

STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis

MEAP

Unedited Draft

 MANNING

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>



**MEAP Edition
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table of Contents

Part 1: Struts 2: A Brand New Framework

Chapter1 – Struts2: The Modern Web Application Framework

Chapter 2 – Saying Hello to Struts 2

Part 2: Getting to the Heart of the Matter: Actions, Interceptors, and Type Conversion

Chapter 3 – Working with Struts2 Actions

Chapter 4 – Adding workflow with Interceptors

Chapter 5 – Data Transfer: OGNL and Type Conversion

Part3: Building the View: Tags, UI Components, Results

Chapter 6 – Building a view: tags

Chapter 7 – UI Component Tags

Chapter 8 – Results in detail

Part 4: Finishing the App: Resource Management, Validation, Data Persistence, and Internationalization

Chapter 9 – Integrating with the rest of the application

Chapter 10 – Validating form Data

Chapter 11 – Understanding Internationalization

Part 5: Advanced Topics

Chapter 12 – Extending Struts with Plugins

Chapter 13 – Best Practices

Chapter 14 – Migration from Struts or WebWork

Chapter 15 – Advanced Topics

Appendix A – Annotations Based Architecture

Appendix B – Struts 2 Architectures

Appendix C – Struts 2 Community

Appendix D – Freemarker in a nutshell

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

1 Struts 2: The Modern Web Application Framework

Welcome to Struts 2! If you've picked up this book, we suspect you are a Java developer working with web applications who has somehow or another heard about Struts 2. Perhaps you have worked with the Struts 1 framework in the past, perhaps you have worked with another framework, or perhaps this is your first step into Java web application development, but whichever path has led you here you're probably looking for a good introduction to the new Struts 2 framework.

This book intends to give you that introduction, and much more. Struts 2 is much more than a revision of the Struts 1 framework. If you hadn't yet heard anything about Struts 2, you might expect, based upon the name, to find a new release of that proven framework. But this is not exactly the case. Its relationship to that older framework is based in philosophy rather than in code base. Struts 1 was an action oriented framework that implemented an MVC separation of concerns in its architecture. Struts 2 is a brand new implementation of those same MVC principles in an action oriented framework. While the general lay of the land will certainly seem familiar to Struts 1 developers, the new framework contains substantial architectural differences. We will cover the new framework from the ground up, taking time to provide a true introduction to this new technology.

However, this book certainly goes deeper than an introduction. This book also takes the *in Action* portion of its title to heart and will provide you with practical information for building full fledged, enterprise class web applications ready for integration with all of the latest and greatest development practices and enterprise components from unit testing to Inversion of Control resource management and built-in Ajax widgets. In addition to exploring all of the core framework components in detailed depth, we will also develop a non-trivial sample application, the Struts 2 Portfolio. We will develop this application in a series of versions that implement increasingly complex features as we progress through the book. The early versions of this sample application will provide simple demonstrations of the frameworks core components. Later versions will boast the full feature set and demo some best practices.

But the first two chapters are introductory in nature. In this first chapter, we will do a quick survey of the context in which Struts 2 occurs, including short studies of web applications and frameworks. We will then take the obligatory architectural look from 30,000 feet. Unless your familiar with WebWork, the true code base ancestor of Struts 2, then this high level overview of the framework will be your first look at a fairly new and interesting way of doing things. It will also give you an introductory grasp of the framework's core components in preparation for Chapter Two's HelloWorld application, a simple tutorial that will give skeletal demonstration of a Struts 2 application. Since we've got a lot on our plates for chapter one, we best get started.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

1.1 Web applications: a quick study

This section intends to provide a rough primer on the technological context of a web application. We will cover the technology stack upon which web applications sit, and we will take a quick survey of the common tasks that all web applications must routinely accomplish as they service their requests. This book has a wide target audience. If you are quite familiar with this information, you could easily skip ahead to the Struts 2 architectural overview. Even if you are familiar with the following material, a quick read through the following sections would still provide an orientation on how we, the authors, view the web application domain. At any rate, we will try to be concise.

1.1.1 Using the Web to build applications

While many Java developers in today's market may have worked on web applications for the largest part of their careers, it's always profitable to revisit the foundations of the domain in which one is working. A solid understanding of the technological context in which a web application framework such as Struts 2 is situated helps to provide an intuitive understanding for the architectural decisions made by the framework. And, again, establishing a common vocabulary for our discussions will make everything easier throughout the entire book.

A web application is simply, or not so simply, an application that runs over the web. With the rapid improvements in Internet speed, connectivity, client and server technologies, the web application has become an increasingly powerful platform for building all classes of applications from the standard business oriented enterprise solutions to the latest migrations of personal software from the desktop to the web. The latest iterations of web applications demand the functional feature set and look and feel of traditional, desktop based applications. Yet, in spite of the increasing variety in the applications built on the web platform, the core workflow of these applications remains markedly consistent, a perfect opportunity for reuse. A framework such as Struts 2 strives to relieve the developer from the mundane concerns of the domain by providing a reusable architectural solution to the core web application workflows.

1.1.2 Examining the Technology Stack

We will now take a quick look at two of the main components in the technology stack upon which a web application sits. In some sense, the web is a simple affair. As with all good solutions, if it weren't simple it probably wouldn't be successful. Figure 1.1 provides a simple depiction of the technological context in which Struts 2 occurs.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

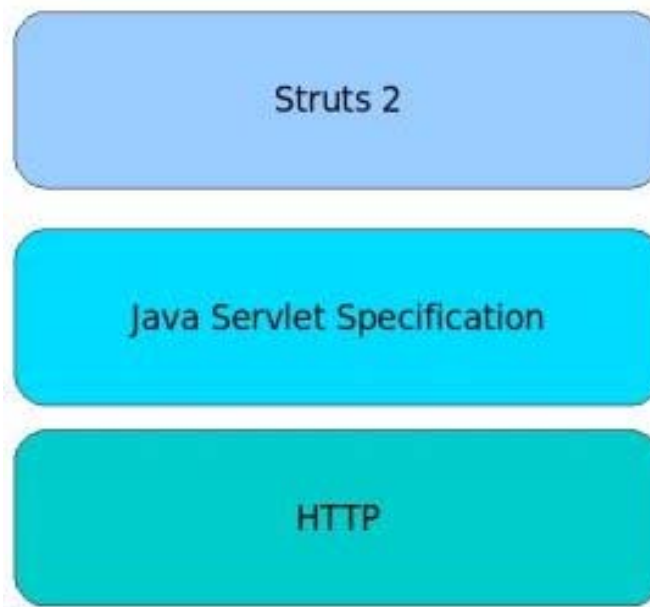


Figure 1.1 Struts 2 is Built on the Java Servlet API's Exposure of the HTTP Client Server Protocol to the Java Platform

As depicted in Figure 1.1, Struts 2 sits on top of two important technologies. At the heart of all Struts 2 applications lies the client – server exchanges of the HTTP protocol. The Java Servlet API exposes the low level communication of HTTP to the Java language. While its possible to write web applications by directly coding against the Servlet API, its generally not considered to be a good practice. Basically, Struts 2 uses the Servlet API so that you don't have to. But while its generally considered a good idea to keep the Servlet API out of your Struts 2 code, it seems somewhat cavalier to enter into Struts 2 development without some idea of the underlying technologies. The next two sections provide concise descriptions of the more relevant aspects of HTTP and Java servlets.

HyperText Transfer Protocol (HTTP)

Most web applications run on top of HTTP. This protocol is a stateless series of client-server message exchanges. Normally, the client is a web browser and the server is a web server or application server. The client initiates communication with the server by sending a request for a specific resource. The resource that the client requests can be a static HTML document that exists on the server's local file system, or it can be a dynamically generated document with untold complexity behind its creation.

Much could be said about the HTTP protocol and the variety of ways of doing things in this domain. We will limit ourselves to the most important implications as seen from the perspective of a web application. We can start first by noting that HTTP was not originally designed to serve in the capacity that web application developers demand of it. It was meant to be a protocol for requesting and serving static HTML documents. This original use case provides us with a multi-part legacy which all web applications built on HTTP must address.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

For web applications, HTTP has two hurdles to address. It is stateless, and it is text based. A stateless protocol doesn't keep track of the relationships between the various requests it receives. Each request is handled as if it were the only request the server had ever received. The HTTP server keeps no records that would allow it to track and logically connect multiple requests from a given client. The server has the client's address, but it will only be used to return the currently requested document. If the client turns around and requests another document, the server will have no cognizance of this client's repeated visits.

But if we are trying to build more complex web applications with much more complicated use cases, this will not work. Take the simplest, most common case of the secure web application. A secure application needs to authenticate its users. To do this, the request in which the client sends the user name and password must somehow be associated to all other requests coming from that same client during that same user session. Without the ability to keep track of relationships between various requests, even this introductory use case of modern web applications is impossible. This problem must be addressed by every modern web application.

Equally as troublesome as its lack of state, HTTP also has the stubborn quality of being text based. Mating a text based technology to a heavily typed technology like Java creates a significant amount of data binding work. While in the form of an HTTP request, all data must be represented as text. Somewhere along the way, this encoding of data must be mapped onto Java data types. Furthermore, this process must occur at both ends of the request handling process. Incoming request parameters must be migrated into the Java environment, and outgoing responses must pull data from Java back into the text based HTTP response. While this is not rocket science, it can create mounds of drudge work for a web application. These tasks are both error prone and time consuming.

Java Servlet API

The Java Servlet API helps alleviate some of the pain. This important technology exposes HTTP to the Java platform. This means that Java developers can write HTTP server code against an intuitive object oriented abstraction of the HTTP client – server communications. The central figures in the Servlet API are the servlet, request and response. A servlet is a Java object whose whole purpose is to receive a request and return a response after some arbitrary back end processing. The request object encapsulates the various details of the request, including the all important request parameters as submitted via form fields and querystring parameters. The response object includes such key items as the response headers and the output stream that will generate the text of the response. In short, a servlet receives a request object, examines its data, does the appropriate back end magic, then writes and returns the response to the client. Simple enough.

Pull-out: Sun and the Servlet Specification. If you are unfamiliar with Sun's way of doing things, here's a short course. Sun provides a specification of a technology, such as the Servlet API. The specifications are generated through a community process which includes the input of a variety of interested parties, not the least of which is Sun itself. The specification merely details the obligations and contracts that the API must honor; actual implementations are provided by various third party vendors. In the case of servlets, the implementations are Servlet containers. These containers can be stand alone implementations such as the popular Apache Tomcat, or they can be containers embedded in some larger application server. They also run the gamut from open source to fully proprietary. If you are unfamiliar with the Servlet Specification, we highly recommend reading it. It's actually short, to the point, and well written.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

We also need to know a bit about how servlets are deployed as working applications. Before you deploy servlets, you must first package them according to the standards. The basic unit of servlet packaging is known as a *web application*. Though it sounds like a general term, a web application is a very specific thing in servlet terminology. The Servlet Specification defines a web application as “a collection of servlets, HTML pages, classes, and other resources . . .” Typically, a web application will require several servlets to service the request of its clients. A web application's servlets and resources are packaged together in a very specific directory structure, and zipped up in a .war file. A war file is a specialized version of the jar file. The letters stand for Web Application Archive. When we discuss Chapter Two's HelloWorld application, we will see how to layout a Struts 2 application to these standards.

Once you have packaged the web application, you need to deploy it. Web applications are deployed in *servlet containers*. Servlets are managed life cycle software entities. This just means that you don't directly execute a servlet. You deploy it in a container and that container manages its execution by invoking the various servlet life cycle methods. When a servlet container receives a request it must first decide which of the servlets that it manages should handle the request. When the servlet container determines that a given servlet should process a request, it invoke that servlet's service() method, handing it both a request and response object. There are other life cycle methods, but the service() method is responsible for the actual work of the servlet.

Figure 1.2 shows the relationship between the key players of the Servlet API: servlets, web applications and the servlet container.

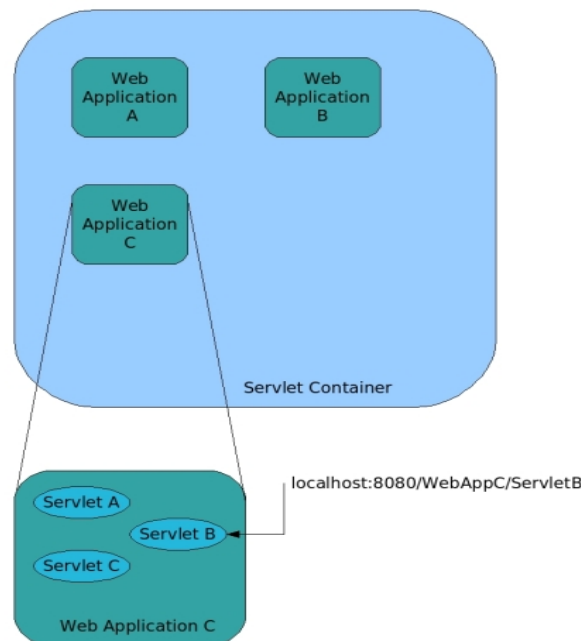


Figure 1.2 – The Organization of the Servlet API: Servlets, Web Applications, and the Servlet Container

As you can see, a servlet container can host one or more web applications. In Figure 1.2, three web applications have been deployed to a single container. All requests, regardless of which web

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

application they ultimately target, must first be handled by the container. The servlet container typically listens on port 8080 for requests. When a request comes to that port, it must then parse the namespace of the request to discover which web application is targeted. From the namespace of the URL, both the web application and the individual servlet targeted there within can be determined. The full details of this parsing process are certainly not in the scope of this overview, but Figure 1.2 gives a rudimentary example of how a URL maps to a specific servlet, assuming the servlet container is on the localhost.

In addition to just exposing HTTP to the Java language, the Servlet API provides one important high level functionality. It provides a session mechanism that allows us to correlate groups of requests from a given client. As we explained earlier, HTTP doesn't provide any sense of state across a set of requests regardless of whether they all came from the same client. This is perhaps the most important thing, in terms of higher level functionality, that we receive from servlets. And this is a very important thing. Without it, we'd be handling cookies and parsing embedded querystring session keys.

Apart from the session mechanism, the Servlet API doesn't provide a whole lot of higher level functionality. It pretty much directly encapsulates the details of the client – server exchange in a set of object oriented abstractions. We're not saying this isn't a great thing. This means that we don't have to parse the incoming HTTP request ourselves. Instead, we receive a tidy request object, already wrapped up in Java. We say this to make the point that, ultimately, the Servlet API is an infrastructural level technology in the scope of modern web applications. As infrastructure, servlets provide the solid low level foundation upon which robust web applications can be built. If you consider the routine needs of a web application, the Servlet API does not really attempt to provide solutions for such things. Now that we know what servlets can do, let's look at what they leave undone. It will be these common tasks of the domain that a web application framework like Struts 2 will need to address.

1.1.3 Surveying the Domain

With the Servlet API addressing the low level client – server concerns, we are now able to focus on the application level concerns. There are many tasks that all web applications must solve as they go about their daily routine of processing requests. Among these are:

- Request parameter binding
- Data validation
- Calls to business logic
- Calls to data tier
- Rendering presentation layer (HTML & Co.)
- Internationalization

We will examine each of the concerns briefly in the following paragraphs.

Request parameter binding and data validation

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Being a text based protocol, HTTP must represent its request parameters in a text encoding. Of course, when these parameters enter our application they must be converted to the appropriate native data type. The Servlet API does not do this for us. The parameters, as retrieved from the servlet request objects, are still represented as strings. Of course, converting these strings to Java data types is easy enough work if you have the time. But this can be time consuming and error prone. Converting to simple types is tedious, converting to more complex types is both complex and tedious. And, of course, the data must also be validated before it can be allowed to enter the system. Note, there are two levels of validation. In the first case, the string must be a valid representation of the Java type to which you want to convert, i.e. a zip code should not have any letters in it. Then, after the value has been successfully bound to a Java type, an application level validation must determine whether the value itself is within the acceptable range of values according the business rules of the application. Perhaps a check that the zip code is a valid zip code. Spending too many hours writing this kind of code can certainly make Jack a dull boy.

Calls to business logic and the data tier

Once inside the application, most requests involve calls to business logic and the data layer. While the specifics of these calls varies wildly from application to application, a couple of generalizations can be drawn. First, despite the variance in the actual details of these calls, they do form a consistent pattern of workflow wherein, at its core, the processing of each request holds a chunk of work that must be done. This work is the *action* of an action oriented framework. Second, this logic and function of this work represents a clear step outside of the web related domain. If you take a quick look back to our list of the common tasks that a web application must do in the processing of its requests, you'll see that these calls to business logic and the data tier are the only ones that don't specifically pertain to the fact that this is a web application, as opposed to, say, a desktop application. If the application is well designed, the business logic and data layers would be completely oblivious to whether they were being invoked from a web application or a desktop application. So, while all web applications must make these calls, the notable thing about these calls is that they are outside the specific workflow concerns of a web application.

Presentation rendering and internationalization

In some manner, the presentation tier of a web application is just an HTML document. However, increasing amounts of complex Javascript, fully realized CSS, and other embedded technologies make it no longer feasible to refer to a web page as “just an HTML document.” Simultaneous to this increase in the complexity of front end user interface technology, comes the increasingly less optional option of internationalization. Internationalization allows us to build a single web application that is capable of discovering the locality of each user and providing locale specific language and formatting of date, time, and currency. Whether an application is returning a simple page of static text, or returning a Gmail-esque super client, the rendering of the presentation layer is a core domain task of all web applications.

