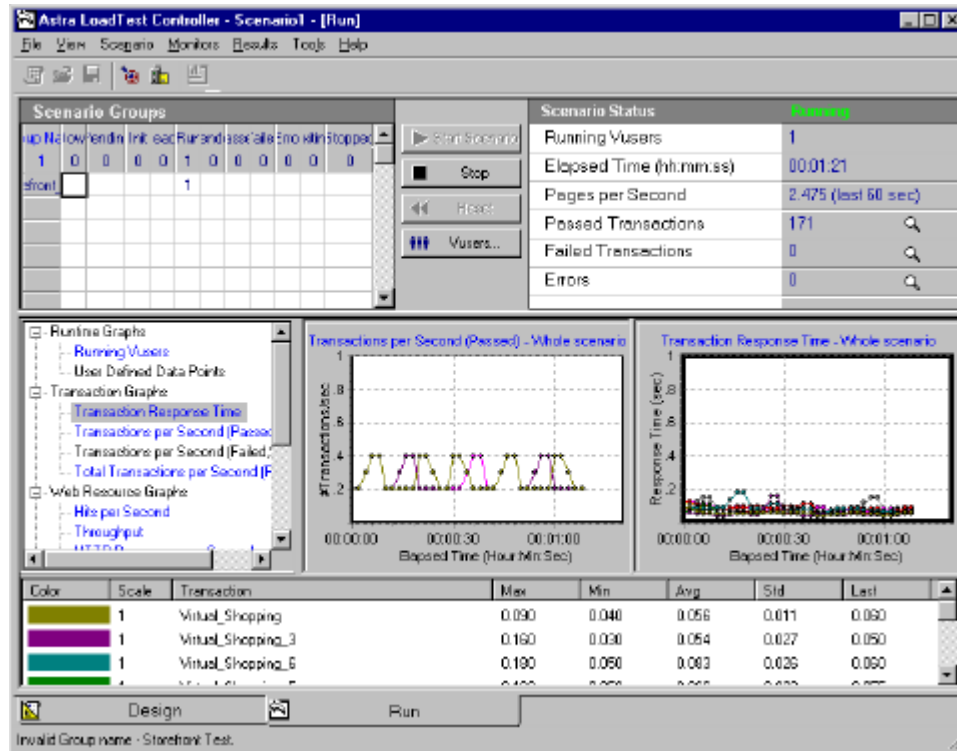


Figure 19-2. Testing the Storefront Application with a Single User

Once the testing scenario is complete, the software gives you a summary report of the performance of your application. The baseline report for the Storefront application with a single is shown in Figure 19-3.

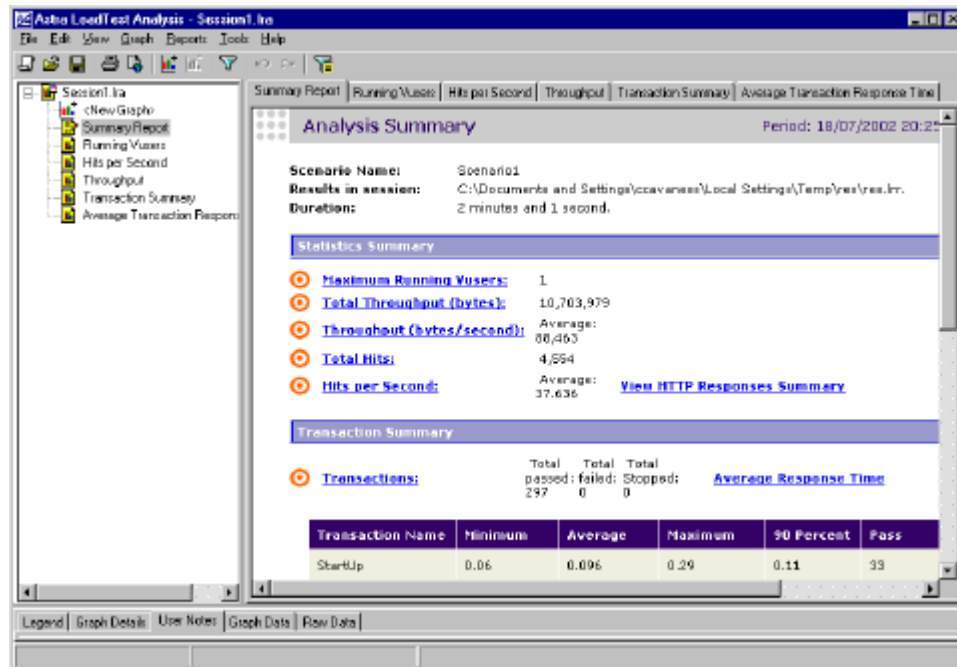


Figure 19-3. The Summary Report for the Storefront Application

Once you have the baseline numbers and it's determined that the performance needs to increase, you can start to modify the application to increase the performance. Unfortunately, it's never that easy. You have to know where the problems are in order to determine where to focus on in the application. There's not much point in speeding up the application in the places are already fast. You need to use the tools to help determine where the bottlenecks are.

Finding the Trouble Areas

Sometimes you get lucky and you find the performance problems quickly. Other times, you need to utilize different tools to locate and isolate the areas that are causing the problems. This is where profile tools can help out.

Profiling your Application

Profiling your application is somewhat different than the performance tests that we've been discussing. The tools and the tactics are somewhat different. Although performance tools might be able to tell you that a particular web request took the longest, it can't tell which Java method took up the most time. This is the purpose of profilers.

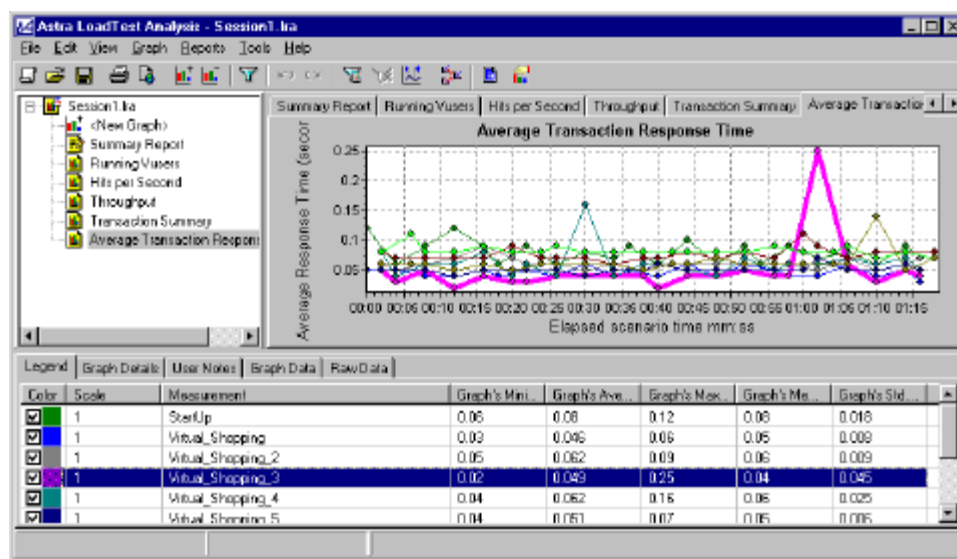
Table 19-2 lists several profiling tools that can be useful in locating troubled areas of the application.

Table 19-2. Commercially Available Profiling Tools

Company	Product	URL
Rational	Quantify	http://www.rational.com
Intuitive Systems	OptimizeIt	http://www.borland.com/optimizeit
Sitraka Software	JProbe	http://www.sitraka.com

Profiling an application is pretty much like debugging one. You see where the application spends most of its time, how many calls are made to a specific function, how many objects are created, how much memory is used, and so on. You start from a high level and work your way down to the methods that are causing the performance or scalability problem. Once you fix the problem areas, you run the tests again. This is repeated until all of the problem areas are resolved, or until you have to ship the product.

The performance tools can also help you determine where the problem areas are. In Figure 19-4 for instance, we see that the average transaction response time for one action seems much higher than the rest.

*Figure 19-4. Higher Average Response Times May Indicate a Problem*

The numbers shown here are really extraordinary good. The worse average response time for the Storefront application is 0.25 seconds. Most developers would kill to have a problem like that! That's because we aren't doing anything "real" in the application. It's all an example application that doesn't really connect to a backend system or thousands of records to sift through. This brings up a good point however. Just because one transaction is slower than the rest doesn't mean that it's slow. It just might be slow relative to other really fast transactions. Don't spend time speeding up something from very fast to

super fast. Concentrate on the parts of the application that are truly slow. 0.25 seconds is very fast and if this were a “real” application, we would ship it immediately.

The operation that shows the worst response time in Figure 19-4 is the *view item detail* action. With Astra, we can break the transaction down even further to see what’s going on. Figure 19-5 breaks the *view item detail* page down into its constituent parts.

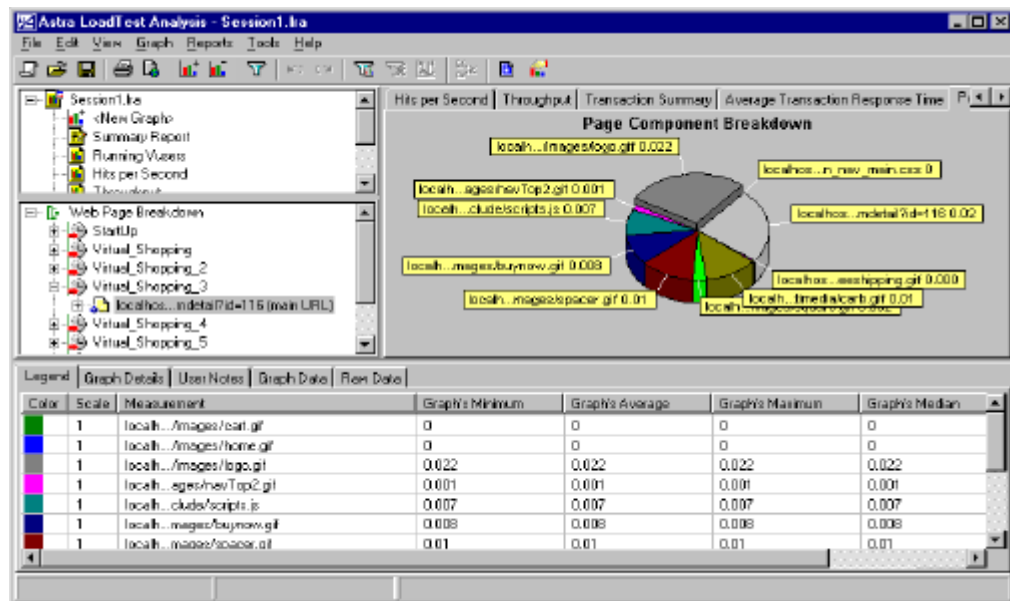


Figure 19-5. The Item Detail Page Broken Down into its Parts

From this, we might try looking at the download times of the various pages to see how the *ItemDetail* page stacks up.

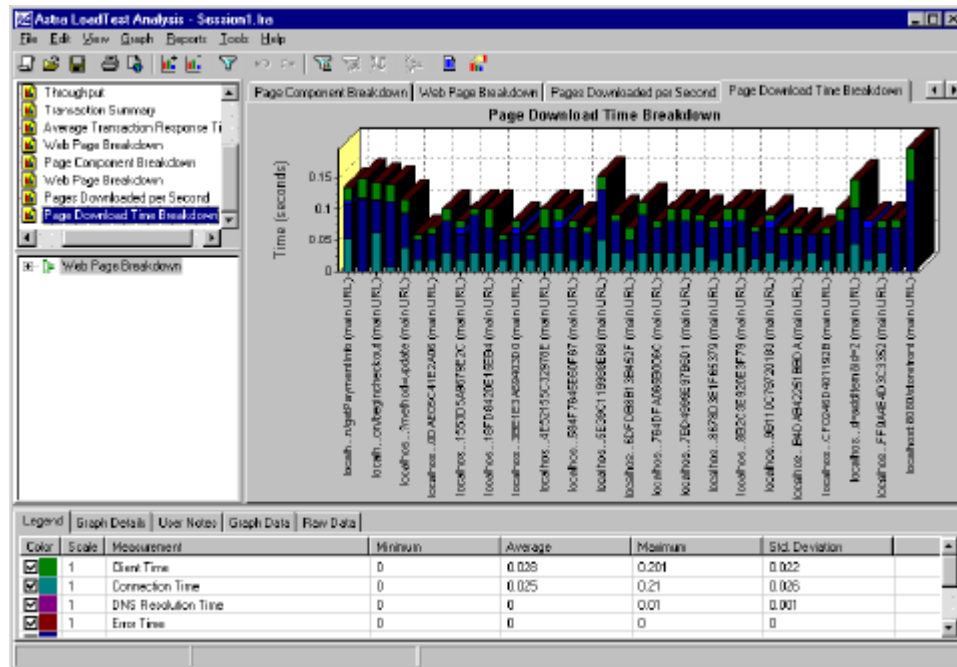


Figure 19-6. Page Download Times of the Storefront Application

The act of tracking down performance problems is like being a detective. You have to look behind every corner and leave no stone unturned.

Make the Necessary Changes to Software and Hardware

Once you think you've found the problem and made the necessary changes, it's back to the testing scripts once again. You keep doing this over and over until all of the performance issues are solved or until you run out of time, which usually comes first.

There's usually not just one problem that you can fix and then everything is fixed. It's a constant back and forth process that might go on forever, except for the fact that you've eventually got to ship the product. Because time is usually fleeting, it's imperative that you concentrate on the biggest performance areas and only tackle the smaller ones if you have leftover time.