These tasks that we have just outlined, being common to processing of nearly every request that comes to a web application, are perfect candidates for re-use. Web application frameworks are a perfect way to provide re-usable solutions to these tasks.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

1.2 Frameworks for Web Applications

Now that we have oriented ourselves to the domain in which web applications operate, we can talk about how a framework can alleviate the work of building them. To build powerful web applications, most developers need all the help they can get. It seems that you must use a framework, and there are a lot of them out there. But let's start with a fundamental question.

1.2.1 What's a Framework?

Fair enough. A framework is a piece of structural software. We say structural because structure is perhaps a larger goal of the framework than any specific functional requirement. A framework tries to make generalizations about the common tasks and work flow of a specific domain. The framework then attempts to provide a platform upon which applications of that specific domain can be more quickly built. The framework does this primarily in two ways. First, the framework tries to automate all the tedious tasks. Second, the framework tries to introduce an elegant architectural solution to the common workflow of the domain in question.

Definition: A web application framework is a piece of structural software that provides automation of common tasks of the domain as well as providing a built in architectural solution that can be easily inherited by applications implemented on the framework.

Automates common tasks

Don't re-invent the wheel. Any good framework will provide mechanisms for convenient and perhaps automatic solutions to the common tasks of the domain, saving the developers the effort of re-inventing the wheel. Reflecting back on our discussion of the common tasks of the web application domain, we can then infer that a web application framework will provide some sort of built in mechanisms for things like converting data from their HTTP string representation to Java data types, data validation, separation of business and data tier calls from web related work, internationalization, and presentation rendering. Good frameworks provide elegant, if not transparent, mechanisms for relieving the developer of these mundane tasks.

Provide architectural solution

While everyone can appreciate automation of tedious tasks, the structural features of frameworks are perhaps more important in the big scheme of things. The structure that the framework provides comes from the workflow abstractions made by the classes and interfaces of the framework itself. Being an action oriented framework, one of the key abstractions at the heart of the Struts 2 architecture is the *action*. We'll meet the others in a few pages. When you build an application on a framework, you are pretty much buying into that framework's architecture. Sometimes you can fight against the architectural imperative of the framework, but a framework should offer its proud architecture in a way that makes it hard to refuse. If the architecture of the framework is good, why not let your application gracefully inherit that architecture?

1.2.2 Why use a framework?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

Well, you don't have to. You do have a few alternatives. For starters, you could forgo a framework altogether. But unless your application is quite simple, we suspect that the work involved in rolling your own versions of all the common domain tasks, not to mention solving all the architectural problems on your own, will quickly deter you. As the 21st century ramps up, various new web application platforms boast light-speed development times and agile interfaces. In the world of Java web applications, using a sleek, new framework is the way to take advantage of these benefits.

If you want, you could roll your own framework. This is actually not a bad plan. It assumes a couple of things though. First, you have lots of really smart developers. Two, they have the time and money to spend on a big project that might seem off topic from the perspective of the business requirements. Even if you have the rare trinity of really smart people, time and money, there are still some drawbacks. I've done work for a company whose product is built on an in-house framework. The framework is not bad. But a couple of glaring points can't be overlooked. First, new developers will always have to learn the framework from the ground up. If you are using a mainstream framework, there's a trained work force waiting for you to hire them. Second, the in-house framework is unlikely to see elegant revisions that keep up with the pace of industry. In fact, in-house frameworks seem to be suspect to architectural erosion as the years pass and too many extensions are less elegantly tacked on than one would hope.

Ultimately, it's hard to imagine creating 21st century web applications without using a framework of some kind. If you have X amount of hours to spend on a project, you might as well spend them on higher level concerns than common workflow and infrastructural tasks. Perhaps it's not a question of whether to use a framework or not. Perhaps it's a question of which framework offers the solutions you need. With that in mind, it's time to look at Struts 2 itself and see what kinds of modern conveniences it offers.

1.3 The Struts 2 Framework

Struts 2 is a brand new, state of the art web application framework. As we said earlier, Struts 2 is not just a new release of the older Struts 1 framework. It is a completely new framework, based on the architecturally esteemed WebWork framework. By now you should be pretty tuned in to what a web application framework should offer. In terms of the common domain tasks, Struts 2 covers the domain quite well. It handles all of the tasks we have identified, and more. During the course of the book, you will learn how to work with the features that address each of those tasks in turn. At the introductory stage of things, it makes more sense to focus on the architectural aspects of the framework. So in this section, we will see how Struts 2 structures the web application workflow. In the next few sections, we will look at the design pattern roots of Struts 2, see how those roots influence the high level architecture, and take a slightly more detailed look at how the framework handles actual request processing.

1.3.1 A brief history

Struts 2 is a second generation web application framework that implements the Model-View-Controller (MVC) design pattern. Struts 2 is built from the ground up on best practices and proven, community accepted design patterns. This was true for the first version of Struts as well. In fact, one of the primary goals of the first Struts was the incorporation of the Model-View-Controller (MVC) pattern into a web application framework. This was, of course, a critical step in the evolution of well designed web applications as it provided the infrastructure for making the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

MVC separation of concerns an easily achieved goal. This allowed developers, with little resources for such architectural niceties, to tap into a ready made best practice solution. Struts 1 can certainly claim responsibility for many of the more well designed web applications of the last ten years.

At some point, the Struts community became aware of the limitations in the first framework. With such an active community, identification of the weak and inflexible points in the framework was not hard to accomplish. Struts 2 takes advantage of the many lessons learned to present a cleaner implementation of MVC. At the same time, it introduces several new architectural features that make the framework cleaner and more flexible. These new features include such things as interceptors for layering cross cutting concerns away from action logic, annotation based configuration to reduce classpath pollution, a powerful expression language OGNL that transverses the entire framework, and a mini-MVC based tag API that supports modifiable and re-usable UI Components. At this point, its pretty much impossible to do more than namedrop. We will have plenty of time in the book to fully explore each of these features. We really need to start with a high level overview of the framework. First, we'll look at how Struts 2 implements MVC. Then, we'll look at the actual parts of the framework work together when processing a request.

Pull-out: Teaching old dogs new tricks: moving from Struts 1 to Struts 2

Since we've stressed that Struts 2 is truly a new framework, you might be wondering how hard it will be to move from Struts 1 to Struts 2. There are certainly some things to learn, interceptors and OGNL in particular. But while this a new framework, it is still an action oriented MVC framework. The whole point of design patterns such as MVC is the reuse of solutions to common problems. Reusing solutions at the architectural level provides an easy transferal of experience and knowledge. If you have worked with Struts 1, you already understand the MVC way of doing things. Since Struts 2 is an improved implementation of the MVC pattern, we believe that Struts 1 developers will not only find it easy to migrate to Struts 2, they will find themselves saying, "That's how it always should have been done!"

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

1.3.2 Struts 2 from 30,000 Feet: the MVC Pattern

The high level design of Struts 2 follows the well established Model-View-Controller design pattern. In this section, we'll orient you to which parts of the framework address the various concerns of the MVC pattern. The MVC pattern provides a separation of concerns that applies well to the domain of web applications. Separation of concerns allows us to manage the complexity of large software systems by dividing them into high level components. The MVC design pattern identifies three distinct concerns: *model*, *view* and *controller*. In Struts 2 these concerns are implemented by the *action*, *result* and *FilterDispatcher*, respectively. Figure 1.3 shows the Struts 2 implementation of the MVC pattern to handle the workflow of web applications.

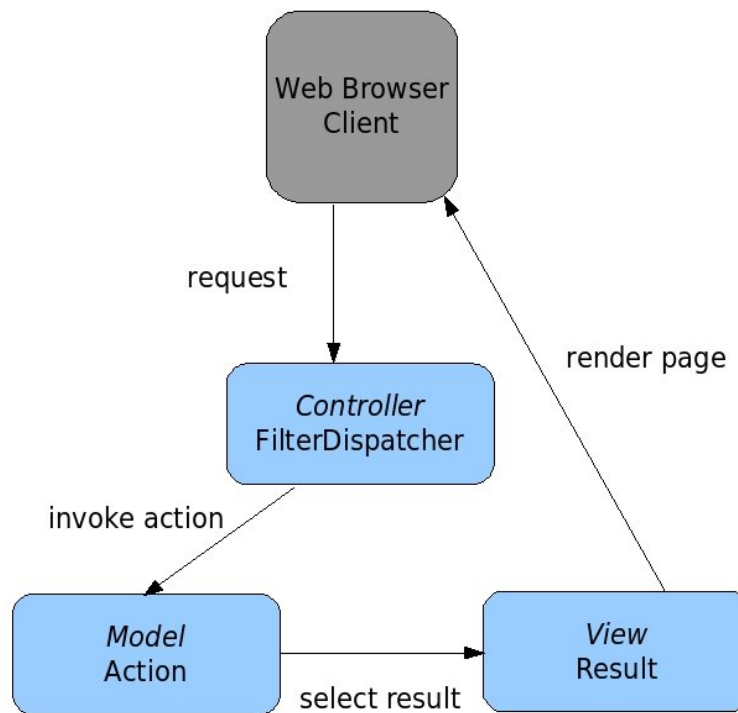


Figure 1.3 Struts 2 MVC is Realized by Three Core Framework Components: Actions, Results, and the FilterDispatcher

Let's take a close look at each part of Figure 1.3. We will now provide a brief description of the duties of each MVC concern and look at how the corresponding Struts 2 component fulfills those duties.

Controller – FilterDispatcher

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

We will forgo the lexical ordering of the MVC name and start with the controller. It just seems to make more sense to start with the controller when talking about web applications. In fact, the MVC variant used in Struts is often referred to as a *front controller* MVC. This basically means that the controller is out front and is the first component to act in the processing. You can easily see this in Figure 1.3. The controller's job is to map requests to actions. In a web application, the incoming HTTP requests can each be thought of as commands that the user issues to the application. One of the more fundamental tasks of a web application is the routing of these requests to the appropriate set actions that should be taken within the application itself. This controller's job is like that of a traffic cop, or air traffic controller. In some ways, this work is pretty administrative. It's certainly not part of your core business logic.

The role of the controller is played by the Struts 2 `FilterDispatcher`. This important object is a servlet filter that inspects each incoming request to determine which Struts 2 action should handle the request. The framework pretty much handles all of the controller work for you. You just need to inform the framework which request URL's map to which of your actions. You can do this with XML based configuration files, or with Java annotations. We will demonstrate both of these methods

Pull-out: Struts 2 goes along way towards a goal of *zero-configuration* web applications. Zero-configuration aims at deriving all of an application's meta-data, such as which URL maps to which action, from convention rather than configuration. The use of Java annotations play an important role in this zero-configuration scheme. While zero-configuration has not quite achieved its goal, you can currently use annotations and convention to drastically reduce XML based configuration.

in the next chapter's HelloWorld application, a simple application that will demonstrate both the general architecture and deployment details of Struts 2 web applications.

Model – Action

Looking at Figure 1.3, it's easy to see that the model is implemented by the Struts 2 action component. But what exactly is the model? I personally find the model the most nebulous of the MVC triad. In some ways, the model is a black box that contains the entire guts of the application. Everything else is just user interface, and wiring. The model is the thing itself. In more technical terms, the model is the internal state of the application. This state is composed of both the data model and the business logic. From the high level black box view, the data and the business logic kind of merge together into the monolithic *state* of the application. For instance, if you are logging in to an application, we know that both business logic and data from the database will be involved in the authentication process. Most likely, the business logic will provide an authentication method that will take the username and password and verify them against some persisted data from the database. In this case, the data and the business logic combine together to form one of two states, “authenticated” or “unauthenticated.” The data on its own, nor the business logic on its own, can produce these states.

Bearing all of this in mind, a Struts 2 action serves two roles. First, an action is an encapsulation of the calls to business logic into a single unit of work. Second, the action serves as a locus of data transfer. It's a bit early to go into details on this, but we will certainly treat this in great depth during the course of this book. At this point, consider that an application has any number of actions to handle whatever set of commands it exposes to the client. As seen in Figure 1.3, the controller, after receiving the request, must consult its mappings and determine which of

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

these actions should handle the request. Once it finds the appropriate action, the controller hands over control of the request processing to the action by invoking it. This invocation process, conducted by the framework, will both prepare the necessary data and execute the action's business logic. When the action completes its work, it will forward to the Struts 2 view component, the result – which we will now consider.

View – Result

The view is the presentation component of the MVC pattern. Looking back at Figure 1.3, we see that the result returns the page to the web browser. This page is the user interface which presents a representation of the application's state to the user. These are commonly JSP pages, Velocity templates or some other presentation layer technology. While there are many choices for the view, the role of the view is pretty clear cut and simple; it translates the state of the application into a visual presentation with which the user can interact. With rich clients and Ajax applications increasingly complicating the details of the view, it becomes even more important to have clean MVC separation of concerns. Good MVC lays the ground work for easy management of the most complex front end.

Pull-out: One of the interesting aspects of Struts 2 is how well its clean architecture paves the way for new technologies and techniques. The Struts 2 result component is a good demonstration of this. The result provides a clean encapsulation of the processing of handing off control of the processing to another object that will write the response to the client. This makes it very easy for alternative responses, such as XML Ajax snippets or XSLT transformations, to be integrated into the framework.

If you look back to Figure 1.3, you can see that the action is responsible for choosing which result will render the response. The action can choose from any number of results with which it might have been associated. Common choices are between results that represent the semantic outcomes of the action's processing, such as 'success' and 'error'. Struts 2 provides out of the box support for using most common view layer technologies as results. These include JSP, Velocity, FreeMarker, and XSLT . Better yet, the clean structure of the architecture insures that more result types can be easily built to handle new types of responses.

Now that we've met the MVC components of the framework, you should probably have a general idea of how Struts 2 handles the workflow of a web application. Now, we'll move in for a closer examination of exactly how the framework processes a request.

1.3.3 How Struts 2 Works

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now that we know what the MVC parts of the framework are, let's look at something more practical. In this section, we'll detail actual processing of a request within the framework. As you will soon see, the framework has more than just its MVC components. We said that Struts 2 provides a cleaner implementation of MVC. These clean lines are only possible with the help of a few other key architectural components that participate in the processing of each and every request. Chief among these are the *interceptors*, *OGNL* and the *ValueStack*. We'll learn what each of these does in the following walk-through of Struts 2 request processing. Figure 1.4 shows the request processing workflow.

Figure 1.4 Struts 2 Request Processing Uses Interceptors that Fire Before and After the

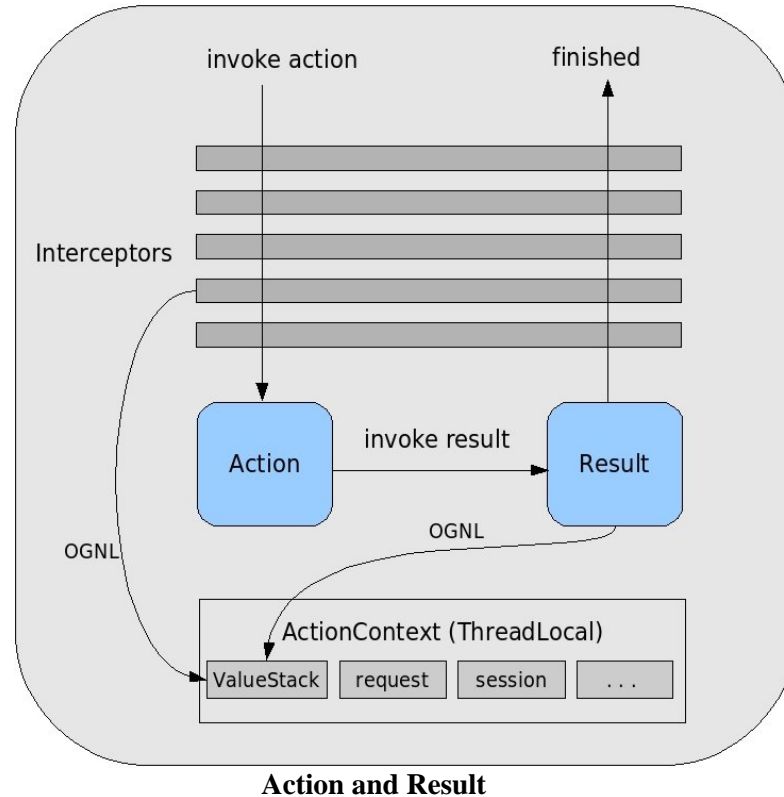


Figure 1.4 assumes that the *FilterDispatcher* has already done its controller work. At this point, the framework has already selected an action and has initiated the invocation process. The first thing we should consider is that the workflow of this diagram still obeys the simpler MVC view of the framework that we saw a bit earlier. An action has been selected by the controller, it will execute, and then select the appropriate result. The rest of the stuff mostly serves to make that basic MVC workflow cleaner.

Figure 1.4 introduces the following Struts 2 components: interceptors, the *ValueStack* and *OGNL*. This diagram, though perhaps a bit busy at first glance, goes a long way towards showing what really happens in Struts 2. It wouldn't be incorrect to say that everything we will discuss in this book is shown in this diagram. As interceptors come first in the request processing cycle, we'll start with them. The name seems obvious, but what exactly do they intercept?

Interceptors

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

You may have noticed, while studying Figure 1.4, that there is a stack of interceptors in front of the action. This is a key part of the Struts 2 framework. We will devote an entire chapter to this important component later in the book. At this time, it is enough to understand that every action will have a stack of interceptors associated with it. These interceptors are invoked both before and after the action. Note in Figure 1.4 that the interceptors actually fire before the action and after the result. Also note that interceptors don't necessarily have to do something both times they fire; sometimes the control merely pass through them. Some interceptors only do work before the action has been executed, and others only do work afterwards. It depends on their work. The important thing is that the interceptor allows common, cross cutting tasks to be defined in clean, re-usable components that you can keep separate from your action code.

Definition: Interceptors are Struts 2 components that execute both before and after the rest of the request processing. They provide an architectural component in which to define various workflow and crosscutting tasks so that they can be easily reused as well as separated from other architectural concerns.

What kinds of work should be done in interceptors? Logging is a good example. Logging is something that should be done with the invocation of every action, but it probably shouldn't be put in the action itself. Why? Because its not really part of the action's own unit of work. It's more administrative, overhead if you will. Earlier, we charged a framework with the responsibility of providing built-in functional solutions to common domain tasks such as data validation, type conversion and file uploads. Struts 2 uses interceptors to do this type of work. While these tasks are important, they are not specifically related to the action logic of the request. Struts 2 uses interceptors to both separate and reuse these crosscutting concerns. Interceptors play a huge role in the Struts 2 framework. And while you probably won't spend a large percentage of your time writing interceptors, most developers will eventually find that many tasks are perfectly solved with custom interceptors. As we said, we will devote all of Chapter 4 Interceptors in Action to exploring this core component.

The ValueStack and OGNL

While interceptors may not absorb a lot of your daily development energies, the ValueStack and OGNL will be constantly on your mind. In a nutshell, the ValueStack is simply a storage area that holds all of the application domain data associated with the processing of a request. You could think of it as a piece of scratch paper where the framework does its data work while solving the problems of request processing. In short, OGNL is an expression language that allows you to reference and manipulate the data on the ValueStack. Developers new to Struts 2 probably ask more questions about the ValueStack and OGNL than any thing else. If you are coming from Struts 1, these are certainly a couple of the more exotic features of the new framework. Due to this, and the sheer importance of this duo, we will carefully treat them throughout the book. In particular, Chapter Five and Chapter Six describe the detailed function of these two framework components.

Definition: Struts 2 uses the ValueStack as a storage area for all application domain data that will be needed during the processing of a request. Data is moved to the ValueStack in preparation of request processing, it is manipulated there during action execution, and it is read from there when the results render their response pages.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

The tricky, and powerful, thing about the ValueStack and OGNL is that they don't belong to any of the individual framework components. In Figure 1.4, note that both interceptors and results can use OGNL to target values on the ValueStack. The data in the ValueStack follows the request processing through all phases; it slices through the whole length of the framework. It can do this because it is stored in a ThreadLocal context called the ActionContext.

Definition: OGNL is a powerful expression language, and more, that is used to reference and manipulate properties on the ValueStack.

The ActionContext contains all of the data that makes up the context in which an action occurs. This includes the ValueStack obviously, but it also includes stuff the framework itself will use internally, like the request, session, and application maps from the Servlet API. You can definitely access these objects yourself if you like, and we'll see how later in the book. For now, we just want to focus on the ActionContext as the ThreadLocal home of the ValueStack. The use of ThreadLocal makes the ActionContext, and thus the ValueStack, accessible from anywhere in the same thread of execution. Since Struts 2's processing of each request occurs in a single thread, the ValueStack is available from any point in the framework's handling of a request.

Typically, it's considered bad form to obtain the contents of the ActionContext yourself. The framework provides many elegant ways to interact with that data without actually touching the ActionContext, or the ValueStack, yourself. Primarily, you will use OGNL to do this. OGNL is used in many places in the framework to reference and manipulate data in the ValueStack. For instance, you will use OGNL to bind HTML form fields to data objects on the ValueStack for data transfer, and you will use OGNL to pull data into the rendering of your JSP's and other result types. At this point, you just need to understand that the ValueStack is where your data is stored while you work with it, and that OGNL is the expression language that you, and the framework, use to target this data from various parts of the request processing cycle.

Now you've seen how Struts 2 implements MVC, and you've got a brief introduction to all the other important players in the processing of actual requests. The next thing we need to do, before getting down to the nuts and bolts of the framework's core components, is to make all of this concrete with a simple HelloWorld application of Chapter Two. But first, a quick summary.

1.4 Summary

We started with a lot of abstract stuff about frameworks and design patterns, but you should now have a fairly good high level understanding of the Struts 2 architecture. If abstraction is not to your taste, you'll be happy to know that we've officially completed the theoretical portion of the book. Starting immediately with Chapter Two's HelloWorld application, the rest of the book will deal with only the concrete, practical matters of building web applications. But before we move on, let's take a moment to review the things we've learned.

We should probably spend a moment or two to evaluate Struts 2 as a framework. Based upon our understanding of the technological context and the common domain tasks, we laid out two responsibilities for a web application framework at the outset of this chapter. First, we said that a framework should provide an architectural foundation for web applications. We've seen that Struts 2 certainly does this, and we discussed the design pattern roots that inform the Struts 2 architectural decisions. In particular, we have seen that Struts 2 takes the lessons learned from first generation web application frameworks to implement a brand new, cleaner MVC based

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

framework. We also met the specific framework components that implement the MVC pattern: the *action* component, the *result* component, and the *FilterDispatcher*.

The other responsibility with which we charged web application frameworks was the automation of many common tasks of the web application domain. These tasks are sometimes referred to as cross cutting concerns because they occur again and again across the execution of a disparate set of application specific actions. Logging, data validation, and other common, cross cutting concerns should be separated from the concerns of the action and result. In Struts 2, the interceptor provides an architectural mechanism for removing cross cutting concerns from the core MVC components. As we go further into the book, you will realize that the framework comes with many built-in interceptors to handle all the common tasks of the domain. Though you will rarely, if ever, need to write an interceptor yourself, you will soon realize that they do most of the work of the framework.

We also took a high level look at the actual request processing of the framework. We saw that each action has a stack of interceptors that fire both before and after the action and result have done their work. In addition to the MVC components and the interceptors, the ValueStack and the OGNL expression language play critical roles in the storage and manipulation of data within the framework. Hopefully, you have a decent grasp of what the framework does. In the next chapter's HelloWorld application, you will see a concrete example of the framework components in action. Once we get the Chapter Two and the HelloWorld application behind us, we will be moving on to explore the core components of the framework and start building our Struts 2 Portfolio sample application.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

2 Saying Hello to Struts 2

In the first chapter, we acquainted ourselves with the web application domain, learned a bit about how design patterns and frameworks help developers do their job, and conducted a quick, high level survey of the Struts 2 architecture and request processing pipeline. As we have indicated, we have now finished the abstract portion of this book. Congratulations. This chapter, which concludes the introductory section of this book, provides the practical and concrete details to materialize the concepts from the first chapter. In particular, this chapter will demonstrate the basic Struts 2 architectural components with the HelloWorld sample application. This application does not intend to demonstrate the full complexity of the framework. As we have said, we will develop a full featured sample application through the course of the book – the Struts 2 Portfolio application. The purpose of the HelloWorld application is just to get a Struts 2 application up and running.

But before we get to the HelloWorld application, we need to look at the fundamentals of configuring a Struts 2 application. In particular, we will introduce at a type of configuration known as *declarative architecture*.

2.1 Declarative Architecture

In this book, we use the phrase declarative architecture to clarify the mechanisms by which one assembles applications in the Struts 2 framework. This is a special type of configuration that allows developers to describe their application architecture at a higher level than direct programmatic manipulation. Similar to the manner in which an HTML document simply describes its components and leaves the creation of their run time instances to the browser, Struts 2 allows you to describe your architectural components through its high level declarative architecture facility and leave the run time creation of your application to the framework itself. In this section, we'll see how this works.

2.1.1 Two kinds of configuration

First, we need to make a clarification regarding some terminology. In this introduction to this chapter, we referred to the act of declaring your application's Struts 2 components as *configuration*. While there is nothing wrong with this nomenclature, we think it can be a bit confusing at times. Hidden beneath the far reaching concept of configuration, we can actually distinguish between two distinct sets of activity that occur in a Struts 2 project. One of these sets of activity, the declarative architecture, is more central to the actual building of Struts 2 applications, while the other is a bit more administrative in nature. Conceptually, it's important to distinguish between the two.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Configuring the framework itself

First, we have configuration in the traditional sense of the word. These are the more administrative activities. Because the Struts 2 framework believes in flexibility, it allows you to tweak its behavior in many areas. If you want to change the URL extension by which the framework recognizes the requests it should handle, you can easily configure the framework to recognize some other extension. By default, Struts 2 looks for URL's ending in `.action`, but, for example, you could configure the framework to look for `.do` (the Struts 1.x default extension, of course). Other examples of configurable parameters include maximum file upload size and the development mode flag. Due to its administrative nature, we will leave this type of configuration for later in the book. For now, we will focus on how to build web applications.

Declaring your application's architecture

The more important type of configuration, which we will refer to as declarative architecture, involves defining the Struts 2 components that your application will use and linking them together, a.k.a. wiring them, to form your required workflow paths. As we have seen, the Struts 2 framework provides a clean architectural foundation for web applications to extend.

Definition: Declarative architecture is a specialized type of configuration that allows a developer to create an application's architecture through description rather than programmatic intervention. The developer describes the architectural components in high level artifacts, such as XML files or Java annotations, from which the system will create the run time instance of the application.

The developer needs only to declare which objects will serve as the actions, results and interceptors of their application. This process of declaration is primarily consists of specifying which Java class implements the necessary interface; almost all of the Struts 2 architectural components are defined as interfaces. In reality, the framework provides implementations for nearly all of the components you will ever need to use. For instance, the framework comes with implementations of results to handle many types of view layer technologies. Typically, a developer will only need to implement actions and wire them to built-in results and interceptors. Furthermore, the use of intelligent defaults and annotations can further reduce the manual tasks needed in this area.

2.1.2 Two mechanisms for declaring your architecture

Now we will look at the nuts and bolts of declaring your architecture. The first thing we need to say is that there are two ways to do this. You can declare your architecture through XML based configuration files, or through Java annotations. Figure 2.1 demonstrates the dual interface to the declarative architecture

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

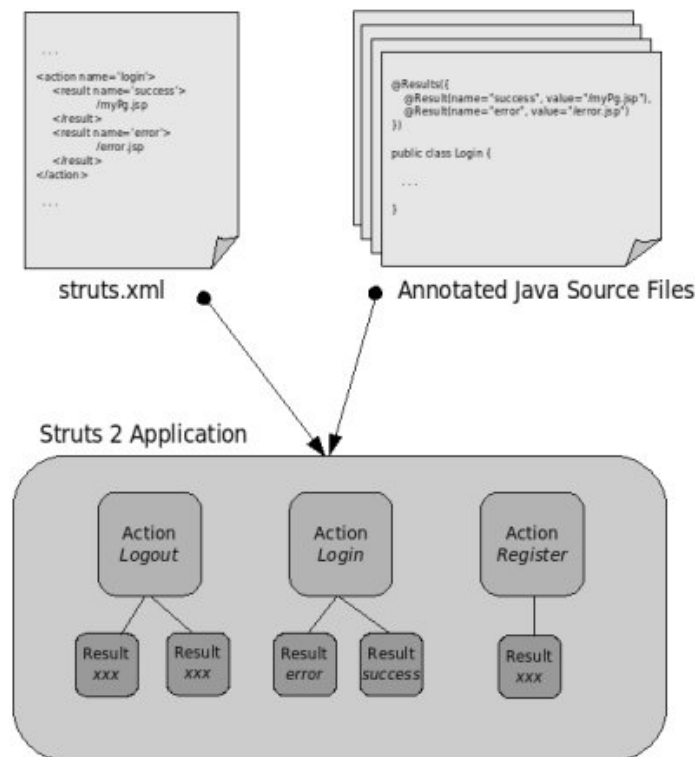


FIGURE 2.1 Declaring your Struts 2 Application Architectural with XML or Annotations

As you can see, whether your application's Struts 2 components are declared in XML or in annotations, the framework translates them both into the same runtime components. In the case of the XML, we have the familiar XML configuration document with elements describing your applications actions, results and interceptors. With annotations, the XML is gone. Now, the meta-data is collected in Java annotations that reside directly with in the Java source for the classes that implement your actions. Regardless of which method you use, the framework produces the exact same run time application. The two mechanisms are redundant in the sense that you can use whichever you like without functional consequence. The declarative architecture is the real concept here. Whichever style of declaration you choose is largely a matter of taste. For now, let's meet the candidates and see how each works.

XML based declarative architecture

Many of you will already be familiar with the use of XML for declarative software development. Struts 2 allows you to use XML files to describe your application's desired Struts 2 architectural components. In general, the XML documents will consist of elements that represent the components of the application. Listing 2.1 shows an example of some XML elements that declare some actions and results. We won't go into the details

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Listing 2.1 Example of XML Declarative Architecture Elements

```
<action name="Login" class="manning.Login">
  <result>/AccountPage.jsp</result>
  <result name="input">/Login.jsp</result>
</action>

<action name="Registration" >
  <result>/Registration.jsp</result>
</action>

<action name="Register" class="manning.Register">
  <result>/RegistrationSuccess.jsp</result>
  <result name="input">/Registration.jsp</result>
</action>
```

of these elements at this time. We just show this as an example of what the XML style declarative architecture looks like. Typically, an application will have several XML files containing elements like these that describe all of the components of the application.

Even though most applications will have more than one XML file, all of the files work together as one large description. The framework uses a specific file as the entry point into this large description. This entry point is the `struts.xml` file. This file, which resides on the Java classpath, must be created by the developer. While its possible to declare all of your components in `struts.xml`, developers, as we have indicated, more commonly use this file only to include secondary XML files in order to modularize their applications.

We will see XML based declarative architecture in action when we look at the HelloWorld application in a few moments.

Java annotations based declarative architecture

A relatively new feature of the Java language, annotations allow you to add meta-data directly to Java source files. One of the loftier goals of Java annotations is support for higher level tools that can read meta-data from a Java class and do something useful with that information. Struts 2 uses Java annotations in this way. If you don't want to use XML files, the declarative architecture mechanism can be configured to scan Java classes for Struts 2 related annotations. Listing 2.2 shows what these annotations look like

Listing 2.2 Example of Using Annotations for Declarative Architecture

```
@Results({
    @Result(name="input", value="/RegistrationSuccess.jsp" )
    @Result(value="/RegistrationSuccess.jsp" )
})

public class Login {

    public string execute() {

        //Business logic for login
    }
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```
    }  
}
```

Note, the annotations are made on the Java classes that implement the actions. Listing 2.2, shows the code from the Login class, which itself will serve as the Login action. Just like their counterpart elements in the XML, these annotations contain meta-data that the framework uses to create the runtime components of your application.

While we will fully explain this material later in the book, it might be worthwhile to note the relationship between the Login action's XML element in Listing 2.1 and the annotations of Listing 2.2, which are made directly on the Login.java source itself. The annotation based mechanism is considered a more elegant solution than the XML mechanism. For one thing, some of the information that must be specified in the XML elements can be deduced automatically from the Java package structure to which the annotated classes belong. For instance, you obviously don't need to specify the name of the Java class as that is clearly implicit in the physical location of the annotations. Many developers also appreciate how annotations eliminate some of the XML file clutter that seems to increase year by year on the web application classpath. We will demonstrate the fundamentals of annotation based declarative architecture in the second version of the HelloWorld application provided later in this chapter.

Which method should you use?

Ultimately, choosing an mechanism for declaring your architecture is up to the developer. The most important thing is to understand the concepts of the Struts 2 declarative architecture. If you understand that, moving between the XML or Java annotations based mechanisms should be quite trivial. For the purposes of our book, we will use XML in our sample applications. We choose to do this because the XML file is probably more familiar to many of our readers and, more importantly, because an XML file provides a more centralized collection of the relevant material. This makes it easier to study the material when one is first learning the framework. We think, however, that many of you will ultimately choose to use Java annotations to declare your application's components because of their elegance. A reference to the details of the annotation based mechanism will be provided in the Appendix A.

2.1.3 Intelligent Defaults

Many commonly used Struts 2 components, or attributes of components, do not need to be declared by the developer. Regardless of which style of declaration you choose, these components and attribute settings are already declared by the framework so that you can more quickly implement the most common portions of application functionality. Some framework components, such as interceptors and result types, may never need to be directly declared by the developer because those provided by the system handle the daily requirements of most developers quite well. Other components, such as actions and results, will still need to be declared by the developer, but many common attribute settings can still be inherited from framework defaults.

Definition: Intelligent defaults provide out of the box components that solve common domain workflows without requiring further configuration by the developer, allowing the most common application tasks to be realized with a minimum of development.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

These predefined components are part of the Struts 2 intelligent defaults. In case you are interested, many of these components are declared in `struts-default.xml`, found in the `struts2-core.jar`. This file uses the XML style of declaration to declare an entire package of intelligent default components, the `struts-default` package. Starting with the upcoming HelloWorld application, and continuing through the rest of the book, we will learn how to take advantage of the components offered in this default package.

2.2 A Quick Hello

Now we will present the HelloWorld applications, one with XML and one with annotations, that will bring all of this to life. First, we will introduce the XML version of the application. We will both explore the use of the XML as well as discuss, from a high level, how the HelloWorld demonstrates the Struts 2 architecture. We will also introduce the basic layout of a Struts 2 application. Then we will revisit the very same application implemented with annotations focusing on the annotations themselves. As we have said, the two styles of declaring your architecture are just two interfaces to the very same declarative architecture. Which one you choose has no functional bearing on your Struts 2 application. The two HelloWorld applications, which differ only in the style of architectural declaration, will make this point concrete.