[\[Note: Reviewers, I got stuck with this section and decided to ask for you help. Do you see value in discussing the following topics? If you have more that you would to see me discuss, just add them to the list and I'll try to cover them. Chuck\]](#)

Performance and Scalability Gotcha's

This section will present several well-known issues that may affect performance and scalability for your Struts applications. This section is not meant to be exhaustive, but to single out a few of the more serious concerns.

Bad Architectural Decisions

Too Many Components on a Page

Request Scope versus Session

Using the `synchronized` Keyword

Synchronization is used to control the access of multiple threads to a shared resource. There are many situations where synchronization makes sense and is absolutely necessary to keep multiple threads from interfering with one another, leading to significant errors. Struts applications are inherently multi-threaded and one might think that certain parts of the web application should be synchronized. However, using the `synchronized` keyword inside of your Struts applications can cause some significant performance problems and reduce the overall scalability of the application

All of us have heard at one time or another that servlets should not contain any instance variables. This is because there may only be a single instance of a servlet running, with multiple client threads executing the same instance concurrently. If you store the state of one client thread in an instance variable, and a different client thread comes along at the same time, it may overwrite the previous threads state information. This is true for Struts `Action` classes and session-scoped `ActionForms` as well. You must be sure to code in a thread-safe manner throughout the application. This means to design and code your application to allow for multiple client threads to run concurrently throughout the application without interfering with one another. If you need to control access to a shared resource, try to use a pool of resources instead of synchronizing on a single object. Also, keep in mind that the `HttpSession` is not synchronized. If you have multiple threads reading and writing to objects in the session, you may experience severe problems that are very difficult to track down. It's up to the programmer to protected shared resources stored in the user session.

Using `java.util.Vector` and `java.util.Hashtable`

You must also be careful which Java classes you use throughout your Struts applications. Especially, when it comes to selecting which collection class to use. The `java.util.Vector` and `java.util.Hashtable` classes for example, are

synchronized internally. If you are using `Vector` or `Hashtable` within your Struts applications, this may have the same effect as if you used the `synchronized` keyword explicitly.

You should avoid using the synchronized version of these classes, unless you are absolutely sure that you need to. Instead of using `Vector` for example, you can use `java.util.ArrayList`. Instead of `Hashtable`, use the `java.util.HashMap` class. Both of these classes provide similar functionality as their counterparts, but without the synchronization overhead.

Using too many Custom Tags

Improperly Tuning the JVM

Too many Remote Calls

Too many Graphics

Dealing with Large ResultSets

18

Logging in a Struts Application

As dedicated and knowledgeable Java programmers, we always want to believe that the software that we create will stand up to the utmost scrutiny. However, as human beings, we are constrained somewhat and all have flaws in that respect. Therefore, it's never a good practice to believe that your software contains no defects; it happens, so accept it. The important thing is to use whatever means available to reduce the defects to an acceptable amount, or eliminate them entirely. To help locate any defects that are present in your software, it's beneficial to generate log messages that tell you what the application is doing.

Logging is important for other reasons as well. Security and auditing might depend on logging to provide information to the system administrators about what the authorized, and more importantly unauthorized, users of the application are doing. By providing real-time information about potential security attacks on the application, logging can give a much-needed edge to the system administrators and allow for quicker reaction to attacks.

This chapter takes a close examination of how the use of logging in your Struts applications can help identify defects before the application gets into production. And if your software is already being used in production, how logging can help identify problems with the software much faster and arrive at a solution in a shorter amount of time.

Logging in a Web Application

The importance of logging has been evident to experienced developers for many years. Arguably, logging can be as important a part of your framework as exception handling or even security, both of which may rely on logging functionality to help carry out their

responsibilities. Without logging in an application, maintenance can become a nightmare for the developers. And as we all know, there's not a single "real" application that doesn't go through a maintenance cycle periodically.

But the question remains, "What about logging in a web application? Is it necessary and as important as logging in other types of applications?" Seeing that web applications can sometimes be smaller and less complex than their enterprise application counterparts, you might think that logging is less important. However, with non-trivial web applications, this is not the case and logging is just as critical there as it is in an enterprise application.

System versus Application Logging

Log messages can be arbitrarily broken down into two categories, system and application messages. System messages have to do more with the internal operation of the application, rather than something specific to a user or data. An example of a system message may be that the application is unable to send an email because the SMTP host is not responding. On the other hand, an application log message might be that user “Jane Doe” tried to submit a purchase order that was above her company’s credit limit. The system message in this case may be logged with a priority of “error”, whereas the application message might only get a priority of “info”.

The importance of this is that we can have our logging environment set up so that error messages generate an email or a pager message to the system administrators for immediate attention, while the messages with a priority of info go into a file for later auditing. There are different types and categories of log messages and they are typically used for different purposes across organizations. Although many applications may log messages with the priority of “error”, what’s an error to one organization may just be a warning to another. There’s not a great deal of consistency across organizations and may never be. This is because organizations have different priorities and what’s critical to one may not necessarily be critical to another.

For the sake of this chapter, we will generalize the discussion of system versus application messages. Because of the differing views of what’s considered an error and what isn’t, there’s no way to specify what’s an error, warning, or just general information for your particular application. That’s a decision that you, your development team, and your product management group will have to make. We’ll keep our discussion at a higher level and not focus on these issues.

Using the Servlet Container for Logging

The servlet specification requires that every servlet container allow developers to log events to the container log file. Although the location of the log file is container dependant, the manner in which these events are logged is dictated by the specification and is portable across web containers.

The `javax.servlet.ServletContext` class contains two methods that can be used for logging messages to the container’s log:

```
| public void log( String msg );
```

```
| public void log( String msg, Throwable throwable); |
```

You can use either of these methods by obtaining the `ServletContext` and passing the appropriate arguments. Example 18-1 illustrates how this can be done using a Struts Action.

Example 18-1. The `LoginAction` using the `ServletContext` to log messages

```
public class LoginAction extends StorefrontBaseAction {

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

        // Get the user's login name and password. They should have already
        // validated by the ActionForm.
        String email = ((LoginForm)form).getEmail();
        String password = ((LoginForm)form).getPassword();

        // Obtain the ServletContext
        ServletContext context = getServlet().getServletContext();

        // Log which user is trying to enter the site
        context.log( "Login email: " + email );

        // Login through the security service
        IStorefrontService serviceImpl = getStorefrontService();
        UserView userView = serviceImpl.authenticate(email, password);

        // Log the UserView for auditing purposes
        context.log( userView.toString() );

        UserContainer existingContainer = null;
        HttpSession session = request.getSession(false);
        if ( session != null ){
            existingContainer = getUserContainer(request);
            session.invalidate();
        }else{
            existingContainer = new UserContainer();
        }

        // Create a new session for the user
        session = request.getSession(true);
        existingContainer.setUserView(userView);
        session.setAttribute( IConstants.USER_CONTAINER_KEY, existingContainer );

        return mapping.findForward( IConstants.SUCCESS_KEY );
    }
}
```

The `LoginAction` in example 18-1 is a very simple example of sending log messages to the container's log file. It calls the `log()` method and passes a literal string message

that will be written to the log. As was mentioned earlier, the name and location of the log is dependent on which web container being used.

Some web containers may assign a separate log file for each web application, while others may use just a single log file. If only one log file is being used, messages from different web applications will end up in the same log file and are usually prefixed with the web application name.

Using Filters

Filters are a new feature of the Java Servlet specification starting with version 2.3. Servlet filters allow you to inspect and/or transform the content of the HTTP request and response objects. Because of the manner in which filters are invoked by the servlet container, they are able to operate on dynamic as well as static content.

Struts 1.1 supports both the 2.2 and 2.3 Servlet Specifications. If you are not using a 2.3 compliant container, you will not be able to take advantage of Servlet filters.

Using filters, servlet developers are able to perform the following tasks:

- Access a web resource before a request to it is invoked.
- Process a request for a resource before it is invoked.
- Modify the request headers and data by wrapping the request with a customized version of the request object.
- Modify the response headers and data by wrapping the response with a customized version of the response object.
- Intercept a method call on a resource after it has been performed.
- Perform actions on a servlet or group of servlets, by one or more filters in a specified order.

Based on the tasks required by servlet developers, the servlet specification describes several possible uses for filters:

- Authentication filters
- Logging and auditing filters
- Image conversion filters
- Data compression filters
- Encryption filters

- Tokenizing filters
- Filters that trigger resource access events
- XSLT filters that transform XML content
- MIME-type chain filters
- Caching filters

All of these possible uses for filters are rather interesting; however for this chapter we are interested in the logging and auditing usage of filters. Using filters, it's possible to log messages using any data that is contained in the request and response objects. Since the filter is coupled tightly to the servlet container, you will likely still need logging functionality elsewhere in your application. Using filters for logging will generally not be enough for the entire application. However, it is a perfect place for auditing or tracking a user's actions through the system.

If you need to get an idea of which parts of the web site your users are visiting the most, or where certain user groups are going most often, filters might be an ideal solution for this. It might even be able to give you information about specific data the users are viewing most often. For example, let's assume that you have an online catalog application and you are interested in knowing which catalogs the users are browsing most often and which ones are not getting much attention. Since the request and response objects contain this information, you could easily track and store this information into a database for further analysis.

There are three basic steps to creating a filter:

1. Create a Java class that implements the `javax.servlet.Filter` interface and that contains a no-argument constructor.
2. Declare the filter in the web application deployment descriptor using the `filter` element.
3. Package the filter class along with the rest of the web application resources.

Creating the Filter Class

The first step in creating a servlet filter is to either create a new Java class or use an existing one and have it implement the `javax.servlet.Filter` interface. Remember that Java classes can implement multiple interfaces, so you don't necessarily have to create a new class. However, the class will eventually need to be loaded by the web container, therefore it shouldn't be one that is only installed on a backend system like an EJB container.

The `Filter` interface has three methods that must be implemented by your class:

```
public void init(FilterConfig filterConfig) throws ServletException;
```

```
public void doFilter(ServletRequest request,
                    ServletResponse response,
                    FilterChain chain )
    throws IOException, ServletException;

public void destroy();
```

The web container calls the `init()` method when it's ready to put the filter into service. You can initialize any needed resources in this method. For example, this would be a good place to initialize a logging service or acquire a database connection.

The `destroy()` method is the opposite of the `init()` method. The web container calls this method when it's taking the filter out of service. You should clean up any open resources that the filter may be using like database connections, inside this method.

Finally, the web container calls the `doFilter()` method every time a request is received and the container determines that the filter instance should be notified. This is the place that you should perform whatever functionality the filter is designed to perform.

Example 18-2 shows an example filter class that could be used to log to the servlet log file or to initialize a third-party logging service.

Example 18-2. A Servlet Filter Example

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletRequest;
import javax.servlet.ServletContext;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
/**
 * An example servlet filter class
 */
public class LoggingFilter implements Filter{

    public final static String LOG_FILE_PARAM = "log_file_name";
    private FilterConfig filterConfig = null;
    private ServletContext servletContext = null;

    public LoggingFilter() {
        super();
    }

    public void init( FilterConfig config ) throws ServletException {
        // Initialize any neccessary resource here
        this.filterConfig = config;
        this.servletContext = config.getServletContext();

        // You can get access to initialization parameters from web.xml
        // although this example doesn't really use it
    }
}
```

```
String logFileName = config.getInitParameter( LOG_FILE_PARAM );

// You can log messages to the servlet log like this
log( "Logging to file " + logFileName );

// Maybe initialize a third-party logging framework like log4j
}

public void doFilter( ServletRequest request,
                    ServletResponse response,
                    FilterChain filterChain)
    throws IOException, ServletException {

    // Log a message here using the request data
    log( "doFilter called on LoggingFilter" );

    // All request and response headers are available to the filter
    log( "Request received from " + request.getRemoteHost() );

    // Call the next filter in the chain
    filterChain.doFilter( request, response );
}

public void destroy(){
    // Remove any resources to the logging framework here
    log( "LoggingFilter destroyed" );
}

protected void log( String message ) {
    getServletContext().log("LoggingFilter: " + message );
}

protected ServletContext getServletContext(){
    return this.servletContext;
}
}
```

Just as you must be careful with multiple threads in Java servlets, you must be careful not to violate any thread safety practices with filters. The servlet container may send concurrent threads to a single instance of a filter class and you must ensure that you don't do anything to cause problems between the threads. In other words, there should be no client-specific data stored in instance variables. Local variables are fine, just as they are in Java servlets, because they are stored on the stack rather than the heap.

Declaring the Filter in the Deployment Descriptor

The second step in creating a servlet filter is to configure the proper elements in the deployment descriptor for the web application. As you learned from chapter 4, the name of the deployment descriptor for a web application is called *web.xml*.

The first step in setting up the filter declaration in the web application's deployment descriptor is to create the actual `filter` elements. Chapter 4 describes the filter element in detail, but the following deployment descriptor fragment illustrates again how this would look using the `LoggingFilter` class from the previous section:

```
<filter>
  <filter-name>MyLoggingFilter</filter-name>
  <filter-class>LoggingFilter</filter-class>
  <init-param>
    <param-name>log_file_name</param-name>
    <param-value>log.out</param-value>
  </init-param>
</filter>
```

You can also optionally specify initialization parameters, icons, a description, and a display label. See Chapter 4 for more details on the attributes of the `filter` element.

Once the filter element is added, you then need to add a `filter-mapping` element that will associate or link the specified filter to a servlet or static resource in the web application. Filters can be applied to a single servlet, or groups of servlets and static content, using two distinct mapping approaches. The following deployment descriptor fragment illustrates a filter mapping to a single servlet called `MyExampleServlet`:

```
<filter-mapping>
  <filter-name>MyLoggingFilter</filter-name>
  <servlet-name>MyExampleServlet</servlet-name>
</filter-mapping>
```

Every time the web container receives a request for the `MyExampleServlet` resource, the `doFilter()` method in the `LoggerFilter` class will be invoked. The following XML fragment illustrates how the example filter can be mapped to all requests sent to the web application:

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

The filter mapping in this case will map all requests to the `MyLoggingFilter` because every request URI will match the `"/"` URL pattern.

Packaging the Filter

The final, and likely the easiest step, is to package the filter class with the rest of the resources of the web application. As with any other Java resource that is part of a web application, the filter class must be bundled with the WAR file and able to be loaded by the web application's class loader. In most cases, the filter class should be placed in the `WEB-INF/classes` directory for the web application. Filter classes may also be inserted into a JAR file and placed in the `WEB-INF/lib` directory.

Using Event Listeners

Web application event listeners are Java classes that implement one or more of the servlet event listener interfaces. Event listeners support event notifications for changes in state in the `ServletContext` and `HttpSession` objects. Event listeners that are bound to the `ServletContext` support changes at the application scope, while those that are bound to the `HttpSession` objects are notified for state changes at the session level.

Multiple listeners can be setup for each event type and the servlet developer may offer a preference regarding the notification order for the listeners based on event type.

Tables 18-1 and 18-2 list the event types and event listener interfaces available to the servlet developer.

Table 18-1. ServletContext Application Event and Listener Interfaces

Event Type	Description	Listener Interface
Life Cycle	The <code>ServletContext</code> is about to service the first request or is about to be shut down by the servlet container.	<code>ServletContextListener</code>
Attributes	Attributes on the <code>ServletContext</code> have been added, removed, or replaced.	<code>ServletContext</code> <code>AttributesListener</code>

Table 18-2. HttpSession Application Events and Listener Interfaces

Event Type	Description	Listener Interface
Life Cycle	An <code>HttpSession</code> object has been created, invalidated, or timed out.	<code>HttpSessionListener</code>
Attributes	Attributes have been added, removed, or replaced on an <code>HttpSession</code> object.	<code>HttpSession</code> <code>AttributesListener</code>

The steps for creating an event listener are very similar to that of creating filters. There are three primary steps to perform.

1. Create a Java class that implements the event listener interface that you are interested in receiving events for. The event listener class must contain a no-argument constructor.
2. Declare the event listener in the web application deployment descriptor using the `listener` element.
3. Package the event listener class along with the rest of the web application resources.

Creating the Event Listener class

Similar to when creating filters, the first step is to create the Java class that implements the appropriate listener interface. As an example, we'll create a Java class that implements the `javax.servlet.ServletContextListener` interface and will be responsible for initializing the logging service when the web application is started. This class is illustrated in Example 18-3.

Example 18-3. The Example of a `ServletContextListener`

```
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
/**
 * An example ServletContext Event Listener class that
 * initializes a logging service.
 */
public class LoggingListener implements ServletContextListener{
    private ServletContext context = null;

    public LoggingListener() {
        super();
    }

    /**
     * Called by the container before the first request is
     * processed. This is a good time to initialize
     * the logging service.
     */
    public void contextInitialized( ServletContextEvent event ){
        this.context = event.getServletContext();

        // Initialize the logging service here

        // Log a message that the listener has started
        log( "LoggingListener initialized" );
    }

    /**
     * Called by the container when the ServletContext is about
     * ready to be removed. This is a good time to clean up
     * any open resources.
     */
}
```

```
public void contextDestroyed( ServletContextEvent event ){
    // Clean up the logging service here

    // Log a message that the LoggingListener has been stopped
    log( "LoggingListener destroyed" );
}

/**
 * Log a message to the servlet context application log or
 * system out if the ServletContext is unavailable.
 */
protected void log( String message ) {
    if (context != null){
        context.log("LoggingListener: " + message );
    }else{
        System.out.println("LoggingListener: " + message);
    }
}
```

The event listener class in example 18-3 contains two methods that are invoked by the web container:

```
public void contextInitialized( ServletContextEvent event );
public void contextDestroyed( ServletContextEvent event );
```

The web container will call the `contextInitialized()` method before the first request is processed. You should initialize any needed resources in this method. For example, this is an ideal location to initialize a logging service. The `contextDestroyed()` method is called when the web application is taken out of service. This is where any open resources that the listener class might be using should be closed.

Since there can be multiple event listener classes for the same event, let's use another `ServletContext` event listener class to make our example more realistic. Example 18-4 shows the `DBConnectionPoolListener` class. Both the `LoggingListener` and the `DBConnectionPoolListener` will receive event notifications when the `ServletContext` is initialized and destroyed.

Example 18-4. The `DBConnectionPoolListener` Class

```
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
/**
 * An example ServletContext Event Listener class that
 * initializes the database connection pooling.
 */
public class DBConnectionPoolListener implements ServletContextListener{

    /**
     * A no-argument Constructor
     */
    public DBConnectionPoolListener() {
```

```
        super();
    }

    /**
     * Called by the container before the first request is
     * processed. This is a good time to initialize
     * the connection pooling service.
     */
    public void contextInitialized( ServletContextEvent event ){
        // Initialize the connection pooling here
    }

    /**
     * Called by the container when the ServletContext is about
     * ready to be removed. This is a good time to clean up
     * any open resources.
     */
    public void contextDestroyed( ServletContextEvent event ){
        // Shut down the connection pooling and open database connections
    }
}
```

The servlet container will notify the listeners in the order that they are configured in the deployment descriptor based on the event type. Since both the `LoggingListener` and `DBConnectionPoolListener` are listening for the same type of application events, they will be notified in the order that they are in the descriptor.

Declaring the Event Listener in the Deployment Descriptor

The following web application deployment descriptor fragment shows you how to setup the event listener:

```
<web-app>
  <listener>
    <listener-class>LoggingListener</listener-class>
  </listener>

  <listener>
    <listener-class>DBConnectionPoolListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>ExampleServlet</servlet-name>
    <servlet-class>ExampleServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ExampleServlet</servlet-name>
    <url-pattern>/example</url-pattern>
  </servlet-mapping>
</web-app>
```

This means that the `LoggingListener` will be notified first, followed by the `DBConnectionPoolListener` instance. When the web application shuts down, the

listeners are notified in reverse order. The `HttpSession` event listeners are notified prior to listeners for the application context.

Packaging the Event Listener

The packaging of the event listener classes follows the same guidelines as described in the previous section for filters. That is, the classes must either be in the `classes` directory underneath the `WEB-INF` directory or installed in a JAR file located in the `lib` directory for the web application.

Jakarta Commons Logging

The Commons Logging package is an open source Logging library that allows developers to utilize a common logging API, while maintaining the freedom to utilize many different third-party logging implementations. The Commons Logging API insulates the application and protects it from becoming coupled to a specific logging implementation. The API provides a small set of Java classes and interfaces that an application imports and relies upon, but which has no implicit dependencies on any one logging product.

The Logging library allows developers to declaratively configure the logging implementation and the library will dynamically discover which implementation is being used. An application that utilizes the Commons Logging API does not have to be modified when the logging implementation is changed; this is the greatest benefit of such a package.

The Commons Logging package supports several different logging implementations out of the box:

- Log4J (<http://jakarta.apache.org/log4j>)
- JDK 1.4 Logging
- LogKit (<http://jakarta.apache.org/avalon/logkit>)
- SimpleLog (Writes Log Messages to `stdout` and `stderr`)
- NoOpLog (Log Messages are ignored)

The Commons Logging Package only includes the `SimpleLog` and `NoOpLog` implementations; it does not contain the other third-party logging implementations. You will need to download those separately.

Another powerful feature of the Commons Logging package is that it is completely extensible. If you are using a logging package that is not yet supported, you can create an adapter to implementation by extending the appropriate components, and your application can utilize the Commons Logging API.

Installing the Commons Logging Package

You can download the latest source and binary for the Commons Logging package at <http://jakarta.apache.org/commons/logging.html>. Struts 1.1 already includes the *commons-logging.jar*, which is the only required binary file. Unless you want the absolute latest from the nightly build, the version included with the Struts framework should suit your needs. The *commons-logging.jar* should be placed into the *WEB-INF/lib* directory for the web application.

You will also need to decide on a logging implementation. The Commons Logging package includes an implementation called *SimpleLog* that will write log messages to *stdout*. If you don't want to worry about getting log4j working and are not using Java 1.4, the *SimpleLog* implementation is a good choice to get things started.

Once you decide on an implementation, you must configure the implementation class so that the Commons logging factory component can discover it at application startup. There are many different ways to do this, but the easiest is to create a properties file called *commons-logging.properties* that contains the class name of the logging implementation. The properties file may contain configuration properties that will set configuration attributes on the instantiated *LogFactory*. The most important of these properties is the *org.apache.commons.logging.Log* key. This key is used to set the implementation class. The following illustrates how to setup the Commons Logging to use the *SimpleLog* implementation:

```
| org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog |
```

At runtime, the logging component will search for the *commons-logging.properties* file and attempt to instantiate the fully qualified class name found there. The class name specified must be available to the web application class loader. The properties file should be placed in the *WEB-INF/classes* directory. To switch to log4j, all you would need to do is switch the class name in the *commons-logging.properties* file like this:

```
| org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JCategoryLog |
```

You would still need to configure log4j for your environment, including creating a *log4j.properties* file. Each logging implementation may have different configuration requirements and are still required to be completed.

Using the Commons Logging API

Once the configuration steps are completed, your application is ready to utilize the Commons Logging API. The first step is to include the following import statements in each class or component that you wish to use the logging API within:

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

To get an instance of a log component, for which you can send log messages to, you need to use either of the `getLog()` factory methods on the `org.apache.commons.logging.LogFactory` class:

```
public static Log getLog(Class clazz);  
public static Log getLog(String name)
```

Both `getLog()` methods return an `Object` that implements the `org.apache.commons.logging.Log` interface. If we wanted to create a `Log` instance to be used within the `LoginAction` class for example, we could pass the class name to the `getLog()` method:

```
Log log = LogFactory.getLog( LoginAction.class );
```

The `Log` instance is then available to be used within the `LoginAction` class:

```
if (log.isInfoEnabled()){  
    // Log which user is trying to enter the site  
    log.info( "Login email: " + email );  
}
```

The `Log` interface implements the logging methods that you can use to send log messages to the intended destination. The most important of these are:

- `debug()`
- `error()`
- `fatal()`
- `info()`
- `trace()`
- `warn()`

Each one of these log methods has an overloaded version that takes a `Throwable`. There are also methods, which allow you to determine whether debug is enabled, or error is enabled, and so on. Checking to see if a particular logging level is enabled before attempting to log a message can increase performance of your application. For example, if you have this code fragment:

```
StringBuffer buf = new StringBuffer();  
buf.append( "Login Successful - " );  
buf.append( "Name: " );  
buf.append( userView.getFirstName() );  
buf.append( " " );  
buf.append( userView.getLastName() );  
buf.append( " - " );  
buf.append( "Email: " );  
buf.append( userView.getEmailAddress() );  
  
// Log the information for auditing purposes  
log.debug( buf.toString() );
```


It would increase the performance of the application if all of the append statements were not executed when the logging threshold was set to not log debug statements. The `isDebugEnabled()` method can be used for this. So instead, our logging fragment looks like:

```
if ( log.isDebugEnabled() ){
    StringBuffer buf = new StringBuffer();
    buf.append( "Login Successful - " );
    buf.append( "Name: " );
    buf.append( userView.getFirstName() );
    buf.append( " " );
    buf.append( userView.getLastName() );
    buf.append( " - " );
    buf.append( "Email: " );
    buf.append( userView.getEmailAddress() );