2.2.1 Why bother with hellos?

In short, we say hello to each other to get a quick idea of who we are. We know that some people believe that a trivial introduction provides nothing of worth, with humans as well as web application frameworks, but we tend to disagree. Though a brief introduction can inherently contain little in-depth material, we believe a properly brief introduction lays the foundation for a more rapidly successful subsequent relationship. In short, we think we have prepared a very simple Struts 2 version of HelloWorld that, while teaching you nothing profound, will help you materialize the abstract discussions of this first chapter as well as soften the learning curve for the upcoming chapters. We will have plenty of time to explore non-trivial examples while we develop the more complex Struts 2 Portfolio sample application throughout the course of the book.

2.2.2 Deploying the sample application

Well, you need a servlet container. Sounds simple enough, but it's not. As authors of this book, we find this a troublesome question. Since servlet containers are built to the Servlet Specification, it doesn't matter which one you use. In short, you just need to deploy the sample applications on a servlet containers. It's your choice. Some books attempt to walk you through the installation details for a specific container. These books intend to smooth the path to the true material of their book. The problem with this is that it somehow never seems to be quite as simple as they make it sound.

Furthermore, we think the benefits gained from learning to install a servlet container far outweigh any short term gains to be had from any container specific quick start we might try to provide. If you are experienced with Java web application development, you will already have your own container preferences and certainly know how to deploy a web application in your chosen container. If you are new to Java web application development, you can probably expect

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

to spend a few hours reading some online documentation and working through the installation process. Deployment of a web application on a running container is typically point-and-click simple. Choosing a servlet container can be a bit overwhelming, but for newbies we recommend Apache Tomcat. It is the open source reference implementation of the Servlet Specification. It's both easily obtained and certain to be as specified.

Though perhaps less fundamental than the choice of a servlet container, choosing an IDE and a build tool can be just as important. Our goal is to provide build and IDE agnostic example applications. We recognize that we might save you some time by providing an Ant build file with Tomcat targets, for instance, but if you don't use Ant and Tomcat, that doesn't help and may even hinder your progress. We should note that the Struts 2 community, along with the largest portion of the Java open source community, has made a strong adoption of Maven as their build / project management tool. If you plan to have more than a fleeting relationship with the Struts 2 source code, then a working knowledge of Maven practice would serve you well.

Provided you have a servlet container, the only thing left to do is deploy the sample application WAR file in accordance with the requirements of your container. You can obtain the sample application from the Manning web site??? All of the sample code from this book is contained within a single Struts 2 web application. This application is packaged in the `manningSampleApp.war` file. Once you have deployed this web application to your container, point your browser to <http://localhost:8080/manningSampleApp/Menu.action> to see the main menu for the sample code. Note, this browser assumes that the sample application has been deployed on your local machine and that the servlet container is listening on port 8080. Figure 2.2 shows this menu.

-
- [HelloWorld](#)
 - [AnnotatedHelloWorld](#)
 - [Struts 2 Portfolio \(Chapter 3\)](#)
 - [Struts 2 Portfolio \(Chapter 4\)](#)
 - [Struts 2 Portfolio \(Chapter 5\)](#)
 - [Struts 2 Portfolio \(Chapter 6\)](#)
 - [Struts 2 Portfolio \(Chapter 7\)](#)
 - [Struts 2 Portfolio \(Chapter 8\)](#)
-

Figure 2.2 The Sample Application is Organized into Several Mini-Applications

As you can see from Figure 2.2, the sample application has been organized into a series of mini-applications. Basically, we have two versions of the HelloWorld application, and then many versions of the Struts 2 Portfolio application. Technically, all of these are just one big Struts 2 application. However, the flexibility of the framework allows us to cleanly modularize the sample code for all of the chapters so that we can present distinct versions of the application for each chapter. This allows us to, for instance, present a simple version of the Struts 2 Portfolio while covering the basics in early chapters, then provide a full featured version for later chapters.

The Layout of a Struts 2 Web Application

The entire `manningSampleApp.war` can be used as a template for understanding what is required of a Struts 2 web application. Most of the requirements for the structure of the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

application come from the requirements put on all web applications by the Servlet API. Figure 2.3 shows the exploded directory structure of the manningSampleApp.war file. Again, if

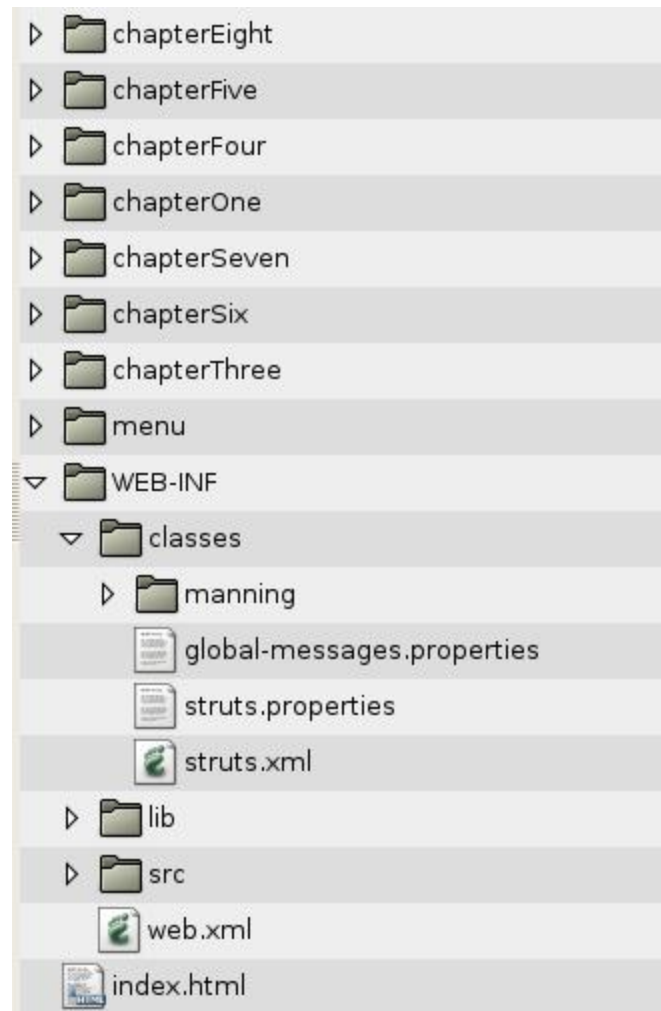


Figure 2.3 The Exploded Directory Structure of our Struts 2 Sample Application

you aren't familiar with the Servlet Specification, a quick read might be worth your while. For our purposes, we will quickly outline the most important aspects. First, all of the top level directories, except WEB-INF, are in the document root of the web application. Typically, this is where our JSP files will go. You can also put Velocity and FreeMarker templates here, as we will do, but those resources can also load from JAR files on the classpath. In the sample application, we have organized our JSP's according to the chapter based version of the sample application to which they belong. One important thing to note about the document root is that these resources can potentially be served as static resources by the servlet container. If not configured to prevent such access, a URL that directly points to resources in the document root can result in the servlet container's spitting out that resource. Because of this, the document root is not considered a secure place for sensitive material.

All of the **important** stuff goes in WEB-INF. As you can see in Figure 2.3, the top level contents of WEB-INF include two directories, lib and classes, and the file web.xml. Note, there's also a directory called src, but that it is our project source code. This is not a required part

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

of the web application directory structure. We have put it here for convenience. You could put it anywhere, depending upon the details of your build process. Ultimately, you will most likely not want source code in a production ready web application. We have done it this way as a convenient, build methodology agnostic alternative.

As for the other two directories, they are quite essential. The lib directory holds all of the JAR file dependencies that your application needs. In the sample application, we have placed all of the JAR's commonly used by Struts 2 web applications. Note, Struts 2 is very flexible and if you add some features that we don't use in this book, you might need to add additional JAR's. The classes directory holds all of the Java classes that your application will use. These are essentially no different than the resources in the lib directory, but the classes directory contains an exploded directory structure containing the class files, no JAR's. In Figure 2.3 you can see that the classes directory holds one directory, manning, which is the root of our applications Java package structure, and it holds several other classpath resource files, such as properties files and the struts.xml file we have already discussed.

In addition to the lib and classes directories, WEB-INF also contains the central configuration file of all web applications, web.xml. This file, formally known as the deployment descriptor, contains definitions of all of the servlets, servlet filters, and other Servlet API components contained in this web application. Listing 2.3 shows the web.xml file of our sample

Listing 2.3 The web.xml Deployment Descriptor of Our Struts 2 Sample Application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app      version="2.4"      xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>S2      Example      Application      -      Chapter      1      -      Hello
World</display-name>

  <filter>
    <filter-name>struts2</filter-name>      ANNOTATION 1
    <filter-
class>org.apache.struts2.dispatcher.FilterDispatcher</filter-
class>
    <init-param>      ANNOTATION 2
      <param-name>actionPackages</param-name>
      <param-value>manning</param-value>
    </init-param>
  </filter>

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

  <servlet>      ANNOTATION 3
    <servlet-name>anotherServlet</servlet-name>
    <servlet-class>manning.servlet.AnotherServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>anotherServlet</servlet-name>
    <url-pattern>/anotherServlet</url-pattern>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>


```

</servlet-mapping>

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

</web-app>
(annotation) <#1"The FilterDispatcher: Struts 2 Begins Here">
(annotation) <#2"Tell Struts 2 Where to Find Annotations">
(annotation) <#3"A Servlet Outside of Struts 2">

```

application. For a Struts 2 application, the most important element in this deployment descriptor is the filter and filter-mapping element that set up the Struts 2 FilterDispatcher. This servlet filter is, basically, the Struts 2 Framework. This filter will examine all the incoming requests looking for requests targeting Struts 2 actions. Note the URL pattern to which this filter is mapped – `/*`. This filter will inspect all requests. The other important thing about the configuration of the FilterDispatcher is the initialization parameter that we pass in. The `actionPackages` parameter is necessary if you are going to use annotations in your application. It tells the framework which packages to scan for annotations. We'll see more about this when we get to the annotated version of HelloWorld in a few pages.

One other interesting thing to note is that we have included a non-Struts 2 servlet in our web application. As we said earlier, a web application is defined as a group of servlets packaged together. Many Struts 2 web applications will not actually have any other servlets in them. In fact, since Struts 2 actually uses a servlet filter, rather than a servlet, many Struts 2 applications will not have any servlets in them – unless you count compiled JSP's of course. Since it's not uncommon to integrate other servlets with the framework, we have included another servlet, as seen in Listing 2.3, in our web application. We will demonstrate actually using this servlet later in the book.

Now, you should know how to get a skeletal Struts 2 application set up. Everything in our sample application is pretty much by the book except for the presence of the source directory in `WEB-INF`. As we said, this has been done as a convenience for the purposes of the book. You'll probably want to structure your build according one of the industry best practices. We haven't provided such a build because we think it only complicates the learning curve. Now, it's time to look at the HelloWorld application.

2.2.3 Exploring the HelloWorld application

The HelloWorld application aims to provide the simplest possible introduction to Struts 2. However, it also tries to exercise all of the core Struts 2 architectural components. The application has a very simple work flow. It will collect a user's name from a form, use that data to build a custom greeting for the user, and then present the user with a web page that displays the customized greeting. This work flow, while ultra-simple, will clearly demonstrate the function and usage of all the Struts 2 components, such as actions, results and interceptors. Additionally, it will demonstrate the mechanics of how data flows through the framework, including the ValueStack and OGNL. As Struts 2 is a rather sophisticated framework, we will be limited to a high level view. Rest assured that the rest of the book will spend adequate time on each of these topics.

HelloWorld user guide

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

First, let's have a look at what the HelloWorld application actually does. Provided you have deployed the application to your servlet container, select the HelloWorld link from the menu we saw earlier. Note, we are starting with the XML version, not the annotated version. You will be presented with a very simple form, seen in Figure 2.4, that asks for your name.

Enter your name so that we can customize a greeting just for you!

Your name:

Figure 2.4 – The First Page Collects the User's Name

Enter your name and hit the submit button. The application will prepare your customized greeting and return a new page for you, as seen in Figure 2.5.

Custom Greeting Page

Hello Charlie Joe

Figure 2.5 – The Second Page Presents the Customized Greeting, Built from the Submitted Name

The above figure shows the customized greeting message built by the action and displayed via a JSP page. That's it. Now, let's see what the Struts 2 architecture of this simple application looks like.

HelloWorld Details

We'll begin by looking at the architectural components used by HelloWorld. This version of HelloWorld uses XML to declare its architecture. As we have said, the entry point into the XML declarative architecture is the struts.xml file. We have also said that many developers usually use this root document to include other XML documents, allowing for modularization. We have done this for the sample application, modularizing on the basis of chapters. Listing 2.4 shows WEB-INF/classes/struts.xml, the most important aspect of which are the include elements that pull in the modularized XML documents.

Listing 2.4 The struts.xml File Serves as the Entry Point into XML Based Declarative Architecture

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

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

<struts>

    <constant name="struts.devMode" value="true" />    ANNOTATION
#1

    <package      name="default"      namespace="/"      extends="struts-
default">
        <action name="Menu">    ANNOTATION
#2

            <result>/menu/Menu.jsp</result>
        </action>
    </package>

    <include file="manning/chapterTwo/chapterTwo.xml"/>    ANNOTATION
#3

    . . .

    <include file="manning/chapterEight/chapterEight.xml"/>

</struts>
(annotation) <#1"Use Constants To Tweak Struts 2 Properties">
(annotation) <#2"Menu Action Belongs to a Default Package">
(annotation) <#3"Include Modularized XML Docs">

```

Though a bit off topic, we should note that the constant element can be used to set framework properties. Here we set the framework to run in development mode. You can also do this with property files, as we will see later. Also off topic, we should note that struts.xml is a good place to define some global actions in a default package. Since our main menu doesn't really belong to any of our modularized mini-applications, we place it here. Finally back on topic, we see the most important aspect of the struts.xml file, a long list of includes that pull all of our chapter based XML documents into the declarative architecture. All of these files will basically be pulled into this main document, in line, to create a single large XML document.

The HelloWorld application belongs to the Chapter Two module of sample code. Listing 2.5 shows the contents of WEB-INF/classes/manning/chapterTwo/chapterTwo.xml. Note, the real

Listing 2.5 The First Version of HelloWorld Uses XML for its Declarative Architecture

```

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

<struts>

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

<package                name="chapterTwo"          namespace="/chapterTwo"
extends="struts-         default">

    <action name="Name">
        <result>/chapterTwo/NameCollector.jsp</result>
    </action>

    <action                name="HelloWorld"
class="manning.chapterTwo.HelloWorld">
        <result name="SUCCESS">/chapterTwo/HelloWorld.jsp</result>
        <result name="ERROR">/chapterTwo/Error.jsp</result>
    </action>

</package>

</struts>

```

file in the application contains detailed comments about the various elements; this will be true for all the examples in this book, but when we print listings here we will remove the comments for clarity. This simple application has only two *actions*, and one of them hardly does anything at all. Both the Name and the HelloWorld action declare some results for their own use. Each result names a JSP page that it will use to render the result page. The only other elements here are the struts document root element and the package element. The struts element is the mandatory document root of all Struts 2 XML files and the package element is an important container element for organizing your actions, results, and other component elements.

For now, the only thing we need to note about the package element is that it declares a namespace that will be used when the framework maps URL's to these actions. Figure 2.6 shows how the namespace of the package is used to determine the URL that maps to our

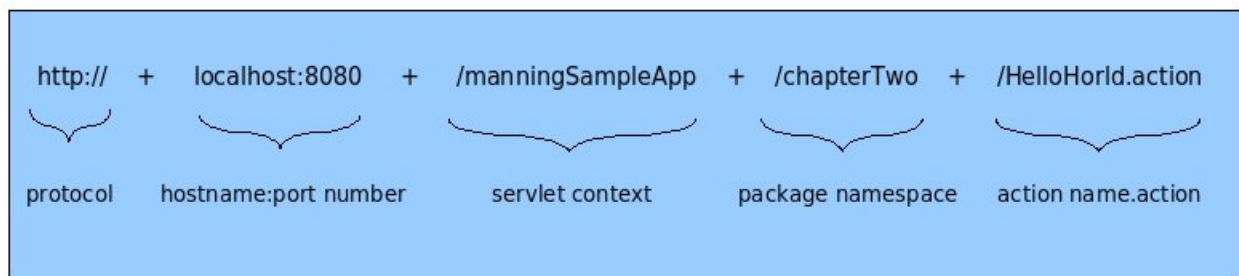


Figure 2.6 Anatomy of a URL: Mapping a URL Namespace to a Struts 2 Action Namespace

actions. The mapping process is quite simple. The URL combines the servlet context with the namespace of the package with the action name itself. Note that the action name takes the action extension. Chapter 3 will more fully cover the mechanics of namespaces.

Back to the actions. The first action, the Name action, does not do any real back end processing. It merely forwards to the page that will present the user with a form to collect her name.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Best Practice: Use empty action components to forward to your results, even if they are simple JSP's requiring no dynamic processing. This keeps the application's architecture consistent, pre-wires your workflow in anticipation of increases in complexity, and hides the real structure of your resources behind the logical namespace of the Struts 2 actions.

While we could technically use a URL that hits the form JSP directly, it's a well accepted best practice to route these requests through actions regardless of their lack of actual processing. As you can see, such pass through actions do not specify an implementation class. They will automatically forward to the result they declare. So, this action points directly at the NameCollector.jsp page, which renders the form that collects the name. Listing 2.6 shows the contents of /chapterTwo/NameCollector.jsp.

Listing 2.6 NameCollector.jsp Uses Struts 2 UI Component Tags to Render the Form