    // Log the UserView for auditing purposes
    log.debug( buf.toString() );
}
```

In this case, the application is not wasting time creating the `StringBuffer` only to have it not be used.

Struts Framework and the Commons Logging Package

The Struts framework does perform some limited internal logging. The framework uses the Commons Logging API as well. This means that whichever logging implementation you configure for your application, the Struts framework will also utilize it.

Other than for debugging purposes, there's no need for you to worry about the Struts logs. They are a great way for you to see what's going on inside Struts as it processes the requests. In most production environments however, the messages generated by the Struts framework should be disabled. The manner in which you disable the framework-specific log messages depends on which logging implementation you choose.

The rest of the chapter is devoted to one of the most popular logging implementations used by developers. Because it's also supported by the Commons Logging package, it makes an excellent choice for your Struts application logging needs.

Using the log4j Package

You might have heard or read about the log4j library from other sources, but in case you haven't we'll briefly discuss a little history about the library. Similar to Struts, log4j is an open source project that is part of the Jakarta set of projects. It's essentially a set of Java classes and interfaces that provide logging functionality to multiple types of output destinations. It has been around for several years and is constantly being refined and tuned for all types of Java development. In fact, log4j has been so successful that it has

been ported to several other very popular languages like C, C++, Python, and even one for .NET.

At the time of this writing, log4j has released version 1.2.5, which is the 22nd major public release of the library. The next major version 1.3 is in the works, but won't be released soon. Version 1.2 is backwards compatible with earlier versions, so if you are using 1.1.3, this material will still be relevant for you.

According to the creators of log4j, it was built with two central concepts in mind, speed and flexibility. One of the distinctive features of the logging framework is its notion of inheritance in Categories, or Loggers as they are now called. log4j supports a parent-child relationship among the configured Loggers in the environment. To provide an example, if we configured a logger for all classes in the `com.oreilly.struts` package and another logger for all classes in the `com.oreilly.struts.storefront` package, then the first logger would be the parent to the second. This hierarchical structure of loggers gives us the flexibility to control what log messages are written out based on things like our package structure.

You don't need to go this far if your requirements don't call for it. If you like, you can configure a single root logger for your entire environment. The nice thing is that you have the flexibility to configure log4j for your specific needs. You can also change this behavior whenever you like by simple editing an external configuration file, not the application source code.

The discussion of log4j could possibly fill a small book itself. We are going to assume that you are familiar with the basic concepts and only cover the essentials of how to integrate log4j with the Struts framework. If you haven't taken the time to become familiar with log4j, this is probably a good time to do so. Although you don't have to know everything there is to know about log4j, we will only be covering the basics from a peripheral vantage point. For a more detailed discussion, you can download or view the documentation at the Jakarta log4j website <http://jakarta.apache.org/log4j>.

Integrating log4j with Struts

To best ensure that the log4j libraries are available to your Struts applications, you should place the log4j JAR in the `WEB-INF/lib` directory for each web application that you deploy. Resist the temptation to put it inside the container wide `lib` directory, even if you have multiple web applications deployed using log4j. If you do attempt to install it at the container level, you will most likely encounter one or more `ClassNotFoundException` problems.

Based on the requirements set forth in the Servlet 2.3 Specification, the web container should automatically load all JARs in the `WEB-INF/lib` directory, including the log4j

library. After this initial step is complete, you are then free to use the log4j as the logging implementation for the Commons Logging package.

Keep in mind that the configuration of log4j is totally independent of the configuration of the logging implementation for the Commons Logging package. You still need to understand how to configure log4j (if that's the implementation you choose) and perform the necessary steps required by the log4j package.

What are Loggers?

The `org.apache.log4j.Logger` is the central class in the log4j toolkit. Other than configuration, most of the functionality is performed through this class. In earlier versions of the log4j project, the `org.apache.log4j.Category` class implemented this functionality. To promote backwards compatibility, the `Logger` extends the `Category` class. Although the methods in the `Category` class have not been deprecated, you should now always go through the `Logger` class itself. Eventually, the `Category` class will be removed from the library.

When using log4j with Struts, other than the configuration aspects of log4j, the majority of the log4j classes and interfaces will be encapsulated within the Commons Logging API.

Configuring log4j Appenders

With log4j, you can send log messages to multiple destinations. Log4j refers to a message destination as an *appender*. There are various appenders provided by the log4j framework out of the box. The following appenders are present in the framework for you to use:

- Console Appender
- File Appender
- Socket Appender
- Java Message Service (JMS) Appender
- NT Event Logger Appender
- Unix Syslog Appender
- Null Appender
- SMTP Appender
- Telnet Appender

- **Asynchronous Appender**

The log4j framework allows one or more appenders to be established for a logging environment. You even have the flexibility to send log messages to particular appenders, based on various conditions. The other nice feature about the appender architecture is if none of these default appenders are what your application requires, you have the freedom to create your own by extending the `org.apache.log4j.AppenderSkeleton` class.

Understanding the log4j Log Levels

A log message in log4j can be assigned one of five different levels or priorities. The levels provide a way to set a threshold for a particular Logger and filter out any log messages that don't reach the particular threshold that the Logger is configured for. The five logging levels are:

- **DEBUG**
- **INFO**
- **WARN**
- **ERROR**
- **FATAL**

An earlier version of log4j also defined the levels “OFF” and “ALL”, however these seem to have been deprecated and probably should be avoided completely. If you set the threshold to “DEBUG”, you'll get the same results as “ALL”, and “OFF” isn't really necessary because you can simply choose not to configure an appender for the environment, which will stop all logging from occurring.

There is a cascading effect that causes only levels equal to the threshold and higher to be logged. For example, if a threshold of WARN was configured, only messages with a level of WARN, ERROR, and FATAL would make it to an output destination.

Initializing log4j

There are many properties that can be configured for the log4j toolkit. In fact, log4j is so flexible that all of the configuration options can't be mentioned here. The best source of information is the log4j manual itself. You can find the manual online at <http://jakarta.apache.org/log4j/docs/documentation.htm> and it also available locally when you download log4j.

Because log4j doesn't make any assumptions about the environment in which it is running, it's necessary for you to configure the environment for your particular needs. In other words, there are no default appenders configured out of the box. There are also various ways in which you can initialize the configuration properties for the log4j environment. We will focus on two related, but distinct approaches here.

The first approach is to create a file called *log4j.properties* that contains the necessary configuration elements for your logging needs. This file must follow the guidelines of the `java.util.Properties` format. One of these guidelines is that each property is in the format `key=value`. Example 18-7 illustrates a very simple log4j configuration file that logs messages with a logging threshold of `INFO` and higher to the console using an `org.apache.log4j.ConsoleAppender`.

Example 18-7. An Example log4j.properties File

```
# A basic log4j configuration file that creates a single console appender

# Create a single console appender that logs INFO and higher
log4j.rootLogger=INFO, stdout

# Configure the stdout appender to go to the Console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

# Configure the stdout appender to use the PatternLayout
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
```

The configuration file shown in example 18-7 is a very simple example of setting up a single appender. The appender used in that example is the `ConsoleAppender`, which directs log messages to `System.out`.

This *log4j.properties* must be installed in the `WEB-INF/classes` directory and the log4j environment will be able to locate it and use it to configure the logging environment for the web application. If you have multiple web applications, you can have a separate *log4j.properties* file for each web application.

The log4j configuration file shown in example 18-7 only sent log messages to a single destination, the console. You can configure the log messages to go to multiple locations and also have certain messages go to certain locations based on the level of the message and other parameters. Example 18-8 shows another simple example of using two appenders.

Example 18-8. A log4j Configuration File using Two Appenders

```
# A sample log4j configuration file

# Create two appenders, one called stdout and the other called rolling
log4j.rootLogger=DEBUG, stdout, rolling
```

```
# Configure the stdout appender to go to the Console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

# Configure the stdout appender to use the PatternLayout
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

# Configure the rolling appender to be a RollingFileAppender
log4j.appender.rolling=org.apache.log4j.RollingFileAppender

# Configure the name of the log file for the rolling appender
log4j.appender.rolling.File=output.log

# Set up the maximum size of the rolling log file
log4j.appender.rolling.MaxFileSize=100KB

# Keep one backup file of the rolling appender
log4j.appender.rolling.MaxBackupIndex=1

# Configure the Layout pattern and conversion pattern for the rolling appender
log4j.appender.rolling.layout=org.apache.log4j.PatternLayout
log4j.appender.rolling.layout.ConversionPattern=%d{ABSOLUTE} - %p %c - %m%n
```

The log4j configuration file in example 18-8 creates an appender that logs messages to the console, just as in example 18-7. However, it also creates an appender that logs messages to a log file called *output.log*. Again, we won't try to explain all of the configuration settings for log4j, but you can learn more from the log4j website.

As we said earlier in this section, there are two approaches for setting up the configuration information for log4j that this section would focus on. You've seen the first approach in examples 18-7 and 18-8. These configuration files used a properties format to specify the configuration information. The second approach is to utilize an XML file for the same purpose. Example 18-9 illustrates an XML file that configures the same information as in example 18-7.

Example 18-9. A log4j Configuration File using an XML Format

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>

  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%5p [%t] (%F:%L) - %m%n"/>
    </layout>
  </appender>

  <root>
    <priority value="debug" />
    <appender-ref ref="STDOUT" />
  </root>
</log4j:configuration>
```

```
</root>
</log4j:configuration>
```

You normally place this XML file in the *WEB-INF/classes* directory, just as with the *log4j.properties*. However, in the case of the XML file, you must set the *log4j.configuration* system property equal to the name of the file. This is so the log4j environment knows which file to load. There's no default name when using the XML format. We didn't have to do this with the properties file because the name *log4j.properties* is part of log4j default initialization. If it locates this file anywhere in the classpath, it will use it to initialize the log4j environment.

There are various ways to set the *log4j.configuration* property. The various containers may also provide alternative methods as well. Tomcat for example, uses a variable called *CATALINA_OPTS* for version 4.0 that can be set to provide this information to the logging environment. In order to use this approach, you would set the value for Tomcat 4.0 like this:

```
| set CATALINA_OPTS=-Dlog4j.configuration=log4j.xml
```

You can then startup Tomcat and the log4j environment will be able to locate the XML configuration file. Other containers may provide alternate methods for setting the value, but you can always set the value on the Java command line as a system property. You will probably need to modify the containers startup script using this approach however:

```
| java -Dlog4j.configuration=log4j.xml
```

If the log4j environment is unable to find a valid configuration file, either properties-based or XML-based, you will see something similar to the following message as you first attempt to initialize the logging environment.

```
| log4j:WARN No appenders could be found for logger XXX.
| log4j:WARN Please initialize the log4j system properly.
```

The XXX in the previous fragment will actually show the name of the logger that no appenders were configured for.

This log4j warning is an indicator that it was unable to find the configuration file. Because the log4j framework doesn't make any assumption about the environment in which it is running, there's no default appender created for it. Therefore, this is the only message you'll get until it is able to locate the configuration file.

Specifying a Relative versus Absolute Path

When you use a system property to configure the log4j configuration file within a web application, the file is relative to the web application by default:

```
| java -Dlog4j.configuration=log4j.xml
```

This means that the log4j will search for a file called *log4j.xml* in the *WEB-INF/classes* directory for the web application. Most containers will use a separate class loader for

each web application and some containers may not allow the web applications to know about classes or JARs loaded by the container itself.

However, if you need to use an absolute path, you can specify one like this:

```
| java -Dlog4j.configuration=file:/c:/dev/env/log4j.xml |
```

Be careful when using an absolute path. Since the configuration file is not relative to a web application, all web applications will share the same one.

Generally speaking, specifying a relative path will be much more flexible than an absolute one. This is because you can't always guarantee the directory structure of all your target environments.

Synchronization Issues

There's one more issue that you should be aware of when logging to resources like file systems. Even though log4j is able to handle multiple client threads using the same appender because all threads are synchronized, if you have multiple appenders writing to the same resource or file, you will have unpredictable results. In other words, there is no synchronization between appenders, even within the same JVM.

This really has nothing to do with a deficiency in the log4j design; it's just a case of not being able to easily synchronize multiple writers to a resource. The easiest way to solve this problem is to ensure that if you have multiple appenders logging to the file system, or more importantly multiple web applications, you should not allow them to log to the same file or you will probably experience synchronization related issues.

Log File Rollover

In a normal production environment, log files can grow quite large if not managed properly. If the logging threshold is set to low or if the files are not purged from time to time, the files might grow without bounds.

It's a good idea to periodically backup the log files and then start again with an empty log file. For some production environments, this "rollover" period may be nightly; others may only need to perform this routine weekly. Unless you're able to shut down the application while you backup the log files, it's very cumbersome to backup the log files manually. Fortunately, log4j provides a type of appender that will automatically swap the log file out with a new one, while at the same time maintaining a backup of the old log file.

The `org.apache.log4j.DailyRollingFileAppender` class provides the ability to log to a file and rollover or backup the log file while the application is still running. You can also specify the rollover frequency and the date pattern that will be used for the backup names. Having this functionality available out of the box makes log4j invaluable to any application that needs to roll over log files on a user-defined frequency.

Using Commons Logging in JSP Pages

So far, you've seen how to use the Commons Logging API within Java components, but we haven't discussed how to use them in JSP pages. Although there are a number of ways to use the library within JSP pages, we'll cover the two easiest approaches here.

The first approach is to use the same three steps defined earlier, but only perform them in the JSP page itself.

1. Import the Commons `Log` and `LogFactory` classes.
2. Define and initialize a logger variable for the page.
3. Start logging.

Example 18-10 illustrates this in a basic JSP page.

Example 18-10. Using Commons Logging in a JSP Page

```
<%@ page import="org.apache.commons.logging.Log" %>
<%@ page import="org.apache.commons.logging.LogFactory" %>

<!-- Get a reference to the logger for this class -->
<% Log logger = LogFactory.getLog( this.getClass() ); %>

<% logger.debug( "This is a debug message from a jsp" ); %>

<html>
<head>
  <title>Using Commons Logging in a JSP page</title>
</head>

<body>
  <% logger.info( "This is another log message in the jsp" ); %>

  There should be two log messages in the log file.
</body>
</html>
```

You must have the Commons Logging environment configured properly for this to work, just as you do when using it in the Java classes. Any JSP page that is part of that web application will be able to utilize the logging utilities. Remember, since most containers use a different class loader for each web application, any JSP that is not part of the log4j configured web application may not be able to use the logging utilities.

Although example 18-10 shows just how easy it can be use the Commons Logging API in your JSP pages, there are a few issues with this approach. The most obvious one is that your JSP pages will contain Java code. As we've talked about during several chapters of this book, many developers see this as something that should be avoided. If you're one of those developers, fortunately there is another way.

The Jakarta Taglibs project contains a custom JSP tag library that is designed for log4j. It's called Log Taglib and is just one of many custom tags provided in the Jakarta Taglibs library. You can view information and download the tag library from the Jakarta Taglibs web site at <http://jakarta.apache.org/taglibs/doc/log-doc/intro.html>.

Just like any other custom tag library, you must properly configure it for your web application. This means putting the *log.tld* file into the *WEB-INF* directory and installing the log tag JAR file into the *WEB-INF/lib* directory. You will also need to add the appropriate *taglib* element to your web application's deployment descriptor similar to the manner in which the Struts tag libraries were added.

```
<taglib>
  <taglib-uri>/WEB-INF/log.tld</taglib-uri>
  <taglib-location>/WEB-INF/log.tld</taglib-location>
</taglib>
```

Once the log custom tag is installed and configured for you web application, you can now use it in your JSP pages as shown in example 18-11.

Example 18-11. An Example JSP Page using the Log Tag

```
<%@ taglib uri="/WEB-INF/log.tld" prefix="logger" %>

<logger:debug message="This is a debug message from a jsp using the Log tag" />

<html>
<head>
  <title>Using the Log Tag in a JSP page</title>
</head>

<body>

  <logger:info message="This is another message using the log4j tag" />

  There should be two log messages in the log4j log file.

</body>
</html>
```

Notice how the JSP page in example 18-11 contains no Java code. Although it doesn't seem like a huge difference between this example and the one in 18-15, with a much larger and more complex JSP page, this approach is much cleaner.

Another nice feature of the Log tag is that it gives you the ability to perform a dump of objects stored at the page, request, session, and application scopes. This is very useful when you are in the middle of debugging your web application. Example 18-12 illustrates how easy this is to do.

Example 18-12. Using the log tag library to dump information

```
<%@ taglib uri="/WEB-INF/log.tld" prefix="logger" %>

<html>
```

```
<head>
  <title>Using the Log Tag in a JSP page</title>
</head>

<body>

  <log:dump scope="page"/>
  <log:dump scope="request"/>
  <log:dump scope="session"/>
  <log:dump scope="application"/>

  The page,request, session, and application dumps should be in the log file.

</body>
</html>
```

Unfortunately, the Log tag doesn't work with the Commons Logging yet. It depends on the log4j implementation. However, it's still a valuable tag if you need to provide additional debugging in your JSP pages.

[\[Note: Reviewers, this next section will probably be taken out of the chapter during final edits due to the current size of this chapter. I left it in case anyone wanted to read it. chuck\]](#)

Creating an Email Appender

It was stated earlier that if none of the appenders included with log4j toolkit suite your needs, you are free to create your own. This section will provide a quick example of how to do exactly that.

Let's assume that your tasked with sending email messages to an administrator when log messages are produced that have a level of **ERROR** or **FATAL**. For our example, we'll assume that we're using a J2EE environment and already have a JMS Queue configured using Message-driven beans that handle the responsibilities of sending the email to a SMTP server.

Message-driven beans were added to the Enterprise JavaBean (EJB) Specification 2.0. Message-driven beans are an asynchronous JMS message consumer that are managed by the container and make for better JMS message scalability. Although many organizations are cautious to jump on the EJB 2.0 bandwagon this early, message-driven beans are an excellent addition to the EJB specification and can be a very beneficial addition to your applications.

We're going to use a JMS Queue here instead of a JMS Topic because we only want to send an email to a single recipient. Figure 18-1 illustrates the architecture for this example.

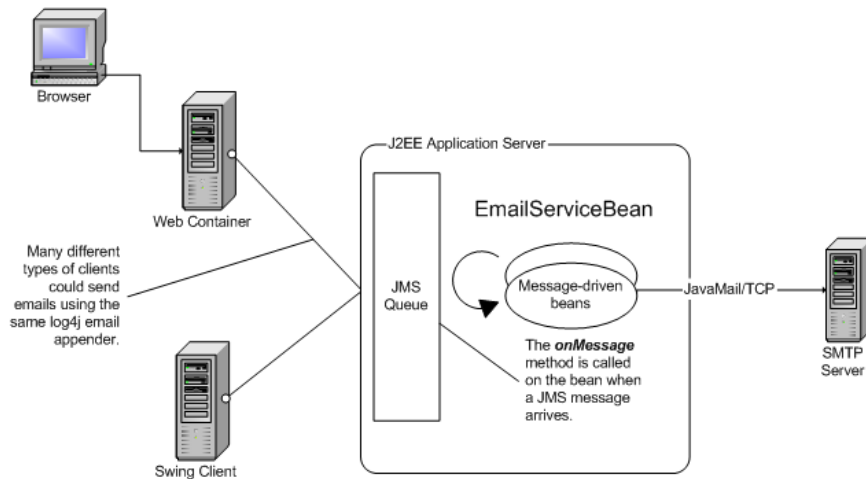


Figure 18-1. The log4j Email Appender generates email messages to a SMTP server

Example 18-13 shows the message-driven bean class that is listening on the Email Queue and sends emails to the SMTP server when the container calls the `onMessage` method.

Example 18-13. The Message-driven bean that Sends Emails to a SMTP server

```
/**
 * A Message-Driven bean that listens on a javax.jms.Queue for
 * messages and gets the email message out and sends it off to
 * a smtp host.
 */
package com.oreilly.struts.service.email;

import java.util.Date;
import java.util.Properties;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class EmailServiceBean
    implements javax.ejb.MessageDrivenBean, MessageListener {

    // Instance ref for the beans context
    MessageDrivenContext context = null;
```

```
// Default Constructor
public EmailServiceBean() {
    super();
}

/** The required onMessage method from the MessageListener interface
 * The onMessage method is not allowed to throw exceptions, so
 * we will catch every checked exception and just print out a
 * stack trace.
 */
public void onMessage( javax.jms.Message message ){
    // Local reference to the javax.mail.Session
    javax.mail.Session mailSession = null;
    try{
        //Make sure it's a type ObjectMessage
        if (!(message instanceof ObjectMessage)){
            return;
        }
        // Make sure it's an EmailMessage
        Object obj = ((ObjectMessage)message).getObject();
        if (!(obj instanceof Emailable)){
            return;
        }

        Emailable email = (Emailable)obj;

        Context ctx = new InitialContext();
        // Get the properties from this bean from the environment. The
        // properties were specified in the env-entry tags in the deployment
        // descriptor for this bean
        Context myEnv = (Context)ctx.lookup( "java:comp/env" );
        String smtpHost = (String)myEnv.lookup( "smtpHost" );

        Properties props = new Properties();
        props.put( "mail.smtp.host", smtpHost );
        // Get a mail session. You would normally get this from
        // JNDI, but some servers have a problem with this.
        // Each Message Driven bean instance is responsible for
        //getting its own unshared javax.mail.Session.
        mailSession = javax.mail.Session.getDefaultInstance( props, null );
        javax.mail.Message msg = new MimeMessage( mailSession );

        // Set the mail properties
        msg.setFrom(
            new javax.mail.internet.InternetAddress( email.getFromAddress() ) );

        InternetAddress[] addresses =
            { new InternetAddress(email.getToAddress()) };

        msg.setRecipients( javax.mail.Message.RecipientType.TO, addresses );
        msg.setSubject( email.getSubject() );
        msg.setSentDate( new Date() );

        // Create the body text
```

```
Multipart parts = new MimeMultipart();
MimeBodyPart mainBody = new MimeBodyPart();
mainBody.setText( email.getBody() );
parts.addBodyPart( mainBody );
// Could also have supported attachments, but not for this version
// it's commented it out.
/*
    MimeBodyPart attachmentBody = new MimeBodyPart();
    attachmentBody.setText( "This is text in the attachment" );
    attachmentBody.addBodyPart( p2 );
*/

// Set some header fields
msg.setHeader( "X-Priority", "High" );
msg.setHeader( "Sensitivity", "Company-Confidential" );
msg.setContent( parts );

System.out.println( "Sending mail to " + email.getToAddress() );
Transport.send( msg );
}catch( Exception ex ){
    // The onMessage method should not throw any kind of exceptions
    // according to the EJB 2.0 specification.
    ex.printStackTrace();
}
finally{
    mailSession = null;
}
}

public void setMessageDrivenContext( MessageDrivenContext ctx ){
    context = ctx;
}

public void ejbRemove(){
    // Not used for this bean
}

public void ejbCreate(){
    // Not used for this bean
}
}
```

Although example 18-13 might look complex, the only real work is going on in the `onMessage()` method. The J2EE container will call this method whenever a message arrives at the destination that this message-driven bean is configured for. In the case of our example, the destination will be the JMS Queue on which the `EmailServiceBean` is listening.

The majority of code in the `onMessage()` method utilizes the JavaMail API to create the appropriate mail object and then sends it to the SMTP server. The message-driven bean gets the information for the email message from an object that is passed in as an argument to the `onMessage()` method. This object should implement a Java interface called `Emailable`, which is shown in example 18-14.

*Example 18-14. The **Emailable** interface*

```
**
 * This interface defines methods that an object that wishes to
 * represent an email message, must implement.<p>
 */
package com.oreilly.struts.service.email;

public interface Emailable extends java.io.Serializable {
    public String getToAddress();
    public void setToAddress(String newToAddress);
    public void setFromAddress(String newFromAddress);
    public String getFromAddress();
    public void setSubject(String newSubject);
    public String getSubject();
    public void setBody(String newBody);
    public String getBody();
}
```

By forcing the objects to implement the **Emailable** interface, we are assured that the message-driven bean will have the information it needs to create a full email message.

The rest of the configuration properties that the message-driven bean uses come from the deployment descriptor that must accompany the bean when deployed into the container. Example 18-15 provides an example of the generic deployment descriptor.

Example 18-15. The Message-driven bean Deployment Descriptor

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>EmailServiceBean</ejb-name>
      <ejb-class> com.oreilly.struts.service.email.EmailServiceBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <jms-destination-type>javax.jms.Queue</jms-destination-type>
      </message-driven-destination>
      <env-entry>
        <description>The IP of the smtp host</description>
        <env-entry-name>smtpHost</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>127.0.0.1</env-entry-value>
      </env-entry>

      <security-identity>
        <run-as-specified-identity>
          <role-name></role-name>
        </run-as-specified-identity>
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

When the message-driven bean is deployed, it has access to the information stored in the `env-entry` tag elements in the deployment descriptor.

The only thing left to show is the log4j appender that will receive log messages and send the appropriate ones along to the JMS Queue. The JMS appender is shown in example 18-16.

Example 18-16. The `JMSQueueAppender` that Sends Email messages to a Queue

```
package com.oreilly.struts.framework.logging;

import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.spi.LoggingEvent;
import org.apache.log4j.spi.ErrorHandler;
import org.apache.log4j.spi.ErrorCode;
import org.apache.log4j.helpers.LogLog;

import java.util.Hashtable;
import java.util.Properties;
import javax.jms.*;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NameNotFoundException;
import javax.naming.NamingException;

import com.oreilly.struts.service.email.Emailable;
import com.oreilly.struts.service.email.EmailMessage;

/**
 * A log4j appender that sends messages to a JMS Queue that will
 * eventually send them to a SMTP server.
 */
public class JMSQueueAppender extends AppenderSkeleton {
    protected QueueConnection queueConnection;
    protected QueueSession queueSession;
    protected QueueSender queueSender;
    protected Queue queue;
    String icFactory;
    String providerUrl;
    String queueBindingName;
    String queueConnectionFactoryBindingName;

    public JMSQueueAppender() {
        super();
    }

    /**
     * Set the InitialContextFactory
     */
    public void setInitialContextFactory(String initialContextFactory) {
        icFactory = initialContextFactory;
    }

    /**
     * Retrieve the InitialContextFactory
     */
}
```



```
    */
    public String getInitialContextFactory() {
        return icFactory;
    }
    /**
     * Set the provider Url
     */
    public void setProviderUrl(String providerUrl) {
        this.providerUrl = providerUrl;
    }
    /**
     * Retrieve the provider Url
     */
    public String getProviderUrl() {
        return providerUrl;
    }
    /**
     * Set the name of the JMS Queue Connection
     */
    public void setQueueConnectionFactoryBindingName(String queueConnectionName){
        this.queueConnectionFactoryBindingName = queueConnectionName;
    }
    /**
     * Retrieve the name of the JMS Queue connection
     */
    public String getQueueConnectionFactoryBindingName() {
        return queueConnectionFactoryBindingName;
    }
    /**
     * Set the name of the JMS Queue
     */
    public void setQueueBindingName(String queueBindingName) {
        this.queueBindingName = queueBindingName;
    }
    /**
     * Retrieve the name of the JMS Queue
     */
    public String getQueueBindingName() {
        return queueBindingName;
    }
}

public void activateOptions() {
    QueueConnectionFactory queueConnectionFactory;

    try {
        Context ctx = getInitialContext();
        queueConnectionFactory = (QueueConnectionFactory)

            ctx.lookup(queueConnectionFactoryBindingName);

        queueConnection = queueConnectionFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
                                                            Session.AUTO_ACKNOWLEDGE);

        Queue queue = (Queue) ctx.lookup(queueBindingName);
```

```
        queueSender = queueSession.createSender(queue);

        queueConnection.start();
        ctx.close();

    } catch (Exception ex ) {
        errorHandler.error(
            "Error in activateOptions", ex, ErrorCode.GENERIC_FAILURE);
    }
}
/**
 * Return a new InitialContext
 */
protected InitialContext getInitialContext() throws NamingException {
    try {
        Hashtable ht = new Hashtable();

        //Populate property hashtable with data to retrieve the context.
        ht.put(Context.INITIAL_CONTEXT_FACTORY, icFactory);
        ht.put(Context.PROVIDER_URL, providerUrl);

        return (new InitialContext(ht));
    } catch (NamingException ne){
        LogLog.error("InitialContext failed");
        throw ne;
    }
}
/**
 * Return true if all of the neccessary resources are initialized.
 * Otherwise return false.
 */
protected boolean checkEntryConditions() {
    String fail = null;

    if(this.queueConnection == null) {
        fail = "No QueueConnection";
    } else if(this.queueSession == null) {
        fail = "No QueueSession";
    } else if(this.queueSender == null) {
        fail = "No QueueSender";
    }

    if(fail != null) {
        errorHandler.error(fail + " for JMSQueueAppender named [" + name + "].");
        return false;
    } else {
        return true;
    }
}
/**
 * Close all the resources used by the appender.
 */
public synchronized void close() {
    if(this.closed){
        return;
    }
}
```

```
LogLog.debug("Closing appender [" + name + "].");
this.closed = true;

try {
    if(queueSession != null){
        queueSession.close();
    }

    if(queueConnection != null){
        queueConnection.close();
    }
} catch(Exception ex) {
    LogLog.error("Error while closing JMSQueueAppender", ex );
}

// Help garbage collection
queueSender = null;
queueSession = null;
queueConnection = null;
}

/**
 * This method is called by AppenderSkeleton to do most of the real
 * appending work. The LoggingEvent will be wrapped in an ObjectMessage
 * to be put on the JMS queue.
 */
public void append(LoggingEvent event) {
    if(!checkEntryConditions()) {
        return;
    }

    try {
        ObjectMessage msg = queueSession.createObjectMessage();
        Emailable emailMsg = new EmailMessage();
        emailMsg.setToAddress( "administrator@somehost.com" );
        emailMsg.setFromAddress( "logMessageService@somehost.com" );
        emailMsg.setSubject( "Log Message for application received" );
        emailMsg.setBody( event.toString() );
        msg.setObject( emailMsg );
        queueSender.send( msg );

    } catch(Exception ex) {
        errorHandler.error("Error sendubg message", ex, ErrorCode.GENERIC_FAILURE);
    }
}

public boolean requiresLayout() {
    return false;
}
}
```

The Performance Impact of log4j

The potential performance impact of logging in an application is significant. For a typical production application, there can be thousands of log messages logged every day, depending on the logging threshold that's configured. The consequences might even be worse if exception stack traces are logged due to errors occurring in the application.