```
<%@ page contentType="text/html; charset=UTF-8" %>    ANNOTATION #1
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>

    <head>
        <title>Name Collector</title>
    </head>

    <body>

        <h4>Enter your name </h4>
        <s:form action="HelloWorld">                                ANNOTATION #2
            <s:textfield name="name" label="Your name"/>
            <s:submit/>
        </s:form>

    </body>

</html>
(annotation) <#1"Standard JSP Directives">
(annotation) <#2"Struts 2 UI Component Tags">
```

At this point, we provide this listing only for the sake of full disclosure. We will cover the Struts 2 UI Component tags fully in Chapter Six. For now, just note that a simple tag or two renders a complete HTML form. And, as you will shortly see, these tags also bind the form to the various features of the framework such as automatic data transfer.

The second action, the HelloWorld action, receives and processes the submission of the name collection form, customizing a greeting with the user's name. While this business logic is still quite simple, it certainly needs a real action. In its XML declaration, the HelloWorld action specifies `manning.chapterTwo.HelloWorld` as its implementation class. Listing 2.7 shows the simple code of this action implementation. As promised, there's not much to it.

Listing 2.7 The HelloWorld Action Provides an Execute() Method to Conduct its Work and JavaBean's Properties to Hold its Data

```
package manning.chapterTwo;
```

```
public class HelloWorld {
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

private static final String GREETING = "Hello ";

public String execute() {           ANNOTATION #1

    setCustomGreeting( GREETING + getName() );
    return "SUCCESS";               ANNOTATION #2
}

private String name;                ANNOTATION #3
private String customGreeting;

public String getName() {
    return name;
}

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

public String getCustomGreeting()
{
    return customGreeting;
}

public void setCustomGreeting( String customGreeting ){
    this.customGreeting = customGreeting;
}
}
(annotation) <#1"The Action's Business Logic">
(annotation) <#2"Control String will Select Result">
(annotation) <#3"JavaBean's Properties Hold the Data">

```

The execute() method contains the business logic, a simple concatenation of the submitted name with the greeting message. After this, the execute() method simply returns a control string that indicates which of its results should render the result page.

While many Struts 2 actions will implement the Action interface, which we will cover in Chapter 3 Working With Struts 2 Actions, they are only obligated to meet an informal contract. The HelloWorld action satisfies that contract by providing an execute() method that returns a string.

The only other important thing to note is the presence of JavaBean's properties to hold the application domain data. For now, recall that we said that the action, as the MVC model component of the framework, both contains the business logic and serves as a locus of data transfer. Though there are other ways the action can hold the data, one common location is on JavaBean's properties. The framework will set incoming data onto these properties when preparing the action for execution. Then, during the execution of the action, the business logic in the action's execute() method can access and work with the data. Looking at the HelloWorld action, we see that the business logic both reads the name value from these properties and writes the custom greeting to these properties. In fact, it will be from this property that the result JSP page will read the custom greeting.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now, let's take a quick look at the JSP that renders the success result for the HelloWorld action. Listing 2.8 shows the HelloWorld.jsp which does this rendering. As you can see, this

Listing 2.8 The HelloWorld.jsp Renders the Result for the HelloWorld Action

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

  <body>

    <h3>Custom Greeting Page</h3>
    <h4><s:property value="customGreeting"/></h4>          ANNOTATION #1

  </body>

</html>
(annotation) <#1"Struts 2 property Tag: Pulls Data from ValueStack">
```

page is quite simple. The only thing to note is that Struts 2 property tag that displays the custom greeting message. Now, you've seen all the code from front to back on a simple Struts 2 application.

We can guess that you still might have questions on how the data gets from the front to the back of this process. So, let's trace the path of data as it comes into, flows through, and ultimately exits the HelloWorld application. First, let's clear up some potential confusion regarding the location of data in the framework. In Chapter One, we learned that the framework provides something called the ValueStack for storing all of the domain data during the processing of a request. We also said that the framework uses a powerful expression language, OGNL, to reference and manipulate that data from various regions of the framework. But, as we have just learned, the action itself holds the domain data. In the case of the HelloWorld action, that data is held on JavaBean's properties exposed on the action itself. So, what gives?

In short, both are true. The data is both stored on the action and in the ValueStack. Here's how. First, domain data is always stored on the action. We will see variants on this, but this is essentially true. This is great because it allows convenient access to data from the action's execute() method. So that the rest of the framework can access the data, the action object itself is placed on the ValueStack. The mechanics of the ValueStack are such that all of the properties of the action will then be exposed as top level properties of the ValueStack itself, and, thus, accessible via OGNL. Figure 2.7 demonstrates how this works with the HelloWorld action as an example.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

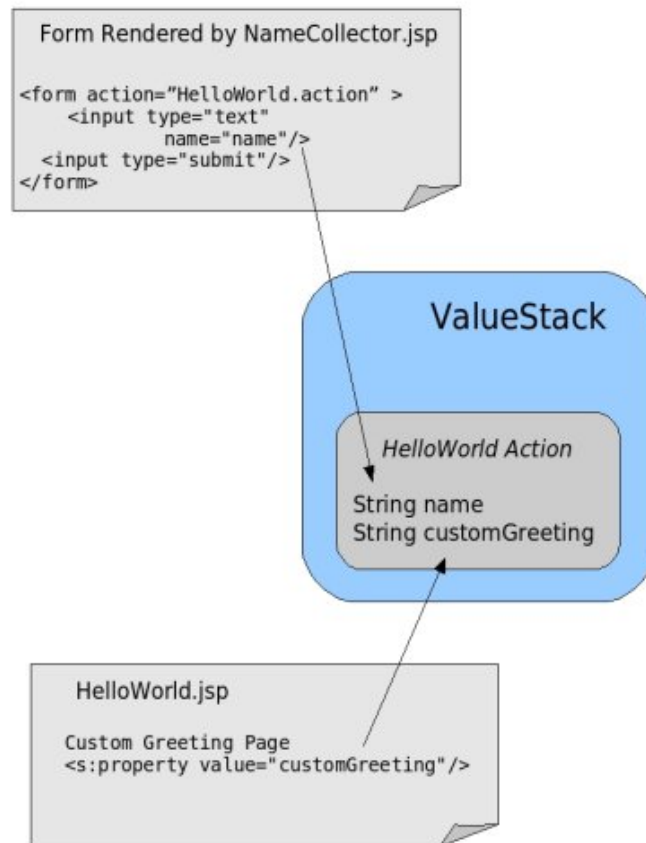


Figure 2.7 Every Action is Placed on the ValueStack So that its Properties are Exposed to Framework Wide OGNL Access

As Figure 2.7 shows, the action holds the data giving its own Java code convenient access. At the same time, the framework makes the properties of the action available on the ValueStack so that other regions of the framework can access the data as well. In terms of our HelloWorld application, the two most important places where this occurs are on the incoming form and the outgoing result page. In the case of the incoming request, the form field name attribute is interpreted as an OGNL expression. The expression is used to target a property on the ValueStack; in this case, the name property from our action. The value from the form field is automatically moved onto that property by the framework. On the other end, the result JSP pulls data off the customGreeting property by likewise using an OGNL expression, inside a tag, to reference a property on the ValueStack. Obviously, this complicated process needs more than a quick sketch. We will cover it fully, particularly in Chapters Five and Six.

That pretty much gives us as much as we need to know at this point. We have seen how to declare actions and results. We have also learned a bit about how the data moves through the framework. You might have noticed that we didn't declare any interceptors. Despite the importance of interceptors, the HelloWorld application declares precisely zero of them. It avoids declaring interceptors itself by making use of the default interceptor stack provided by the framework. This is very common practice.

One of the exciting new features of Struts 2 is the use of annotations instead of XML. What's the big deal? Let's see.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

2.3 HelloWorld Using Annotations

Now we will take a quick peak at the annotation based version. We won't go back through how the application works. It's all exactly the same. The only difference is in the declarative architecture mechanism used to describe our applications Struts 2 components. We have reimplemented the HelloWorld application and you can check it out by hitting the AnnotatedHelloWorld on the main menu page. However, it will work exactly the same of course. However, if you examine the source code, you will see the difference.

As we have said, the annotations are placed directly in the source code of the actions themselves. If we tell the framework where we keep our action classes, it will automatically scan them for annotations. The location of our actions is specified in an initialization parameter to the Struts 2 FilterDispatcher, defined in the web.xml deployment descriptor for the application. The following code snippet shows the relevant portions of that file.

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

    <init-param>
      <param-name>actionPackages</param-name>
      <param-value>manning</param-value>
    </init-param>
  </filter>
(annotation) <#1"Please Scan the manning Package for Annotations!">
```

ANNOTATION #1

Note, the value of the parameter is a Java package name. But just telling the system where our action classes are is not enough. In addition to this, we must somehow identify which classes are actions. To mark our classes as actions, we either make our actions implement the `com.opensymphony.xwork2.Action` interface, or use a naming convention where our class names end with the word Action. We will explain all about the Action interface in Chapter 3. For now, all you need to know is that it's an interface that identifies the class as an action. In the annotations version of HelloWorld, one of our actions, the `AnnotatedNameCollector`, implements the Action interface to let the system know its true identity. The other action, the `AnnotatedHelloWorldAction`, uses the naming convention.

Now that the framework knows how to find our annotations, let's see how they actually work. As we discuss annotations, we will refer back to their counterpart elements in the XML of the first version of HelloWorld. The most notable thing is that several of the elements disappear entirely. We do not have to provide any meta-data for the package. If we accept the intelligent defaults of the annotation mechanism, the framework can create a package to hold our actions without our help. The framework makes some intelligent assumptions and generates the package for us. Most importantly, the framework assumes that our namespace for this package will be derived from the Java package namespace of the action class.

The entire Java package namespace is not used however. The framework only uses the portion of the package namespace beneath the package specified in the `actionPackages`

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

parameter. In our case, we told the framework to look in the `manning` package, and our action classes reside in the `manning.chapterOne` Java package. The framework will give this package a namespace of “chapterOne”. In this fashion, all annotated actions in the same Java package will be added to the same Struts 2 package / namespace.

We also don't have to define the action element when using annotations. In fact, it's not possible to define the action ourselves. Since the annotations reside in the action class itself, the framework can easily deduce the details of the action component. The class providing the object is obviously the class that contains the annotation. The name of the action is derived from the name of the Java class by a simple process. First, the Action ending on the name, if present, is dropped. Second, the first letter is dropped to lower case. For instance, this version of the HelloWorld uses the `AnnotatedHelloWorldAction` class. After removing the ending and changing the case of the first letter, we end up with:

`http://localhost:8080/manningHelloWorld/chapterTwo/annotatedHelloWorld.action`

We'll look at the annotations in the `AnnotatedHelloWorldAction` class first since its really the core part of our application. If you look at the source you see that everything is the same as before. The only difference is the name and the presence of a class level annotation. The most important part of the name is the use of the naming convention, whereby our class name ends with the word Action, to identify our class as an action. The annotation comes just before the class declaration:

```
@Result(name="SUCCESS", value="/chapterTwo/HelloWorld.jsp" )
```

Even if your not familiar with Java annotations, it should be easy enough to see what's going on here. The information is exactly the same as in the XML elements. While the package and action elements are gone, they have been derived from the Java class itself. Finally, the result annotation resides nested with in the containing Java class and Java package just as the result element was nested in the equivalent XML elements. The same wiring occurs. The same intelligent defaults are inherited.

We could leave off here, but we probably better say something about this other action. Recall that we wrapped the `NameCollector.jsp` in an empty action element in the first example? We said that this use of a pass through action, rather than hitting JSP's directly, was considered a best practice. So, we better try to do the same with our annotations version of the HelloWorld application. This is technique is slightly complicated by the fact that annotations occur within the action classes. This forces us to create an action class even though we don't really need one. But this is not a problem.

If we look at `AnnotatedNameCollector.java`, we see that it is an empty class. This is shown in the code snippet below. It provides nothing but a container for the result annotation that points to our JSP page. This result annotation provides the exact same information as the corresponding result element from the XML version. Like the result XML element, it accepts the intelligent defaults for all attributes except the page it will render.

```
@Result( value="/chapterTwo/AnnotatedNameCollector.jsp" )

public class AnnotatedNameCollector extends ActionSupport {

    /* EMPTY */

}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

There's also a bit more going on here than we currently need to know, but we'll give the short story for now. This class extends `ActionSupport`. `ActionSupport` is a framework provided implementation of the `Action` interface we mentioned above. In terms of action logic, it does nothing but render the result we have defined. It does provide some support to help some common tasks, which we will learn about in Chapter 3. For now, just note that it is a convenient help class. Also note that since it implements the `Action` interface for us, we can drop the class naming convention and the framework will still pick this up while scanning for actions.

Now that we've covered the details of the two interfaces to Struts 2's declarative architecture, we better make sure you're not confused. Really, the declarative architecture is the thing we are trying to learn, and it's quite cleanly designed. If the two styles of meta-data make you dizzy, just breathe slowly and contemplate the clarity of the Struts 2 architecture that they both describe. Ultimately, the HelloWorld application, no matter how you describe it, consists of a pair of Struts 2 actions. The first one receives the request for the name collection form. The second one receives the name from the form, processes it, and returns the customized greeting.

2.4 Summary

There you go! We've broken the ice, taken the lay of the land, and got something up and running. Not only have we finished our HelloWorld hand shaking, we have completed the PART I: Struts 2: A Brand New Framework. At this point, you should have a good grasp of the fundamentals of building a Struts 2 web application. Let's review what we've learned in this chapter before moving on.

In this chapter we introduced the declarative architecture of Struts 2. While some may refer to this stuff as configuration, we like to distinguish between configuration of the framework itself and configuration of your application's architecture. The first, probably more correctly labeled configuration, involves tweaking or tuning the behavior and performance of the framework. The second type plays a much more central role in the development of our web applications; it involves nuts and bolts of defining your applications structure. This is the declarative architecture of Struts 2.

The framework provides two interfaces for declaring the architectural components of which your application consists: XML and Java annotations. As we have seen, both of these are fairly simple. While annotations are considered to be more elegant than XML, we have opted to use XML because of its educational convenience. But we expect that many of you will ultimately choose annotations over XML, and we provide Appendix A as a full reference to the annotations based version of the Struts 2 declarative architecture.

The rest of the book? You've already met the main components of the framework, but you need to learn a lot more about them. In *Part II: Getting to the Heart of the Matter: Actions, Interceptors and Type Conversion*, we will go into all of the configurable details as we give focused treatment to each of these core components of the framework. In *Part III: Building the View: Tags, UIComponents, Results* we will examine the results and the framework support for specific view technologies such as custom tags. In *Part IV: Finishing the App: Resource Management, Validation, Data Persistence, and Internationalization* we look techniques for adding the finishing touches to the application and managing resources from the business and data tiers. And, finally, *Part V: Advanced Topics* will address such topics as framework extension with plugins and best practices.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

3 *Working with Struts 2 Actions*

In this chapter we will conduct a detailed study of the Struts 2's *action* component. As we have already learned, the action component serves as a tidy container for all of the calls to business logic and the data tier that should be invoked by a particular request. In MVC terms, we can say that the action component encapsulates the interaction with the model. Or we could try to rephrase this in more down to earth language by saying that the action represents the work that should occur when a given request arrives at the framework. Sometimes this work is as trivial as the pass through action we saw in the HelloWorld application, which automatically forwarded to the JSP page without any consideration of any kind. But more often this unit of work includes complex interaction with business logic and the data tier, the logical outcome of which is used to determine which of several possible pages should be returned. Regardless of complexity, the details of the work associated with a given request will always be contained within an action component. In the end, you will spend much of your time as a Struts 2 developer working with actions.

This chapter will give you everything you need to start building your application's actions. First, we cover the details of how an action fits into the framework. Using the XML based mechanism for declarative architecture, we will explore all of the options available to us when we declare our actions. We will also take an in depth look at the *package* structure in which your actions must be organized. Then we will learn about some of the convenience classes provided with the framework to ease implementation of actions. After that we will introduce the mechanisms that support the framework's easy transfer of data from the request to the action to the rendering of the result page. Along the way we will also start to learn how actions and interceptors work together on many tasks, such as data validation and internationalization. Finally, we'll take a look at a useful case study of a file upload action. By the end of this chapter,

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

you will be able to get the core of your application up and running. You must be excited, so off we go.

3.1 *Introducing Struts 2 Actions*

By the end of this chapter we will know how to implement an action, wire it into the framework so that it executes when we want it to, and even tweak its execution to our own liking. To set the stage, we will start with a descriptive sketch of the role that this important Struts 2 component plays in the framework. We will explain the purpose and various roles of the action component. We will contrast the Struts 2 action with the similarly named component of the Struts 1 framework. And we will study the obligations that an object serving in the role of an action has towards the framework in general.

3.1.1 What does an action do?

Three things actually. As you've probably understood by now, an action's most important role, from the perspective of the framework's architecture, is the encapsulation of the actual work to be done for a given request. But the action also serves in two other important capacities. First, the action plays an important role in the transfer of data from the request through to the view, whether its a JSP or other type of result. Second, the action must assist the framework in determining which result should render the view that will be returned in the response to the request. Let's see how the action component fulfills each of these various roles.