The log4j documentation states that much attention has been given to the performance impact of logging using log4j, however, nothing is free and there still is an additional cost associated with generating log messages. The question is how much of a cost and does this cost outweigh the benefits gained from the additional logging information.

The time cost of creating a log message using log4j depends on several factors:

- The type of Appender being used.
- The log4j Layout being used to format the messages.
- The parameter construction time to create the log message.
- Depth of the Logger hierarchy and where the log threshold is assigned.

The type of appender has much to do with how long it will take to create a log message. Logging a message using an SMTP appender will take a different amount of time than logging to the file system. On a typical machine using Java 1.3, a log message to the file system will take from 20 microseconds to upwards of 300 microseconds, depending on which layout you use. It's true that this is a very small number, but over time it can add up.

Probably the most significant impact on logging is what information you attempt to log and how you format that information. Log4j uses a subclass of `org.apache.log4j.Layout` to determine how the message should be formatted in the output destination. Using the `SimpleLayout` class is probably the fastest layout because it only logs the log level and the message. On the other hand, the `PatternLayout` class allows for a great amount of flexibility for the format of the message. You can log all sorts of information such as the class creating the message, the line number, and even the Java thread that is generating the message. However, all of this additional information comes at a severe price in terms of performance. The log4j documentation contains several warning messages in the JavaDocs for the `PatternLayout` class stating that performance may suffer greatly if certain information is written out with the log message.

You must be very particular and honest about what information you need placed in the log message. You should be able to get by with the class, the level, and the message. The other information is nice to have, but in most ordinary cases, just superfluous.

Creating the message that goes into the log statement can also have a lot to do with the time and performance. Creating a message using values such as:

```
| logger.debug("Session id is: " + sessId + " for user " + user.getFullName() ); |
```

can add significant time to the overall cost of logging. This obviously has nothing to do with log4j, but rather the cost associated with making Java method calls and concatenating the strings together before the actual log statement can be generated.

As mentioned earlier, loggers can be connected together in a parent/child relationship. Since the logging threshold can be assigned at any level, the log4j environment might have to search up the hierarchy to determine if the log message should be written or not. If this hierarchy is very deep, this traversal could add significant time to the log statement creation.

Generally, in the development stages of your application, the logging costs are not as important. The performance of logging shouldn't matter that much while you're still developing the application and debugging it. In fact, this is typically where you want the as logging as the application can generate. When it's time to go to QA or production, it's time to turn down the logging levels.

With other languages, the logging code might not ever make it into the compiled binaries. Preprocessors might remove the logging code to keep the binary size smaller and the log messages from showing up. However, this is not necessary using log4j. You have the flexibility of controlling how much logging, or none at all if you desire, just by changing the configuration file. Of course, the log statements are still present in the binary code once it's compiled.

Third-Party log4j Extensions

There are several helpful third-party tools and utilities available for log4j. Most of them are free to use or are open source. There are several Swing-based GUI applications that allow you to view and filter log messages dynamically. This would be an ideal application for an administrator in a production environment.

There are also several other types of appenders that other developers have created that might be helpful to you. You can find these third-party extensions in the log4j download area at <http://jakarta.apache.org/log4j/docs/download.html>. It's definitely worth the time to take a look at see what's there. The list of available extensions is constantly growing.

Java 1.4 Logging API

Some of you might already be using Java 1.4. At least, many of you have heard of it and the new features that have been included. One of the new features is the logging API now

included with the core library. You may be asking yourself what's the difference between log4j and this new library and why shouldn't you be using that instead.

Java 1.4 went final in early 2002. JSR47, which was the *Java Specification Request* (JSR) for the logging specification, had gone through some changes since it was first made public.

There seems to be several similarities between log4j and the 1.4 logging implementation. However, there also seems to be a few major differences. Let's discuss what's similar first. Both log4j and Java 1.4 logging implementation utilize a hierarchical name-space for the loggers. This allows you to configure loggers that basically align along your application's package structure, although this is not the only way to structure the loggers.

They also both support multiple levels or priorities. Actually the 1.4 logging implementation contains a few more levels than log4j, although you might not ever use the extra ones because they are so fine-grained.

The differences between the two implementations are generally not big enough to cause you to miss out on important functionality. However, it does appear that log4j offers more functionality for those that really need it. More importantly, log4j will work with Java version 1.1 and above, while the 1.4 logging implementation is dependent on 1.4 to work. There was talk about making it backwards compatible, but that hasn't happened yet and may never happen. Right now there are many more appenders types available for log4j than Java 1.4. However the most important ones will be present in 1.4, so this is not a major issue. There seems to be much attention being paid to the logging API in 1.4. You'll have to wait until it's final release to see how useful it will actually be.

Regardless of whether you use Log4J or Java 1.4 as your logging implementation, you should leverage the Commons Logging API to protect your application from inevitable change. Coupling your application to any single third-party implementation is not recommended, whether we are talking about logging or anything else.

20

Packaging Your Struts Application

Contrary to what many developers might assume, designing and building an application is only half of the battle. When the application is finished, it must then be packaged and deployed into the target environment. Many applications are built and packaged with the intent of being installed into a customer's production environment. For others, the target deployment environment is an in-house need. For web applications fortunately, the work that has to be accomplished in either case is very similar.

Internal deployments may be less formal and less nerve-racking. However, they still should be taken serious and conducted in an efficient and professional manner. Whether the customer is a "real" customer or another department within the organization, an unprofessional deployment can leave a bad impression. By formalizing the packaging and deployment process, the developers are more able to focus on building a quality application, and spend less time worrying whether the application will install and run correctly when it's finished.

This chapter discusses the best practices for packaging and deploying a Struts application, including coverage of what it takes to automate the build process for your environment. Special coverage will be given to Ant, the Java based build tool available from Jakarta.

To Package or Not to Package

Applications need to be deployed to be useful. There's really no point in developing an application that never gets deployed; although this occurs more often than some might think. The need for deployment is probably obvious to everyone, but what about

packaging? Does every Struts application have to be packaged before it gets deployed? The short answer is yes. In this section, though, we'll examine the long answer.

Before we get into the details of packaging and deploying Struts applications, let's define exactly what these two concepts mean in the context of web applications that are built using the Struts framework. Although the two concepts are closely related, they are not the same thing.

What is Packaging?

Packaging a Struts application involves gathering all of the files and resources that are part of the application and bundling them together in a logical structure. There are usually many different types of resources that are included in a Struts application, and all of these need to be bundled with the application. The following list provides some of the most common types of resources that must be packaged with a typical Struts application:

- HTML and/or XML Files
- Images, Audio, and Video Files
- Style Sheets
- JavaServer Pages
- Properties Files
- Java Utility Classes
- Configuration Files
- Action and ActionForm classes
- Third-party JARs

During the design stage, time and effort should be spent on deciding how you are going to structure the application. Not every detail needs to be figured out and resolved before construction begins, but you should generally understand the target environment requirements and how these requirements will affect your packaging and deployment strategy. Ideally, you should decide on the principal package and directory structure for the application before construction gets underway. This will help to alleviate the normal miscommunication between developers and reduce the redundant resource files.

What is Deployment?

As the previous section mentioned, packaging and deployment are closely related, but involve different tasks. While packaging determines where the resource files will reside in the package structure and how those resources will be bundled, deployment deals with

how the bundled application will be installed and configured inside a target web container.

There are two approaches that you can use when deploying a web application into a container. The first approach is to deploy the web application in a *Web ARchive* (WAR) format file. Most web containers can take a WAR file, install it and make it available for users, often without even requiring a restart of the container. This approach is convenient because once you have the WAR file properly built, the rest of the work is handled by the container. One of the downsides with this approach is that if there is a change made to any file or resource within the web application, a new WAR must be created and redeployed into the container. The details of how to deploy your Struts application as a WAR file are discussed later in the chapter.

The second approach to deploying a Struts application puts more work on the developer. It still involves packaging your application as a WAR file, but includes manually unpackaging the WAR into a directory of the container. The exact location of the directory is container dependant. In Tomcat and Resin for example, the default web applications directory is *webapps/*. In the WebLogic application server, you have to unpack the WAR file underneath the *applications/* directory.

All three of these containers allow you to specify alternate installation directories.

When expanding the WAR file, you need to create a directory to unpack the files into. The name of the directory is usually the name of the web application. So for example, if the web application was called *storefront* and you wanted to install it into Tomcat, you could create a directory called *storefront/* under the *webapps/* directory and unpack the WAR file there.

The WAR file should not contain the storefront directory as part of its structure.

This deployment approach is referred to as an *Exploded Directory Format*, because the WAR file is exploded back into its original structure within the container. The benefit with this approach over deploying the WAR file itself is that when there are changes, only the changed files need to be redeployed into the container. This is much easier while you're still developing or debugging the application, but you may not want to leave it like this for production.

When deploying a web application into production, it's a better approach to leave the WAR packed because there's less chance of one or more files getting contaminated. Leaving it as a single WAR forces you to replace the entire WAR entirely. There's no chance of a version of a file getting out of synch with the rest of the application.

Deciding on How to Package Your Application

Because Struts applications are web applications, most of the questions about how to package them are answered in the servlet and JavaServer Pages specifications. A web application must follow a very strict set of guidelines and conventions, which are completely specified, and which make the web application portable across other compatible web containers.

Fortunately, packaging your Struts application in the web application format solves much of the hassle regarding packaging. However, there are still many questions that you will have to resolve. For example, where do all of the JSP pages go, whether you store your Struts `Action` and `ActionForm` classes together or in separate action and form packages, and other issues similar to these will have to be decided. Although there are definitely best practices, there are no hard and fast rules for all of these questions. Some of these will depend on your organization's requirements and policies.

Namespace Management

A namespace is simply a set of names. These names may or may not have an association between each other. A namespace is normally used to prevent conflicts or collisions between similar entities and to allow clients to reference these entities by some logical name.

In software development, a *namespace* is a way to package classes, interfaces, and other types of information into a hierarchical structure. There are many examples of namespaces throughout the computer industry. The manner in which the Internet Domain Name System (DNS) works is a type of namespace. When you look up a web site address like <http://www.oreilly.com>, the DNS doesn't store the *oreilly.com* domain; it stores a reference to the IP address that has been associated with this name. Within the *oreilly.com* domain, there are other IP addresses that are linked together in a hierarchal fashion. All of this referencing helps prevent IP addresses from colliding. Another example, which is more closely related to software development, is the namespace that is used within JNDI. But by far the most familiar use of a namespace in Java development is for creating classes and interfaces that reside in a package.

As you know, Java applications are organized as a set of packages. Even when you don't specify a package explicitly, it's still part of a package¹. The purpose of the package is to prevent name collisions and to help identify entities (in this case Java classes) easily. When extending the Struts framework with your own classes and interfaces, you need to

¹ Any Java class that doesn't have a package statement declared in the source file is considered to be part of the *default package*.

decide on how best to package these components for your application. Not every application will be identical and contain the same classes and interfaces.

JSP File Placement

For many developers, where you place the JSP pages may seem like an easy question to answer. Although you may have to decide which directory a particular JSP page belongs in, that's normally all that has to be decided. However, there are situations where more control may need to be placed on who and what has access to the JSP pages. One suggestion has been to place JSP pages in a directory underneath the **WEB-INF/** directory.

The purpose of this approach is three-fold:

- Forces all requests to go through **ActionServlet**
- Prevents user from bookmarking a page
- The **WEB-INF/** directory is protected and helps protect JSPs from being called directly

This third approach has gained some popularity; however, there are a few problems with it. First, not all containers support this. Although the Servlet 2.3 Specification seems to indicate something to this effect may be possible, different vendors interpret the meaning of the specification differently. WebLogic for example, interprets the specification literally where it states in SRV.9.5 "No file contained in the **WEB-INF/** directory may be served directly to a client by the container.

The contents of the **WEB-INF/** directory are visible to the container using the **getResource()** and **getResourceAsStream()** methods using the **ServletContext**. Because of this, WebLogic will actually return 404 or 500 error codes when you attempt to access a JSP underneath the **WEB-INF/** directory.