Encapsulates interaction with the MVC model

As we learned in chapter one, Struts 2 implements an MVC base architecture. The MVC pattern separates the concerns of an application into three main components. In this separation, the model contains the business logic and data of the application. In it's implementation of the MVC pattern, Struts 2 use the action component to isolate these model concerns from the rest of the application. The code inside your actions should concern itself with, and only with, the logic of the work associated with the request. The following code snippet, from the previous chapter's HelloWorld application, shows the work done by the HelloWorldAction.

```
public String execute() {  
  
    setCustomGreeting( GREETING + getName() );  
  
    return "SUCCESS";  
}
```

The action's work is to build a customized greeting for the user. As we can see, this action's execute() method does little else than build this greeting. In this case, the business logic amounts to little more than a concatenation. If it were much more complex, we would probably have

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

bumped that logic out to a business component and injected that component into the action. The data tier, in this simple case, is merely the static field GREETING. Again, if the data tier were more complex, we would bump that logic out to a data component and inject that component as well. We will learn some techniques for injecting these components later in the book. For now, we just want to come to grips with the notion that an action encapsulates the calls to business logic and the data tier, and nothing but those calls.

Provides locus for data transfer

Now that we've just made a big deal about how the action should do nothing other than encapsulate the crystalline interactions with the model, we have to tell you about a couple of other things the action does. Actually, this is not at all contradictory. These other tasks are, in truth, side effects of the action's dedication to its first role. In order to do its work, the action must have access to the input data from the request. In previous web application frameworks, this data was passed into the entry method call, i.e. `execute()`. In order to clean up the method signature, the benefits of which we will discuss in a few paragraphs, the Struts 2 framework allows the action component to also serve as a data transfer object itself.

Listing 3.1, also from the HelloWorldAction, shows the code that allows that action to serve as a data transfer object.

Listing 3.1 Struts 2 will Automatically Transfer Request Data onto the JavaBeans Properties that your Action Class Exposes

```
private String name;

public String getName() {
    return name;
}

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

private String customGreeting;

public String getCustomGreeting()
{
    return customGreeting;
}

public void setCustomGreeting( String customGreeting ){
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>


```

        this.customGreeting = customGreeting;
    }

```

The action merely implements JavaBeans properties for each piece of data that it wishes to carry. Request parameters with matching names will automatically be transferred onto this object by the framework. In this case, the name parameter, from the name collection form, will be set on the name property above. In addition to receiving the incoming data from the request, these JavaBeans properties on the action will also expose the data, after processing, to the result. The HelloWorld action's logic sets the custom greeting onto customGreeting property, which makes it available to the result as well. The result is a JSP page in the case of the HelloWorldAction.

In addition to these simple JavaBeans properties, there are a couple other techniques for using the action as a data transfer object. We will examine these alternatives later in this chapter, and we will also examine the mechanisms by which the actual data transfer occurs. For now, we just want to recognize that the action serves as a centralized data transfer object that can be used to make the application data available in all tiers of the framework.

The use of actions as data transfer objects should probably ring some alarms in the minds of alert Struts 1 developers. In Struts 1, the action classes were singletons. If this were still true, we couldn't use the action object itself as a data carrier for the request. In a multi-threaded environment, such as a web application, it would be very problematic to store data in instance fields in the manner of the JavaBeans properties we have seen above. Struts 2 solves this problem by creating a new instance of an action for each request that maps to it. This fundamental difference allows Struts 2 objects to exist as dedicated data transfer objects for each request. If you have concerns about the performance of a system that creates a new action object for each request, just consider the fact that even with a singleton action object you still would have to create some sort of data transfer object for the request. So, object instantiation is certainly not higher in Struts 2.

Returns control string for result routing

The final duty of an action component is to return a control string to the framework. As with the data transfer duties, this duty actually helps to keep the action more focused on its sole purpose of interacting with the model. As with the data transfer, previous frameworks passed routing objects into the entry method of the action. Again in the interest of a clean method signature, the manner in which the framework controls the choice of the result that should handle the view rendering was changed. Struts 2 controls the routing to the appropriate result by a control string returned by the action's `execute()` method. The value of this string will be inspected by the framework in order to determine which result will render the view for this request. The value of the string must match the name of the desired result as configured in the declarative architecture. The HelloWorldAction returns the strings "SUCCESS". As you can see from the our XML declaration of this action, "SUCCESS" is the name of the result component which will forward to our JSP:

```

<action name="HelloWorld" class="manning.chapterOne.HelloWorld">
    <result name="SUCCESS">/chapterOne/HelloWorld.jsp</result>
</action>

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

As we indicated, the HelloWorld application has a rather simple logic for determining which result it will choose. In fact, it will always choose the “SUCCESS” result. Most real world action's will have more a more complex determination process, and the result choices will almost always include some sort of error result to handle problems that might occur during the action's interaction with the model. Regardless of the complexity, actions must ultimately return a string that maps to one of the result components available for rendering the view for that action.

Now that we've got a good idea of what an action does, it's time to see how it all falls in place in a real application. In the next section, we'll see how to organize our actions into packages and we'll take our first glimpse at the Struts2 Portfolio application, the main sample application for this book.

3.2 Packaging your actions

Whether you declare your action components with XML or Java annotations, when the framework creates your application's architecture it will organize your components into logical containers called *packages*. Struts 2 packages are very similar to Java packages. They provide a mechanism for grouping your actions based upon commonality of function or domain. Many important operational attributes, such as the URL namespace to which actions will be mapped, are defined at the package level. And, importantly, packages provide a mechanism for inheritance, which, among other things, allows you to inherit the components already defined by the framework. Let's check out the details of Struts 2 packages by examining the Struts 2 Portfolio application's packaging.

3.2.1 The Struts 2 Portfolio application

Throughout this book we will develop and examine a sample application called the Struts 2 Portfolio. This application provides a simple web-based portfolio software to artists. Artists can use the system to have an online portfolio of their work. When an artist decides that they want to use The Struts 2 Portfolio, they will have to register with the system. It's free of charge, so we'll just collect some harmless personal info and create an account for them. Once the artist has an account, they can login to the secure portion of the application to conduct such sensitive business as the addition and deletion of images to and from their portfolio. The other side of the portfolio is the public side that allows any interested web surfer to view the images in the portfolio. This public face of the portfolio will not be protected by security.

While this application remains functionally simple, it has enough complexity to allow us to demonstrate the core Struts 2 concepts. A quick analysis of our requirements tells us that we have three distinct regions in our web application. The first region contains the functionality related to managing accounts. This includes the creation and deletion of artist's accounts, as well as the login functionality. The second region of our application contains the administrative functionality that the artists will use to maintain their portfolios. Finally, the third region contains all of the functionality for showing the public view of the portfolio. Let's see how we can use Struts 2 packages to organize our application according to these three regions that we have identified.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

3.2.2 Organizing your packages

It's entirely up to you to decide on an organizational theme for your application's package space. We will organize The Struts 2 Portfolio's packages based upon commonality of functionality, a common choice. The important thing is just to see how the packages can be declared and configured to achieve a given organizational structure. As we have noted earlier, you can declare your application's architectural components with XML files or with Java annotations located in your action class files. We also noted that we will use XML files for our sample code. With that in mind, let's take a look at chapterThree.xml file that declares the components for our first cut of the Struts 2 Portfolio application. This XML file contains declaration of two packages, one for the public actions of the application and one for the secure actions of the application. Listing 3.2 shows the declaration of the secure package.

Listing 3.2 Declaration of a Package

```
<package name="chapterThreeSecure" namespace="/chapterThree/secure"
    extends="struts-default">

    <action name="AdminPortfolio" >
        <result>/chapterThree/AdminPortfolio.jsp</result>
    </action>

    <action name="AddImage" >
        <result>/chapterThree/ImageAdded.jsp</result>
    </action>

    <action name="RemoveImage" >
        <result>/chapterThree/ImageRemoved.jsp</result>
    </action>

</package>
```

The package declared in Listing 3.2 contains all of the secure actions for the Struts 2 Portfolio application. These actions require user authentication. A quick glance at the names of these actions should be sufficient to give a good idea of their functional purpose. Obviously, the actions that add and delete images from a portfolio should be secured behind authentication. We do, after all, want to make sure that the user who is removing images from the portfolio actually owns that portfolio. Grouping these together allows us to share declarations of components that might be useful to our authentication mechanism. Additionally, we have chose to give a special URL namespace to these secure actions. We want our users to notice from the URL that they have entered a secure region of the web site.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now, let's take a look at the declaration of the package itself. Actually, there's only four attributes that you can set on a package: name, namespace, extends, and abstract. Table 3.1 summarizes these attributes.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 3.1 The Attributes of the Struts 2 Package

Attribute	Description
name (required)	name of the package
namespace	namespace for all actions in package
extends	parent package to inherit from
abstract	if true, this package will only be used to define inheritable components, not actions

While it may be a bit challenging to decide upon the organizational strategy by which you divide your actions into packages, the actual declaration of them is quite simple. The only required attribute is the name. The name attribute is merely a logical name by which you can reference the package. In Listing 3.2, we have named our package “chapterThreeSecure”. This name, as all good names, indicated the purpose of this package. This package contains the secure actions of the Chapter 3 version of the Struts 2 Portfolio sample application.

Next, we have set the namespace attribute to “/chapterThree/secure”. As we have seen in the case of the HelloWorld application, the namespace attribute is used to generate the URL namespace to which the actions of these packages are mapped. In the case of the AddImage action from Listing 3.2, the URL will be built up as follows:

`http://localhost:8080/manningHelloWorld/chapterThree/secure/AddImage.action`

When a request to this URL arrives, the framework consults the “/chapterThree/secure” namespace for an action named “AddImage”. Note, you can give the same namespace to more than one package. If you do this, the actions from the two packages map to the same namespace. This is not necessarily a problem. You might choose to separate your actions into separate packages for some functional reason that doesn't necessarily warrant a distinct namespace. In our case, we have decided that we want the user to see a URL namespace change when they enter the secure region of the application.

If you don't set the namespace attribute, your actions will go into the *default* namespace. The default namespace sits beneath all of the other namespaces waiting to resolve requests that don't match any explicit namespace. Consider the following:

`http://localhost:8080/manningHelloWorld/chapterSeventy/secure/AddImage.action`

If this request arrives at our sample web application, the framework will attempt to locate the “/chapterSeventy/secure” namespace. As this namespace does not exist, the AddImage action will certainly not be found in it. As a last resort, the framework will search the default namespace for the AddImage action. If it is found there, the URL resolves and the request is serviced. Note, the default namespace is actually the empty string “”. You can also define a root namespace such as “/”. The root namespace is treated as all other explicit namespaces and must be matched. It's important to distinguish between the empty default namespace, which can catch all request

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

patterns as long as the action name matches, and the root namespace which is an actual namespace that must be matched.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

The next attribute that we set in Listing 3.2 is the `extends` attribute. This important attribute names another package whose components should be inherited by the current package. This is similar to the `extends` comment in Java. If you think of the named package being the superclass of your current package, you will understand that the current package will inherit all the members of the superclass package. Furthermore, the current package can override members of the super package. Our `chapterThreeSecure` package extends something called the “`struts-default`” package. This is a very common practice. The all important `struts-default` package is predefined by the framework and contains many useful components. By extending this package, you can use the built in components in your own applications. Familiarity with the components made available by the `struts-default` package will allow you to save hours and hours of development time. The intelligent defaults we have spoke of come largely from this package. We will make due note of these features as we use them throughout the book.

For now, we'll just take a quick look at one of the intelligent default components that we've already made use of in the `HelloWorld` application. It's probably a good idea to get the concrete details of these intelligent defaults on the table as early as possible. Otherwise, it starts to sound like so much fluff. We have already talked about how the framework uses interceptors to do common tasks such as the transfer of request parameter data onto the JavaBeans properties of the action object. We have also mentioned that each action has a stack of interceptors that fire before, and after, the action itself executes. Even though we know we have used interceptors in the `HelloWorld` application, we have not yet seen any elements in our XML that declare these interceptors for our application. Where do they come from?

Definition: The `struts-default` package, defined in the system's `struts-default.xml`, contains a huge set of commonly needed Struts 2 components ranging from complete interceptor stacks to all the common result types.

Here's the secret. Most of the interceptors that you will ever need are found in the `struts-default` package that is declared in the `struts-default.xml` file. You can think of this important file as the framework's own declarative architecture artifact. The `struts-default` package that it defines contains common architectural components that all developers can reuse simply by having their own packages extend it. If you want to see the whole `struts-default.xml` file, it can be found at the root level of the distribution's main jar file, `struts2-core.jar`. Listing 3.3 shows the elements from this file that declare the default interceptor stack that most applications will make use.

Listing 3.3 The Built In `struts-default` Package Declares Many Commonly Used Components

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

...

    <interceptor-stack name="defaultStack">
        <interceptor-ref name="exception"/>
        <interceptor-ref name="alias"/>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>


```

<interceptor-ref name="servlet-config"/>
<interceptor-ref name="prepare"/>
<interceptor-ref name="i18n"/>
<interceptor-ref name="chain"/>
<interceptor-ref name="debugging"/>
<interceptor-ref name="profiling"/>
<interceptor-ref name="scoped-model-driven"/>
<interceptor-ref name="model-driven"/>
<interceptor-ref name="fileUpload"/>
<interceptor-ref name="checkbox"/>
<interceptor-ref name="static-params"/>
<interceptor-ref name="params">
    <param name="excludeParams">dojo\..*</param>
</interceptor-ref>
<interceptor-ref name="conversionError"/>
<interceptor-ref name="validation">
    <param name="excludeMethods">input,back,cancel,browse</param>
</interceptor-ref>
<interceptor-ref name="workflow">
    <param name="excludeMethods">input,back,cancel,browse</param>
</interceptor-ref>
</interceptor-stack>

...

<default-interceptor-ref name="defaultStack"/>

...

</package>

```

Inside the package element, an interceptor stack is declared. It is named, appropriately, “defaultStack”. Near the end of the listing you can see that this stack is declared as the default interceptor stack for this package, as well as any other packages that extend the struts-default package. For now, just note that one of the interceptors in this stack is the “params” interceptor. This is the one that moves the request parameters on to our JavaBeans properties. We will cover

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

the details of the other common interceptors as we move along through the book. We will even learn how to write our own interceptors. For now, we are just interested in the inheritance of the useful components declared in the struts-default package. If your package extends this package, you inherit these components. Very nice.

Before we move on, we should make brief mention of the fourth attribute that you can define for a package. The abstract attribute allows you to define a package that doesn't actually contain any actions. Why would you want to do this? Again, consider the Java class as an example. If you declare a Java class as abstract, it means the class will be used only as a skeleton of inheritable elements. It's not complete enough to stand on its own, but it serves a good place to define common elements that your other packages can inherit. (how does the struts-default package avoid using this abstract attribute since it doesn't have any actions in it ???)

Now we have a pretty good idea of the mechanics of organizing our application into packages. We've even seen one of the packages from the Struts 2 Portfolio sample application. In the next sections, we start to implement our actions. In particular, we will introduce a couple of convenience items that can speed implementations of actions.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

3.3 Implementing actions

Implementing Struts 2 actions is pretty easy actually. Earlier in this chapter we have seen that the contract between the framework and the classes that back the actions provides a great deal of flexibility. Basically, any class can be an action if it wants. It simply must provide an entry method for the framework to invoke when the action is executed. Let's take a look at how the framework makes it even easier to implement your actions. As a case study, we'll continue our look at the Struts 2 Portfolio by digging into a couple of its own action classes.

3.3.1 The optional Action interface

As we have seen, the framework does not place any type requirements on the class that backs the action component. Indeed, a hallmark of the Struts 2 framework is the flexibility it demonstrates towards all of its components. In fact, the only hard requirements for an action class are that it provide an execute method that returns a control String and throws an Exception. And, actually, we've already seen that there's even a bit of wiggle room on these *hard* requirements.

But before you start to get confused, put these flexible details out of your mind. Another equally important hallmark of the Struts 2 framework is its easy path to the most common cases of development. Most of the flexibility in the platform was put there in order to give developers a way to handle special cases without resorting to laborious workarounds. In general, you probably will use the standard execute signature and be completely happy about it. It will certainly suffice for your first steps into Struts 2 development.

The `com.opensymphony.xwork2.Action` interface defines one method:

`String execute() throws Exception`

Since the framework doesn't make any type requirements, you could just put the method in your class without having your class implement this interface. This is fine. But the Action interface also provides some useful String constants that can be used as return values. The constants defined by the Action interface are:

```
public static final String ERROR    "error"
public static final String INPUT    "input"
public static final String LOGIN    "login"
public static final String NONE     "none"
public static final String SUCCESS  "success"
```

These constants can be used as convenient control Strings for your execute method to return. The true benefit is that these constants are also used internally by the framework. This means that you can take advantage of even more intelligent defaults if you use these constants for your routing to your results.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

As an example let's consider the pass through action. Remember the pass through action we used in the HelloWorld application. We said that it was a best practice to route even simple requests through actions. In the HelloWorld application, we used one of these empty actions to hit our JSP page that presents the form that collects the user's name. Here's the declaration of that action:

```
<action name="Name">
    <result>/chapterOne/NameCollector.jsp</result>
</action>
```

The “Name” action doesn't specify a class to provide the action implementation. This is because there's nothing for the action to do. We just want to go to the JSP page that presents the form. Conveniently, the Struts 2 intelligent defaults provide a default action implementation that has an empty execute method which does nothing but automatically return the Action interfaces SUCCESS constant as its control string. The framework must use this String to choose a result. Luckily, or maybe not so luckily (a conspiracy perhaps?), the default name attribute for the result element is also the SUCCESS constant. Since our sole result forgoes defining its own name, it inherits this default and is automatically selected by our action. This is the general pattern by which many of the intelligent defaults operate.

3.3.2 *The ActionSupport class*

In case you were wondering about that mysterious default implementation from the pass through action example, we better not wait much longer to introduce that most convenient of convenience implementations, that most helpful of helper classes, assisting your actions with 24 hour support, making life . . . okay, enough. In this section we're going to introduce, and use, the ActionSupport class, a convenience class that provides default implementations of the Action interface and several other useful interfaces, giving us such things as data validation and

ActionSupport provides the implementation of several important services. If you extend this class, you automatically gain the use of these implementations without having to write the code yourself. This alone makes this class worth learning. In fact, its so useful, most developers will always extend this class. However, the implementations provided by this class also provide a great case study in how to make an action cooperate with interceptors to achieve powerfully reusable solutions to common tasks. In this case, validation and text localization services are provided via a combination of interceptors and interfaces. The interceptors control the execution of the services while the action's implement interfaces with methods that are invoked by the interceptors. This important Struts 2 pattern will become clearer as we work through the details of ActionSupport by examining its use in our Struts 2 Portfolio application.

Basic Validation

While Struts 2 provides a rich and highly configurable Validation Framework, which we will fully examine in Chapter Ten, ActionSupport provides a quick form of basic validation that will serve well in many cases. Moreover, it serves as a great case study of how a crosscutting task such as validation can be factored out of the action's execution logic through the use of interceptors and interfaces. The typical pattern is that the interceptor, while controlling the execution of a given task, may coordinate with the action by invoking methods that it exposes. Usually, these methods are part of a specific interface implemented by that action. In our case, ActionSupport implements two interfaces that coordinate with one of the interceptors from the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

default stack, the `DefaultWorkflowInterceptor`, to provide basic validation. If your package extends the `struts-default` package, thereby inheriting the default interceptor stack, and your action extends `ActionSupport`, thereby inheriting implementation of the two necessary interfaces, then you already have everything you need for clean validation of your data.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

For point of reference, Listing 3.4 shows the declaration of the workflow interceptor as its found in the struts-default.xml file.

Listing 3.4 Declaration of DefaultWorkflowInterceptor from struts-default.xml

```
...

<interceptor name="workflow"
class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>

...

<interceptor-stack name="defaultStack">

    ...

    <interceptor-ref name="params"/>

    ...

    <interceptor-ref name="workflow">
        <param name="excludeMethods">input,back,cancel,browse</param>
    </interceptor-ref>

    ...

</interceptor-stack name="defaultStack">

...
```

In Listing 3.4, we first see the declaration element for the workflow interceptor, specifying a name and an implementation class. (Note, this interceptor is called the workflow interceptor because it will divert the workflow of the request back to the input page if a validation error is found.) Next, we see the declaration of the default interceptor stack. We have already seen one of these default interceptors, the parameters interceptor. That interceptor helped move the request data onto our action object. Now, another default interceptor will help us validate that data before accepting it into our model. Note that the workflow interceptor must fire after the parameters interceptor has had a chance to move the data on to the action object. As with most interceptors, sequence is very important.

Now, let's see how this validation works. As with the parameters interceptor, the workflow interceptor seeks to remove the logic of a crosscutting task, validation in this case, from

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

the action's execution logic. In order to coordinate with the parameters interceptor, our action must expose JavaBeans properties, a kind of informal interface. The parameters interceptor invokes the setter methods of this informal interface when it moves the data over from the request. The same strategy is used with validation. When the workflow interceptor fires it will also look for a `validate()` method to invoke on the action. This method is exposed via the `com.opensymphony.xwork2.Validateable` interface. Let's look at how the Struts 2 Portfolio implements this method to provide validation of the data submitted by a user registering with the application. Listing 3.5 shows the validation logic that resides in this method.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Listing 3.5 Basic Validation Logic Implemented in the Validateable Interface's Validate Method

```
public void validate(){

    PortfolioService ps = getPortfolioService();

    if ( getPassword().length() == 0 ){
        addFieldError( "password", "Password is required." );
    }
    if ( getUsername().length() == 0 ){
        addFieldError( "username", "Username is required." );
    }
    if ( getPortfolioName().length() == 0 ){
        addFieldError( "portfolioName", "Portfolio name is required." );
    }

    if ( ps.userExists(getUsername() ) ){
        addFieldError("username", "This user already exists.");
    }

}
```

As you can see, all of the validation logic is contained within the `validate()` method. This leaves this action's `execute()` method a nice and tidy encapsulation of the interaction with the model. Note that technically speaking, `ActionSupport` implements the `validate()` method, but we have to override its empty implementation with our own specific validation logic.

The validation logic that we have provided is quite simple. Certainly not ready for the enterprise big leagues. Basically, we test that the three fields are not empty. One thing we should note is that the validation code retrieves the data from the JavaBeans property getter methods. As we have noted, it's quite important that the parameters interceptor has already set the data on these properties. If we are using the default stack, this should be the case. We also test that the user doesn't already exist in the system. This test requires a dip into our business logic and data tiers. At this point in the book, the Struts 2 Portfolio application uses a very simple encapsulation of business logic and data persistence. The `PortfolioService` object is capable of conducting our simple business needs at this stage. In case you're interested, it contains all the business rules in its simple methods, and it persists its data to an XML file. Even our current management techniques are a bit crud; our action just instantiates a `PortfolioService` object when it needs one. Later in the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

book we learn how to integrate with more sophisticated technologies for managing such important resources. But for now, this keeps our study of the action component more clear.

What happens if validation fails? If any of the fields are empty, or if the username is already in the system, we call a method that adds an error message. After all the validation logic has executed, control returns to the workflow interceptor. Note that there is no return value on the validate method. The secret, as will see, is in the error messages that our validation generates.

Even though control has returned to the workflow interceptor, it's not finished. What does the workflow interceptor do now? Now it's time to earn the workflow name. After the validate() method has checked all the data, the workflow method will check to see if any error messages were generated by the validation logic. If it finds errors, then the workflow interceptor will alter the workflow of the request. It will immediately abort the request processing and return the user back to the input form where the appropriate error messages will be displayed on the form. Try it out! Fire up your application and hit the Chapter 3 version of the Struts 2 Portfolio at:

`http://localhost:8080/manningHelloWorld/chapterThree/PortfolioHomePage.action`

Choose to create an account and fill out the form, but omit some data. For instance, we have omitted the password. When we submit the form, the validation fails and diverts the workflow back to the input form again as seen below in Figure 3.1.

Complete and submit the form to create your own portfolio.

Username:

Password is required.

Password:

Enter a name for your portfolio:

Figure 3.1 Default Workflow Interceptor Returns us to Input Form with Validation Error Messages Displayed on Appropriate Fields

One obvious question remains. Where were those error messages stored and how did the workflow interceptor check to see there were any? The `com.opensymphony.xwork2.ValidationAware` interface defines methods for storing and retrieving error messages. A class that implements this important interface must maintain a collection of error messages for each field that can be validated, such as username, as well as a collection of general error messages that pertain only to the action as a whole. Luckily for us, all of these methods and the collections that back them, are already provided by the `ActionSupport` class. To use them, we simply invoke the following methods:

```
addFieldError ( String fieldName, String errorMessage )
addActionError ( String errorMessage )
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

To add a field error, we must pass the name of the field, as we do in Listing 3.5, and the message that we want displayed to the user. Adding an action scoped error message is even easier as you don't need to specify anything other than the message.

The ValidationAware interface also specifies methods for testing whether any errors exist. The workflow interceptor will use these to determine whether it should redirect the workflow back to the input page. If it finds that errors exist, it will look for a result with the name “input”. The following snippet from chapterThree.xml shows that the Register action has indeed declared such a result:

```
<action name="Register" class="manning.chapterThree.Register">

    <result>/chapterThree/RegistrationSuccess.jsp</result>

    <result name="input">/chapterThree/Registration.jsp</result>

</action>
```

In this case, the workflow interceptor, if it finds errors, will automatically forward to the result that points to the Registration.jsp page because its name is “input”. And, of course, this JSP page is our input form.

Now we've seen how interceptors can help remove validation logic from our action's true execution logic. But we suspect some of you might not be convinced. If you are tempted to complain, “But the validation method is still on the action object!” Of course, you are right. But this does not taint the most important separation of concerns; the validation logic is distinctly separate from the action's own execution logic. This is what keeps our action focused on its pure unit of work – registering a new user. Check out the execute() method's succinct phrasing of the business logic of this task:

```
public String execute(){

    User user = new User();
    user.setPassword( getPassword() );
    user.setPortfolioName( getPortfolioName() );
    user.setUsername( getUsername() );

    getPortfolioService().createAccount( user );
    return SUCCESS;

}
```

We just make the user object and create the account. No problem. If there were some exception that might be generated from our business object's account creation process, we might have a bit of added complexity in our choice of which result we should display. For our simplistic purposes, we're just assuming success.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

But its not just about clean looking code. A more subtle point to take is that the very control flow of the validation process is also separated from the action. This is not just a case of factoring the validation logic out of the execute() method and into a more readable helper method. The validation workflow is itself layered away from the action's workflow because the validation logic is invoked by the validation interceptor. In other words, the validation interceptor is really the one controlling the execution of the validation logic. The interceptors all fire before the action itself gets a chance to execute. This separation of control flow is what allows the workflow interceptor to abort the whole request processing and redirect back to the input page without ever entering the action's execute() method. This is exactly separation that interceptors are meant to provide.

Now, its beginning to feel like we've really learned something. We can write actions that automatically receive and validate data. That's certainly cool, but let's not get distracted from the real topic at hand. The real lesson to take out of this section is about how action's work together with interceptors to get common chores done without polluting the action's core logic. If you can wrap your mind around this action / interceptor teamwork, then you'll find the rest of the book merely an elaboration on that theme. Of course, there's a lot of cool stuff waiting for you in the remaining chapters, but this bit is the essence of the framework's approach to solving problems. Before we move on, though, we need to look at the other problem that ActionSupport solves for you – localized message text.

Using resource bundles for message text

In our Register action's validation logic, we set our error messages using String literals. If you look back at Listing 3.5, you can see that we pass the String literal “Username is required” to the addFieldError() method. Use of String literals like this creates a well known maintenance nightmare. Furthermore, changing languages for different user locales is virtually impossible with out some layer of separation between the source code and the messages themselves. The well-established best practice is to bundle these messages together into external and maintainable resource bundles, commonly implemented with simple properties files. ActionSupport provides built in functionality for easily managing just such a thing.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

ActionSupport implements two interfaces that work together to provide this localized message text functionality. The first interface, `com.opensymphony.xwork2.TextProvider`, provides the access to the messages themselves. This interface exposes a flexible set of methods by which you can retrieve a message text from a resource bundle. ActionSupport implements these methods to retrieve their messages from a properties file resource. No matter which method you use to retrieve your message, you will refer to the message by a key. The TextProvider methods will return the message associated with that key from the properties file associated with your action class.

Getting started with a properties file resource bundle is quite easy. First, we need to create the properties file and give it a name that mirrors the action class for which it provides the messages. Listing 3.6 shows our Register action's associated properties file, `Register.properties`.

Listing 3.6 Register.properties Contains the Message Texts for the Register Action, Stored as Key – Value Pairs

```
user.exists=This user already exists.  
username.required=Username is required.  
password.required=Password is required.  
portfolioName.required=Portoflio Name is required.
```

In case you are unfamiliar with properties files, they are just simple text files. Each line contains a key and its value. In order to have the ActionSupport implementation of the TextProvider interface find this properties file, we just need to add it to the Java package that contains our Register class. In this case, you can find this file in the source at `manning/chapterThree/Register.properties`, right next to `Register.java`.

Once the properties file is in place, we can use one of the TextProvider `getText()` methods to retrieve our messages. Listing 3.7 shows our new version of the Register action's validate logic:

Listing 3.7 The Register Action's Validation Code using ActionSupport's Resource Bundle Support to get the Validation Error Messages

```
public void validate(){  
  
    PortfolioService ps = getPortfolioService();  
  
    if ( getPassword().length() == 0 ){  
        addFieldError( "password", getText("password.required") );  
    }  
    if ( getUsername().length() == 0 ){  
        addFieldError( "username", getText("username.required") );  
    }  
    if ( getPortfolioName().length() == 0 ){
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

        addFieldError( "portfolioName", getText( "portfolioName.required" ));
    }
    if ( ps.userExists(getUsername()) ){
        addFieldError("username", getText( "user.exists"));
    }
}

```

As you can see, instead of String literals we now retrieve our message text from ActionSupport's implementation of TextProvider. This allows us to retrieve our messages from properties files based upon a key. This layer of separation makes our message text much more manageable. Changing messages means only editing the properties file; the source code's semantic keys never need to be changed.

ActionSupport also provides a basic internationalization solution for the localizing message text. The `com.opensymphony.xwork2.LocaleProvider` interface exposes a single method, `getLocale()`. ActionSupport implements this interface to retrieve the user's locale based upon the locale setting sent in by the browser. You could of course implement your own version of this interface to search somewhere else for the locale, such as in the database. But if the browser setting is good enough for your requirements, you don't have to do too much to achieve a basic level of internationalization.

You still retrieve your message texts as we did above. Even when we weren't taking advantage of it, the ActionSupport's TextProvider implementation has been checking the locale every time it retrieves a message text for us. It does this by calling the `getLocale()` method of the `LocaleProvider` interface. With the locale in hand, the TextProvider, a.k.a. ActionSupport, tries to locate a properties file for that locale. Of course, you have to provide the properties file for the locale in question, or it will just serve up the standard English. But it's very simple to provide properties files for all locales that you wish to support. In the Struts 2 Portfolio, we are providing a Spanish properties file. The hard part is finding a translator. Although, for our purposes, a certain well known web based translator will suffice. In order to see this in action, set your browser's language support to Spanish and submit the registration form again, omitting one of the fields to see the error message that it provides. *Hablo Espanol?*

As with validation, the internationalization provided by ActionSupport is relatively primitive. If it suffices for your application, great. But if you need more, we'll see how to get cutting edge validation and internationalization from the Struts 2 framework in later chapters dedicated to each subject in turn. For now, we've got a pretty decent start on building actions. Next, we'll look at implementing our data transfer with complex objects instead of simple JavaBeans properties, something that will save us a lot of work.

3.4 Deeper data transfer with object backed properties and ModelDriven actions

Up until now, our action's have all received the data from the request on simple JavaBeans properties. While powerful and elegant, we can do even better. Rather than receiving each piece of data individually, then creating an object ourselves on which to place these pieces of data, we can expose the complex object itself to the data transfer mechanisms of the platform. Not only does this save time by eliminating the need to create and populate the object that

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

aggregates the individual pieces of data, it can also save work by allowing us to directly expose an already existing domain object to the data transfer. While a couple of caveats must be noted, and kept in mind, the use of these complex objects as direct data transfer objects presents a powerful option to the developer.

Struts 1 to Struts 2 Perspective: In case you're feeling homesick, we should note the departure of the familiar Struts 1 ActionForm. ActionForms played an important role in data validation and type conversion for the Struts 1 framework. But the cost was high. For each domain object, you typically had to create a mirroring form bean. To add insult to injury, you were then tasked with an additional manual data transfer when you finally moved the valid data from the form bean onto your domain object. For many, one of the biggest thrills of Struts 2 will be letting the framework transfer, validate and bind data directly onto application domain objects, whereupon it can stay!

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

If we want to use complex objects, rather than simple JavaBeans properties, to receive our data, we have a couple of options for implementing such deep transfers. Our first option is also JavaBeans based. We can expose a complex object itself as a JavaBeans property and have the data moved onto the object directly. Another alternative is to use something called a ModelDriven action. This option involves a simple interface and another one of the default interceptors. Like the object backed JavaBeans property, the ModelDriven action also allows us to use a complex Java object to receive our data. The differences in these two methods are slight, and there are no functional consequences to choosing one over the other. But we suspect you might prefer one over the other depending upon your own specific project requirements. We'll learn each technique with examples from the Struts 2 Portfolio.

3.4.1 Object backed JavaBeans properties

We've already seen how the parameters interceptor, included in the default interceptor stack, automatically transfers data from the request onto our action objects. To enable this transfer, the developer needs only to provide JavaBeans properties on her actions, using the same names as the form fields being submitted. Quite easy to do. Despite this ease, we frequently find ourselves occupied with another tedious task. This tedious task consists of collecting these individually transferred data items and transferring them to an application domain object which we must instantiate ourselves. Listing 3.8 shows our previous version of the Register action's execute method.

Listing 3.8 Collecting Individual Data and Building the Application Domain Object by Hand

```
public String execute(){

    User user = new User();

    user.setPassword( getPassword() );

    user.setPortfolioName( getPortfolioName() );

    user.setUsername( getUsername() );

    getPortfolioService().createAccount( user );

    return SUCCESS;

}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

While we were dully impressed with this succinct quality of this method only a few pages ago, we can now see that five of the seven lines do nothing more than assemble the individual pieces of data that the framework has transferred onto our simple JavaBeans properties. Of course, we're still psyched that the data has been automatically transferred and bound to our Java data types, but why not ask for more?

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Why not ask the framework to go ahead and transfer the data directly onto our user object? Why not ask the framework to instantiate the user object for us? Since Struts 2 provides powerful data transfer and type conversion facilities, the true power of which we will learn later in the book, we can certainly ask for these things. And we get them. In this case its quite simple. Let's rewrite our Register action so that it replaces the individual JavaBeans properties with a single property backed by the user object itself. Listing 3.9 shows the new version of our new action as implemented in the `manning.chapterThree.objectBacked.ObjectBackedRegister` class.