Even though some containers do support this, you may not need to put the JSP pages underneath the **WEB-INF/** directory. If you only call Struts actions from your web application and don't link to JSP pages directly, which is a requirement for Struts 1.1, then this approach will not add a great deal of benefit for your applications. There are alternatives that you can use that are portable. For example, you can use the **security-constraint** element in the **web.xml** document. Just create the required directories for the JSP pages that you wish to protect. In the storefront example, suppose users shouldn't be able to access the JSP pages underneath the **order/** directory. A **security-constraint** element could then be added like this:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SecureOrderJSP</web-resource-name>
    <description>Protect the Order JSP Pages </description>
    <url-pattern>/order/*</url-pattern>
```

```
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
<auth-constraint>
  <role-name></role-name>
</auth-constraint>
</security-constraint>
```

Figure 20-1 shows what happens when a client attempts to access a JSP page directly within the *order/* directory.

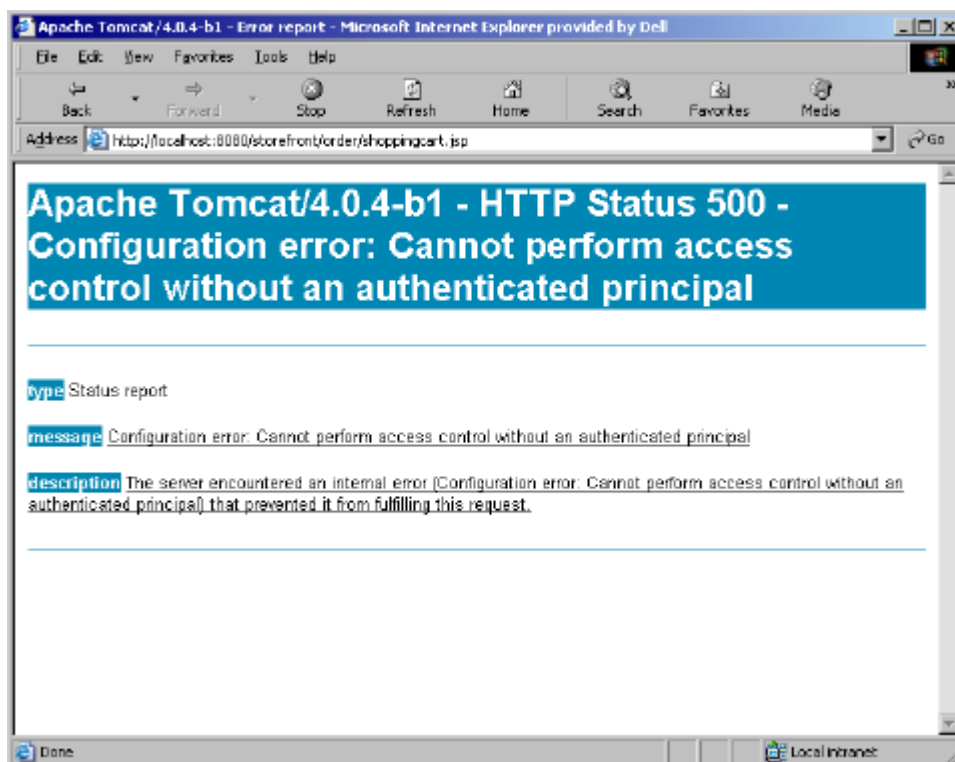


Figure 20-1. The container will return a 500 error when attempting to access a JSP page that has been protected.

When the *security-constraint* element is added to the *web.xml* document for the storefront, it says that no user without the proper authentication can access the order JSP pages. However, in this example, the *role-name* element was left intentionally blank and therefore will never match an actual role. You could also have specified that only users with an admin role could access the pages and then never give that role to an actual user. The container is still able to access these pages through forwards and includes.

You must be careful in what you put in the `url-pattern` element, however. If you have resources like images in sub-directories, they will not be available when the browser attempts to download them. Using the `/order/*` pattern means nothing under the `order/` directory can be requested directly by the client; this includes images in sub-directories, and the browser will have difficulty requesting the images for a returned HTTP response message.

Pre-Compiling JavaServer Pages

If you are using JavaServer Pages as your Struts presentation technology, you are probably very familiar with the compilation process that the JSP container performs on the application's behalf. When the web container receives a request for a resource with an extension of `.jsp`, the JSP container is selected to process the request. If this is the first request for the JSP page or if the timestamp for the JSP page is newer than the existing servlet class, the page will normally² be compiled.

The JSP container uses a two-step process. First, the JSP page is translated from the JSP code into a Java source file. Each container may have its own implementation to translate the JSP page. The second step takes the Java source file and compiles it into a servlet class file. This is done using whichever Java compiler is installed for the JSP container.

Although the just-in-time compilation is a nice feature while you are developing or debugging an application, you may not want to make the JSP source available in a deployment package. This is both an issue of security and licensing. Although the JSP pages should only contain presentation logic, it's still intellectual property.

If this is the case, then you can pre-compile the JSP pages and not provide the JSP source code to customers. The benefit is that you only have to distribute the compiled byte code, making it harder for the source code to be seen. Once the JSP pages are compiled into Java class files, you can even run them through an obfuscator to make it harder for them to be viewed through decompilation tools. By pre-compiling the JSP pages, you have more options for dealing with issues like these.

Like anything else, though, there are some downsides to pre-compiling JSP pages. For one thing, you lose the ability to update or fix a problem quickly. Instead of just placing a new JSP file onto the server and letting the JSP container compile it when it's accessed, you must now do the recompilation by hand and deploy the servlet class. Another downside is that when some containers detect a problem with a single JSP, that container will stop processing the rest of the application's JSP pages. When this occurs, you must

² Most containers have an option that disables detection and will not recompile pages that have changed. This is usually done in a production environment because incremental changes should not be introduced like this in production. To allow this to occur would also amount to a security risk as well.

make sure that every JSP page compiles successfully before the container will deploy the web application. Developers may actually consider this a benefit, rather than a disadvantage, but it can be problematic in production environments if you are trying to make a quick fix to a bug.

Unfortunately, there's not a standard way to pre-compile the JSP pages. Each container has a different way of doing it. We'll briefly discuss how to pre-compile pages using three of the available containers on the market, Tomcat 4.0, Resin 2.0, and WebLogic 7.0.

Precompiling JSP Pages with Tomcat

The JSP implementation in Tomcat, which is called Jasper, provides a reference implementation for the latest specification. It's packaged along with Catalina, which is the reference implementation of the latest servlet specification, inside of Tomcat. The JSP to Java compiler is made available as a separate program that can be executed from the command line. Its job is to convert a JSP page into a Java source file. From there, a standard Java compiler can convert the source code into byte code.

The program is called *jspc.bat* (or *jspc.sh*, depending of which platform you're using) and is located in the *bin/* directory of the Tomcat install directory. There are many options that can be set to configure everything from the output directory to the package name of the Java source. There's even an option that will cause the compiler to create XML that can be added to the *web.xml* document for each JSP page that gets precompiled. You can do one page at a time or specify an entire web application.

Precompiling JSP Pages with Resin

To precompile JSP pages using Resin, you can use the *httpd.exe* command from the Resin *bin/* directory like this:

```
| resin/bin/httpd -conf conf/resin.conf -compile <URL>
```

Here, the URL is a JSP resource within a web application installed in Resin.

You can also use the command line version like this:

```
| resin/bin/httpd -e <URL>
```

With this approach, you can only compile a single JSP page at a time, although you could easily create a script that went through your entire web application. An easier way to configure this would be to use Ant, which will be discussed later in this chapter.

Precompiling JSP Pages with WebLogic

With WebLogic 6.0, you have to include a *context-param* element in the deployment descriptor for each web application. This will instruct WebLogic to compile every JSP page that is part of the web application when the application server starts up. The

following example shows the `context-param` element that must be added to the *web.xml* file:

```
<context-param>
  <param-name>weblogic.jsp.precompile</param-name>
  <param-value>true</param-value>
</context-param>
```

WebLogic 6.0 is one of those containers that will not deploy the web application if any one of the JSP pages fails to compile. Once it detects an error compiling a page, all compilation stops and the web application will not be deployed. You will have to fix whatever is causing the problem and then restart WebLogic to start the process all over again.

Packaging EJB Resources with Struts

If you're communicating with EJBs in the middle tier, then it might be necessary to package some of the EJB resources along with your web application package. Since the web tier acts as a client to the EJB server, there are certain resources that are required to connect and communicate with the beans.

The beans' home and remote interfaces, for example, will need to be packaged either in the *classes/* directory or as a JAR in the *lib/* directory of the web application. Also, some of the JNDI classes will need to be included so that clients can acquire a home and remote object. The actual client-side EJB stub classes are not required, however. This wasn't always the case, but the latest specification now describes a mechanism that will allow these to be automatically downloaded when a request is made using an EJB remote interface.

In many cases, it's enough to put the EJB container JAR in the *WEB-INF/lib/* directory. For example, if you are using WebLogic 6.0 or higher, you can put the *weblogic.jar* in the web tier, as it contains all of the necessary client-side resources.

Packaging the Struts Application as a WAR

Packaging your web applications using the WAR format is very convenient. The structure is very precise and because it is specified very carefully, portability across the various web containers is much easier. The next section describes the exact steps that you must perform to package your web application as a WAR.

Creating the WAR File

The first step in packaging your application as a WAR is to create a root directory. In most cases, this will be the name of your web application. For our example, we will create a directory called *storefront/*.

After deciding how your JSP and HTML pages will be placed within your application, place them underneath the storefront directory, in their appropriate sub-directories. For the storefront package example, our directory structure so far would look like Figure 20-2.

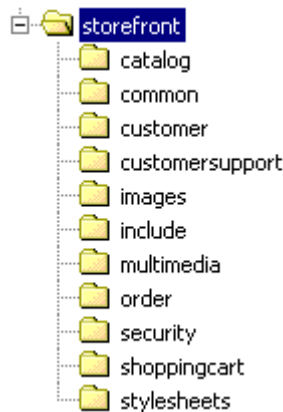


Figure 20-2. The Storefront Application Directory Structure

Figure 20-2 shows eleven sub-directories, eight of which contain JSP pages for the application. The *images/* directory will contain images that are used globally throughout the application. The *stylesheets/* directory store cascading stylesheets for the application and the include directory contains files that are included within the main JSP pages.

The next step in setting up the WAR is to ensure that you have a *WEB-INF/* directory underneath your root web application directory. The *WEB-INF/* directory contains all of the resources for the application that are used internally by the application. For example, the TLD files for your custom tag libraries reside in this directory, as does the web deployment descriptor. This is also where the Struts configuration files belong. No resource within this directory can be made public outside of the application.

Underneath the *WEB-INF/* directory, create two sub-directories; one called *classes* and the other called *lib*. In the *classes/* directory, you should place all of your utility and servlet classes. The *lib/* directory contains JARs that the application depends on.

Once all of the resources for the web application are accounted for and are inside the appropriate directory, you need to use Java's archiving tool *jar* to package the directories and files. From a command line, change to the root directory of your web application and use the jar utility. You need to give the file a *.war* extension like this:

```
| jar cvf storefront.war .
```

Since we changed to the root directory, the *storefront/* directory will not be included in the WAR file. This is fine, because you may want to call the root directory something else when it's installed. When you install the web application, if you plan to explode the WAR file, all you need to do is to create a directory and un-jar the files like this:


```
C:\tomcat\webapps>mkdir storefront
C:\tomcat\webapps>cp storefront.war storefront
C:\tomcat\webapps>cd storefront
C:\tomcat\webapps\storefront>jar xf storefront.war
C:\tomcat\webapps\storefront>
```

The location of where the WAR file gets expanded is dependant on the container. If you plan to distribute the WAR without exploding it, all you have to do is place it in the appropriate place for the container. You don't have to re-create the *storefront/* directory.

Although section 9.8 of the Servlet 2.3 Specification is a little ambiguous about replacing a web application without restarting the server, most containers will allow you to drop in new WAR file or replace certain files, without restarting. There's always a danger with leaving this features on in a production environment, however while still developing or debugging a problem, you'll quickly learn to appreciate this functionality. Of course, with the containers that do support this feature, there's always a way to disable it when you deploy a web application into production.

Building your Struts Applications with Ant

Although there are several different mechanisms to compile and deploy your Struts application, by far the easiest and most flexible is the Ant build tool. This section discusses how to use the open source tool to perform all of the compilation and deployment for your project.

What is Ant?

Ant is a platform independent build tool that can be configured to compile your Java source code files, build your deployment jar and WAR files, unit test your code, and create your project Javadoc documentation. It has many other uses and can be expanded to perform new tasks of your own creation.

Ant is similar to the Unix *make* utility (also known as *gmake* in Linux and *nmake* in DOS/Windows). *make* utilities have been used for generations to manage projects for many languages such as C and C++, but these utilities are platform dependent because the rules they use are typically shell commands executed by the host operating system. Unlike *make*, Ant's rules (or tasks, in ant terminology) are Java classes and can be run on any platform with a supported JVM.

A Brief History of Ant

Ant is another Jakarta project along with Struts and Tomcat and was created by James Duncan Davidson who also wrote the original Tomcat servlet container. Initially Ant was written as a utility to build Tomcat in early 1998. Very quickly, others saw the benefits of it over *make*, and it was moved to its own CVS project and officially became a separate Jakarta project in 2000. The creators of Ant had no idea how popular and widespread its use would eventually become.

Installing and Configuring Ant

Ant can be downloaded from the Apache/Jakarta website at <http://jakarta.apache.org/ant>. The examples provided in this chapter were tested with Ant version 1.4, but should also be compatible with at least Ant version 1.3. Download the binary *zip* file (for Windows) or the *tar.gz* (for Unix) and uncompress the archive into your desired installation directory. You should also download the 1.4.1 *optional.jar* and install it in the *ANT_HOME/lib/* directory. While not used in this project, the optional *jar* has many extra tasks that you may find useful in the future.

Ensure the *ANT_HOME/bin/* directory is added to your system *PATH*. Your installation may also require adding the environment variable, *ANT_HOME*, which should be set to the Ant installation directory. The Ant binary typically can determine what *ANT_HOME* should be, but if you get an error when trying to run Ant, set this environment variable. There is also a caveat when running under Windows 95/98 – do not install in a directory with long path names because the batch file used to run the installation script may not be able to handle the long path names. See the Ant installation documentation for more information.

Getting Started

Ant uses an XML file to read its build commands from. By default it looks for a file called *build.xml*, but any XML file can be a build file by using the *-buildfile* *<file>* option when running Ant. From a command prompt, change directories to the base project directory for the storefront project, which is called *storefront/*. In here you should see the *build.xml* file.

The Ant build file consists of a project that has one or more targets, and each target consists of one or more tasks. The project element is defined at the top of the build file:

```
| <project name="storefront" default="war" basedir="."> |
```

The project is named `storefront` and the default target to execute is the `war` target. The default target is what gets executed if you only type `ant` from the command line, without specifying a target. Because the project root directory is the same directory that the `build.xml` file is located, “.” is used to indicate the base directory property.

The build directory structure for the storefront application is shown in Figure 20-3.

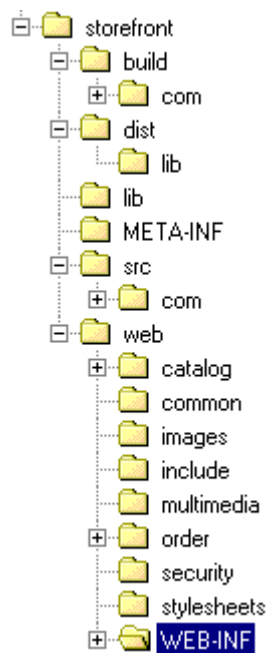


Figure 20-3. The Build Structure for the Storefront Application

A target takes the form:

```
<target name="dostuff">
  <task1 param1="value1" param2="value2"/>
  <task2 param="value"/>
  ...
</target>
```

The target should consist of one or more tasks. A target must have a name and may have several additional attributes that determine when and if the target actually gets executed.

A task, on the other hand, is an atomic unit of work in the Ant world. Each task is bound to a Java class file that Ant executes passing to it any arguments or sub-elements defined with that task. For this project, and most likely all projects you’ll create, the built-in and optional tasks included with the Ant distribution are all you will need. If, however, you needed to create a new task, you could do so by defining the task in the `build.xml` file using the `taskdef` task to bind the task name to a Java class file. The Java class file must extend `org.apache.tools.ant.Task` and be located in the Ant

CLASSPATH. There are several other requirements that are beyond the scope of this book, but details can be found in the Ant documentation.

Before you run Ant, there are a couple properties in the storefront *build.xml* file you may have to change to suit your development environment:

```
<property name="webserver.home"
  value="c:/tomcat"/>
<property name="webserver.deploy"
  value="${webserver.home}/webapps"/>
<property name="servlet.jar"
  value="${webserver.home}/common/lib/servlet.jar"/>
```

These three properties define where the servlet container is located, where its deployment directory is located, and where it keeps the servlet API classes. First, the *webserver.home* property is set to the root directory of the servlet container. In this case, the Tomcat 4.0 web server and servlet container is being used. The storefront *build.xml* file supports several other containers, but they are commented out. You'll just need to uncomment the one that you wish to use. Tomcat's deployment directory is the *webapps/* directory found just off the Tomcat root directory. Tomcat keeps the servlet API classes in the *common/lib/servlet.jar* relative to the Tomcat root directory.

Lastly, we need to define our project **CLASSPATH** that will be used during compilation of our project. Ant provides the capability to associate a set of files to a property name. In the following *build.xml* fragment, the list of all the jars necessary to compile the storefront example is bound to the *build.classpath* property. We could have used *include* elements to define the *build.classpath* property in far fewer lines; however it's much more clear to explicitly list each *jar* used during the build process so that nothing extra is added or omitted which might prevent a successful build:

```
<path id="build.classpath">
  <pathelement location="${servlet.jar}"/>
  <pathelement location="${lib.dir}/commons-beanutils.jar"/>
  <pathelement location="${lib.dir}/commons-collections.jar"/>
  <pathelement location="${lib.dir}/commons-dbcps.jar"/>
  <pathelement location="${lib.dir}/commons-digester.jar"/>
  <pathelement location="${lib.dir}/commons-logging.jar"/>
  <pathelement location="${lib.dir}/commons-pool.jar"/>
  <pathelement location="${lib.dir}/commons-services.jar"/>
  <pathelement location="${lib.dir}/commons-validator.jar"/>
  <pathelement location="${lib.dir}/jdbc2_0-stdext.jar"/>
  <pathelement location="${lib.dir}/log4j.jar"/>
  <pathelement location="${lib.dir}/poolman.jar"/>
  <pathelement location="${lib.dir}/struts.jar"/>
  <pathelement location="${lib.dir}/tiles.jar"/>
  <pathelement path="${build.dir}"/>
</path>
```

The *build.classpath* property can be used by the tasks by de-referencing the property name using the Ant syntax *\${property}*. You'll see this used later in the chapter within the Java compiler task to define the **CLASSPATH**.

Compiling Java Source Files

The Java source files for the storefront application are compiled using the Ant `javac` task. The compiling target is called, simply enough, `compile`, and only depends on the `prepare` target:

```
<target name="compile" depends="prepare">
  <javac destdir="${build.dir}" deprecation="on">
    <classpath refid="build.classpath" />
    <src path="${src.dir}" />
  </javac>
</target>
```

A target can depend on no other targets, or it can depend on multiple targets by using the following syntax:

```
<target name="final-jar" depends="jars, wars">
```

By specifying a `depends` attribute, we can control the order in which Ant targets are executed. In this case, the `compile` target is not executed until the `prepare` target has been executed:

```
<target name="prepare">
  <tstamp/>
  <mkdir dir="${build.dir}" />
  <mkdir dir="${dist.dir}/lib" />
</target>
```

The `prepare` target generates timestamp values which could be turned into properties and attached to compilation products such as jar and WAR files. For this small project however, timestamps will not be used. The `prepare` target also creates the necessary output subdirectories for our java classes and WAR file.

The `compile` target instructs Ant to run the `javac` compiler on all the files within the source directory and send all the class files to the build directory. The deprecation option is on so that you'll get a detailed message if you've accidentally included a deprecated method in one of the source files:

```
<target name="compile" depends="prepare">
  <javac destdir="${build.dir}" deprecation="on">
    <classpath refid="build.classpath" />
    <src path="${src.dir}" />
  </javac>
</target>
```

The `javac` task uses the `build.classpath` property that was shown earlier in the chapter.

Using Ant to build the WAR File

The Ant `war` task builds the web archive. The `war` target used to build the web archive is shown here:

```
<target name="war" depends="compile">
  <echo>building war...</echo>
  <war warfile="${dist.dir}/lib/storefront.war"
      webxml="${web.dir}/WEB-INF/web.xml">
    <fileset dir="${web.dir}" />
    <classes dir="${build.dir}" />
    <classes dir="${lib.dir}">
      <include name="*.properties" />
      <include name="poolman.xml" />
    </classes>
    <lib dir="${lib.dir}">
      <include name="*.jar" />
    </lib>
  </war>
</target>
```

As mentioned previously, the `war` target is the default target for the project. This means when Ant is run from the command line without a target argument, the `war` target is executed. The `war` target only runs if the `compile` target has been ran first. The `war` task requires you to define the name of the WAR file and the location of the `web.xml` descriptor. All the other attributes are optional, but if you are interested in seeing them, they are listed in the online Ant documentation (<http://jakarta.apache.org/ant/manual/CoreTasks/war.html>). You should also check out *Ant: the Definitive Guide* (O'Reilly), by Jesse Tilly and Eric Burke.

The nested elements tell the `war` task where the contents of the WAR file are located. The `fileset` element defines the base web content of the WAR file. This element is used to declare where the HTML files, JSPs, images, and such are located. The `classes` element points to your Java class files that should be included in the `WEB-INF/classes/` directory in the WAR, and the `lib` element declares which files should be included in the `WEB-INF/lib/` folder.

In the storefront example, everything in the `web/` subdirectory is included. The various sub-directories contain all of the necessary resources like HTML, JSPs, images, and so on. All of the compiled classes in the `build/` subdirectory are copied into the WAR's `WEB-INF/classes/` directory along with the properties files. All of the third-party jars in the `lib/` subdirectory are copied into the WAR's `WEB-INF/lib/` directory. If the `lib/` subdirectory contained one or more jars that you didn't want to be included, you can also use the following snippet:

```
<lib dir="${lib.dir}">
  <include name="*.jar" />
  <exclude name="dont_need.jar" />
</lib>
```

Here, all the jars in `lib/` except `don't_need.jar` would be copied into the WAR's `WEB-INF/lib/` directory. The last option, and often the most clear, is to explicitly include each desired `jar` file. While slightly more verbose, it is immune to changes to the `lib/` folder if other developers in the project start adding `jars` indiscriminately. It is also much easier to see exactly what is going to be included in the WAR file.

Cleaning Up

The final two targets are trivial, but important. The `clean` target deletes the `build/` directory, thus removing all the Java class files:

```
<target name="clean">
  <delete dir="${build.dir}" />
</target>

<target name="distclean">
  <antcall target="clean"/>
  <delete dir="${dist.dir}" />
</target>
```

The `distclean` target reverts the project back to its pristine “distribution” state. That is, all build products – class files and WAR file – are removed so that the project directory looks the same as it would have when the project directory tree was first installed.

Note how the `distclean` target calls the `clean` target. While not really necessary for this small project, it demonstrates more of the power of Ant, by invoking another target via the `antcall` task. In fact, the `antcall` task can even call a target with arguments, but that is beyond the scope of this book.

There are various plugins that you can download that allow you to use Ant inside of your specific IDE. For example, the AntRunner plugin allows Ant to be used within the Jbuilder IDE. For this and other plugins, see the external tools section of the Ant site: <http://jakarta.apache.org/ant/external.html>.

Creating an Automated Build Environment

Once you have put together a satisfactory build environment, you should go the extra step to automate it. This means that there’s no human interaction needed to execute new builds. It’s very common to need to modify the times that builds occur as you get closer to the end of a construction phase. You can of course, just manually kick off builds when you need them, but a better, and much more efficient approach is to use a scheduling mechanism to invoke your Ant build environment.

There are two scheduling mechanism that you can employ, depending on your environment. In Unix, you can use the `cron` daemon and on the Windows platform, you can take advantage of the Task Scheduler.

Using *cron* to Invoke Ant

cron is a program that allows you to automate tasks by running user-defined programs at regular intervals. *cron* allows you to define both the program that is to be run, as well as the exact time to run the program. The *cron* program is a daemon program, which means that it runs in the background until it's needed. It wakes up every minute, checks to see if there are any scheduled tasks that need to be performed, and after it runs the tasks, it goes back to waiting.

The list of tasks for *cron* to execute comes from a file called a *cron* table, which is commonly referred to as *crontabs*. The *crontabs* is a schedule that lists the tasks to run and the date and time to run them.

Some system administrators may disable *cron* from running on a server, to save on processing power. If this is the case, you'll need to ask them to allow *cron* to resume running. You can verify that *cron* is running and that you have permissions to access it by typing *crontab -l* from the command prompt.

All you need to do is edit the *cron* table and add an entry that calls your *ant* program, which in turn will kick off the build. Check the Unix man pages for more information on using *cron*.

Using Microsoft Task Scheduler

If you're using the Windows platform, it contains an application called Task Scheduler, or Scheduled Tasks in newer versions of the platform. This program performs the same function as *cron*, but for the Windows platform. It has a wizard that walks you through setting up a task that gets fired off at regular intervals. Figure 20-4 is the main screen of the scheduled task application. A task has been created that will invoke a batch file that can in turn call Ant to perform a build for your environment.

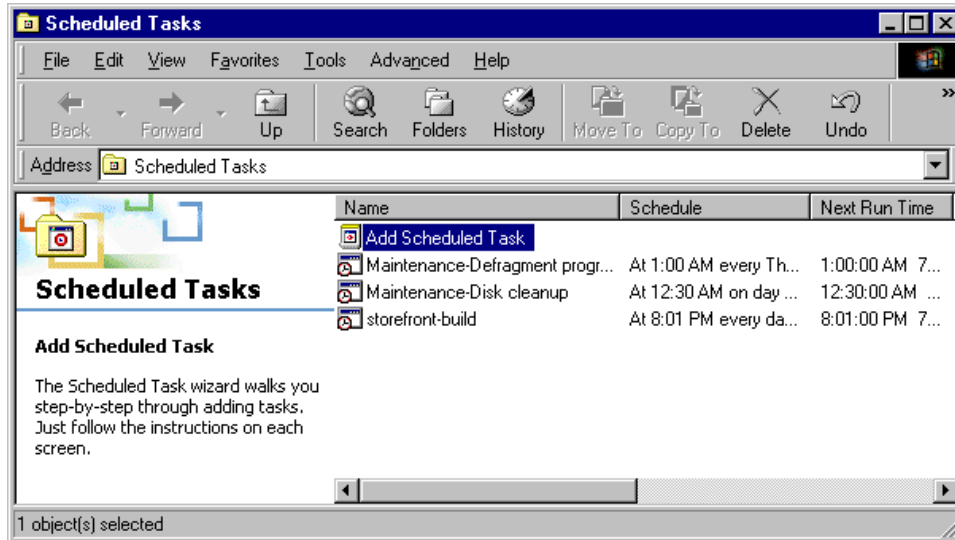


Figure 20-4. The Scheduled Task Application on the Windows Platform

The amount of control that you have over the time intervals is not as granular as it is on the Unix platform. However, it is sufficient enough to kick off daily builds at a certain time. See the Windows documentation for your specific version for more details.

FTPing and Restarting your Server

If your deployment server is located on a different server from the build environment (and it really should be) you will need to get the deployment files moved over and possibly restart the server. The easiest way to do this is with the Ant `ftp` and `telnet` tasks. You can even have Ant email you reports on the result of the latest build using the `email` task. See the Ant documentation for more information on these and other helpful tasks.