Listing 3.9 Using an Object Backed JavaBeans Property to Receive Deep Data Transfers

```
public String execute(){
    getPortfolioService().createAccount( user );
    return SUCCESS;
}

private User user;

public User getUser() {
    return user;
}

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

public void validate(){
    . . .
    if ( getUser().getPassword().length() == 0 ){
        addFieldError( "user. password",
            getText("password. requi red") );
    }
    . . .
}
```

Listing 3.9 has now reduced our business logic to a one liner. This is pretty clean business logic. This logic is so much cleaner because we have let the framework handle the task of instantiating our User object and populating its attributes with data from the request. Previously, we had done this ourselves. In order to let the framework handle this tedious work, we simply replaced the individual JavaBeans properties with a single property backed by the User object itself. We don't even have to create the User object that backs the property because the framework's data transfer will handle this for us when it starts trying to move the data over. Note, our validation code now must use a deeper notation to reach the data items because they must go through the user property to get to the individual fields themselves.

Similarly, we also have to make a couple of changes in the way in which we now reference our data from our results, JSP's in this case. First of all, we have to change the field names in the form that submits to our new action. The bottom line is that we now have another layer in our JavaBeans properties notation. The following code snippet shows the minor change to the textfield name in our form, found in our `Registration_OB.jsp` page.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

OB: `<s:textfield name="user.username" label="Username"/>`

individual properties: `<s:textfield name="username" label="Username"/>`

The names of the other fields in this form are similarly transformed. We also make a similar alteration at the other end of the request. When we render our resulting success page, we must use the deeper property notation to access our data. The following code snippet from `RegistrationSuccess_OB.jsp` shows the new notation.

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="user.portfolioName" />
Portfolio</h3>
```

As you can see, directly using an application domain object as a JavaBeans property allows us to let the framework do even more of our work for us. The minor consequences to this come when we have to go a dot deeper when we reference our data from the JSP pages. Now, we'll take a look at another method of exposing rich objects to the framework's data transfer facilities, one that gives us the same cleaner `execute()` method as the object backed JavaBeans property, but doesn't introduce the extra dot in our view tier data access.

3.4.2 ModelDriven actions

ModelDriven actions depart entirely from the use of JavaBeans properties as a method of exposing our domain data. Instead, they expose an application domain object via the `getModel()` method which is declared by the `com.opensymphony.xwork2.ModelDriven` interface. While this method introduces a new interface, as well as another interceptor, it is quite simple in practice. The interceptor is already in the default stack, and the data transfer is still automatic and even easier to work with than the previous techniques. Let's see how it works.

Implementing the interface requires almost nothing. We do have to declare that our action implements the interface, but there's only one method exposed by ModelDriven, the `getModel()` method. By model, we mean the model in the MVC sense. In this case, it's the data that comes in from the request and is altered by the execution of the business logic. That data is then made available to the view, JSP pages in the case of our Struts 2 Portfolio application. Listing 3.10 shows the new action code from the `manning.chapterThree.modelDriven.ModelDrivenRegister` class.

Listing 3.10 Using ModelDriven Actions to Let the Framework Automatically Transfer Request Data onto Application Domain Objects

```
public class ModelDrivenRegister extends ActionSupport implements
ModelDriven {
```

```
    public String execute() {
        getPortfolioService().createAccount( user );
        return SUCCESS;
    }
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

    }

    private User user = new User();

    public Object getModel () {
        return user;
    }

    public void validate(){
        . . . .
        if ( user.getPassword().length() == 0 ){
            addFieldError( "password",
                getText("password.required") );
        }
        . . . .
    }
    . . . .
}

```

First, we see that our new action implements the ModelDriven interface. The only method required by this interface is the getModel() method that returns our model object, the familiar User object. Note, with the ModelDriven method, we actually have to initialize the User object ourselves. We'll see why in Chapter 5 when we explore the details of the data transfer mechanisms, but for now just keep an eye on this slight but important detail.

We should note one pitfall to avoid. By the time the execute() method of your ModelDriven action has been invoked, the framework has obtained a reference to your model object which it will use throughout the request. Since the framework acquires its reference from your getter, it won't be aware if you change the model field internally in your action. This can cause some data inconsistency problems. If, during your execution code, you change the object to which your model field reference points, your action's model will then be out of synch with the one still held by the framework. The following code snippet demonstrates the problem.

```

public String execute(){
    user = new User();
    user.setSomething();
    getPortfolioService().createAccount( user );
    return SUCCESS;
}

private User user = new User();
public Object getModel () {
    return user;
}

```

In this action's execute() method, the developer has, for some reason, set the user reference to a

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

new object. But the framework still has a reference to the original object as initialized in the instance field declaration for user. When the framework invokes the result, your JSP page data access will be resolved against the old object. The *something* that this erroneous code has set, will be unavailable. You can, of course, manipulate that original model object to your heart's content. Just don't make a new one, or point the existing reference to another one!

Like with the previous object backed JavaBeans property method, the use of a domain object to receive all of the data once again allows us the luxury of a ultra clean execute method. And, again, we do incur a slight penalty related to the depth of our references. As you can see in the validation code of Listing 3.10, we now refer to the password by referencing the model object en route to the password field.

However, we do not incur any depth of reference penalty in our view layer. All references in the JSP pages return to the simplicity of the original Register action that used the simple, individual JavaBeans properties for data transfer. The following code snippets, from Registration_MD.jsp

```
<s:textfield name="username" label="Username"/>
```

and RegistrationSuccess.jsp

```
<h5>Congratulations! You have created </h5>  
<h3>The <s: property value="portfolioName" /> Portfolio</h3>
```

show the renewed simplicity of view layer references to data carried in the ModelDriven action fashion. This is considered to be one of the primary reasons for choosing the ModelDriven method over the object backed JavaBeans property method of exposing domain objects to the data transfer.

3.4.3 Last Words on Using Domain Objects for Data Transfer

First, we want to point out a potential danger in using domain objects for data transfer. The problem comes when the data gets automatically transferred onto the object. As we have seen, if the request has parameters that match the attributes on your domain object, the data will be moved onto those attributes. Now, consider the case where your domain object has some sensitive data attributes that you don't really want to expose to this automatic data transfer. Perhaps an id. A malicious user could conceivably add an appropriately named querystring parameter to the request such that the value of that parameter would automatically be written to your exposed object's attribute. Of course, you can remove these attributes from the object, but then you start to lose the value of reusing existing objects rather than writing new ones. Unfortunately, there is no good solution to this issue yet. Frequently, you won't have anything to worry about, but it's something to keep in mind when you are developing your actions.

Ultimately, it will be up to you to choose a method of receiving the data from the framework. Each method has its purpose and we believe that the requirements of your projects will typically determine the approach that is most appropriate. Throughout the rest of this book, we will see many examples of best practices and integration with other technologies that will spell out some of the cases when one or another of the methods serves best. Sometimes, its appropriate to use a little of each. Did we forget to mention that you can do all of them at the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

same time, if you like? Again, the platform is very flexible. Now it's time to look at some advanced topics.

3.5 File Uploading: A Case Study

At this point, you pretty much have the tools you need to write your application's action components and wire them into a rudimentary Struts 2 application. We suspect you've even deduced enough to get started implementing a view layer with JSP results. Later in the book, you'll see how much more the framework has to offer your view layer when we get to the details of results, tags, UI components, and Ajax integration. For now, we want to round out our treatment of the action component by showing a very useful case study.

3.5.1 Why File Uploads?

Uploading files are one of the more common, and commonly messy, domain tasks of a web application is the file upload. Most of you will have to implement them. Our sample application, the Struts 2 Portfolio, will certainly need to upload some image files, otherwise the portfolio might be somewhat drab. One reason we are choosing to show you how to upload files now, rather than in a later chapter in the book, is that we believe the file upload mechanism helps demonstrate the framework's persistent pattern of using interceptors to layer the logic of common tasks out of the action itself. So, let's learn how actions can work with an interceptor from the default stack to implement ultra-clean file uploading.

3.5.2 Getting Built-in Support via the *struts-default* Package

As with most tasks that you find yourself routinely asked to implement, Struts 2 provides default implementations of the necessary functionality in its intelligent defaults. In this case, the default interceptor stack includes the FileUploadInterceptor. As you might recall, *struts-default.xml* is the system file that defines all of the built-in components. Listing 3.11 shows the elements from that important file that declare the fileUpload interceptor and make it a part of the default interceptor stack.

Listing 3.11 Declaration of the FileUploadInterceptor and its Inclusion in the Default Interceptor Stack

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

    <interceptors>

        ...

        <interceptor name="fileUpload"
            class="org.apache.struts2.interceptor.FileUploadInterceptor"/>

        ...
    </interceptors>
</package>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```
</interceptors>

...

<interceptor-stack name="defaultStack">

    ...

    <interceptor-ref name="model-driven"/>
    <interceptor-ref name="fileUpload"/>
    <interceptor-ref name="params"/>

    ...

</interceptor-stack>

</package>
```

As you can see in the listing, the struts-default package contains a declaration of the fileUpload interceptor, backed by the `org.apache.struts2.interceptor.FileUploadInterceptor` implementation class. This interceptor is then added to the defaultStack so that all packages extending the struts-default package will automatically have this interceptor acting upon the execution of their actions. We, of course, make our Struts 2 Portfolio packages extend this package to take advantage of these built-in components.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

3.5.3 What does the fileUpload interceptor do?

Basically, it creates a file upload version of the automatic data transfer mechanisms we have already seen. With the previous data transfers we were dealing with the transfer of the form field data from the request to the matching JavaBeans properties on our action objects. We explained that the params interceptor, also part of the default stack, was responsible for moving all of the request parameters onto the action object everywhere that the action provided a JavaBeans property that matched the name of the request parameter. In Listing 3.11 you can see that the default stack places the fileUpload interceptor just before the params interceptor. When the fileUpload interceptor executes, it processes the multi-part request and transforms the file itself, along with some meta-data, into request parameters. It actually uses a wrapper around the Servlet request to make this happen, but you don't really need to know that, do you? Table 3.1 shows the request parameters that are added by this fileUpload interceptor.

Table 3.1 Request Parameters Exposed by the FileUploadInterceptor

Parameter name	Parameter type and value
[file name from form]	File - the uploaded file itself
[file name from form]ContentType	String - the content type of the file
[file name from form]FileName	String - the name of the uploaded file, as stored on the server

After the fileUpload interceptor has exposed the parts of the multi-part request as regular looking request parameters, it's time for the next interceptor in the stack to do its work. Conveniently, the next interceptor is the params interceptor. When the params interceptor fires, it simply moves all of the request parameters, including those listed in Table 3.1, onto the action object. Thus, all a developer needs to do in order to conveniently receive the file upload, is add JavaBeans properties to her action object that match the names in the above table.

Pullout: We should take a moment to note the elegant use of interceptors as demonstrated by the fileUpload interceptor. As we have said, the Struts 2 framework tries desperately to keep its action components as clean as possible. A large part of this effort consists of the use of interceptors to layer crosscutting tasks away from the core processing tasks of the action itself; the fileUpload interceptor demonstrates this by providing a re-useable encapsulation of the processing of multi-part requests that injects the processed upload data into the action object's JavaBeans setter methods.

We have also referred to the role of interceptors in terms of preprocessing and post-processing. In the case of the fileUpload interceptor, the preprocessing is the transformation of the multi-part request into regular looking request parameters that the params interceptor will automatically move onto our action. The post-processing comes when the interceptor gets the chance to fire again after our action, an opportunity taken by the fileUpload interceptor to dispose of the temporary version of the uploaded file.

3.5.4 Looking at the Struts 2 Portfolio example code

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

The Struts 2 Portfolio uses this file upload mechanism to upload the new images to the portfolio. We should look at the code that implements this feature. The first part of such a task is presenting the user with a form through which they can upload their file. You can visit this form page by logging in as a user on the Chapter 3 version of the Struts 2 Portfolio sample application. After you log in, choose to add a new picture to the portfolio. You will see a page presenting you with a simple form to upload an image. The following code snippet, from chapterThree/ImageUploadForm.jsp , shows the markup from that creates the form you see.

```
<h4>Complete and submit the form to create your own portfolio.</h4>

<s:form action="ImageUpload" method="post" enctype="multipart/form-data">

    <s:file name="pic" label="Picture"/>

    <s:submit/>

</s:form>
```

When we create this form we have to take note of a couple of points. First, note that we are using Struts 2 tags to build the form. We will cover the Struts 2 tag library in Chapters 6 and 7 of this book. For now, just accept that this tag generates the HTML markup of a form that allows the user to upload a file. Next, note that we set the encoding type of the form to “multipart/form-data.” This important attribute signals to the framework that the request needs to be handled as a upload. Without this setting, it won't work. Finally, note that the name attribute provided to the file tag, will be the name attribute under which the file will be submitted by the form. This detail is important because you will use this name to build the JavaBeans properties that will receive the upload data.

Now that we have the JSP to present the form, let's see the action that will receive and process the upload. First, make sure that the package to which your action belongs is extending the struts-default package so that it inherits the default interceptor stack, and, thus, the fileUpload interceptor. Listing The following snippet from our manning/chapterThree/chapterThree.xml file shows that we have done this.

Listing 3.12 Extending the struts-default Package in Order to Inherit File Upload Processing

```
<package name="chapterThreeSecure" namespace="/chapterThree/secure"
extends="struts-default">

    . . .

    <action name="AddImage" >
        <result>/chapterThree/ImageUploadForm.jsp</result>
    </action>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```
<action name="ImageUpload" class="manning.chapterThree.ImageUpload">
    <result>/chapterThree/ImageAdded.jsp</result>
    <result name="input">/chapterThree/ImageUploadForm.jsp</result>
</action>

...

</package>
```

This is our package of secure actions for the Struts 2 Portfolio. We haven't added security yet, but we know that these actions will require security of some kind so we've separated them into a separate package. We'll add the security with a custom interceptor in the Chapter 4.

With the interceptor in place, we just need to add JavaBeans properties to our action object that match the parameter names as seen in Table 3.1. We've already seen that our file will be submitted under the name “pic.” Using the naming conventions in Table 3.1 we can easily derive the JavaBeans property names that we need to implement. Listing 3.13 shows the JavaBeans properties implemented by the `manning.chapterThree.ImageUpload` class.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Listing 3.13 The JavaBeans Properties that will Receive the Uploaded File and Meta-data

```
File pic;
String picContentType;
String picFileName;

public File getPic() {
    return pic;
}

public void setPic(File pic) {
    this.pic = pic;
}

public String getPicContentType() {
    return picContentType;
}

void setPicContentType(String picContentType) {
    this.picContentType = picContentType;
}

public void setPicFileName(String picFileName) {
    this.picFileName = picFileName;
}

public String getPicFileName() {
    return picFileName;
}
```

Of course, you're not obligated to implement all of these. If you choose not to implement some of them, you just won't receive the data. No harm, no foul. At any rate, using the fileUpload interceptor is about as easy as writing these JavaBeans properties. Thanks to the separation of the upload logic, the action's work itself is quite simple. As shown in the following code snippet, from the ImageUpload action, the action can focus on the task at hand.

```
public String execute(){

    getPortfolioService().addImage( getPic() );
    return SUCCESS;
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

}

3.5.5 Multiple files and other settings

Uploading multiple files with the same parameter names is also supported. All you have to do is change your action fields to arrays: that is, File becomes File[], and the two strings become string arrays. The three arrays are always the same length, and their order is the same, meaning that index 0 for all three arrays represents the same file and file meta-data. There are also many other configurable parameters regarding the fileUpload interceptor ranging from the maximum file size to the implementation of the multi-part request parser that will be used to handle the request. In general, the extreme flexibility of the Struts 2 framework makes it impossible to provide a complete coverage of all the details in a book such as this. Actually, this book strives to filter out as much of the extraneous details as possible in order to make the concepts of the framework more visible. For such details and minutia the Struts 2 web site serves as a good reference.

3.6 Summary

In this chapter we have learned a lot about building Struts 2 actions. We began our tour of this important Struts 2 component by examining the role of the action within the framework. Actions have to do three things. First, and foremost, the encapsulate the framework's interaction with the model. This means, ultimately, that the calls to the business logic and data tier will be found within the execute() method of the action class. The second job of the action is to serve as the data transfer object for the request processing. We suspect we've made this point particularly clear by now. Finally, the action also takes responsibility for returning a control string that will be used by the framework to select the appropriate result component for rendering the view back to the user.

We also saw how to package our action components into Struts 2 packages. These packages help provide a logical organization to your applications framework components, actions in particular. Using the package structure we can do several important things. We can map URL namespaces to groups of actions. We can also take advantage of the inheritance mechanisms of packages to define re-useable groups of framework components. We've already much use of this feature by having our Struts 2 Portfolio packages extend the built-in struts-default package in order to take advantage of its default interceptor stack, among other things.

We then showcased a couple of key players provided by the framework to ease your work. First we saw the Action interface which provides some important definitions of constants that the framework uses for commonly used control strings. After that we took a long look at the functionality provided by the ActionSupport class. This helpful class implements several important interfaces, and cooperates with several key interceptors from the default stack, to provide built-in implementations of such valuable web application domain tasks as validation and a basic form of internationalization. We demonstrated all of this with our Struts 2 Portfolio sample application.

One of the more important considerations when implementing your own Struts 2 actions will be the method of data transfer that you use. We have seen in this chapter that several options are available. We covered two methods that both implement JavaBeans properties on the action object itself. The first of these matches simple properties to individual parameters on the incoming request. The next JavaBeans properties method provides properties that are backed by

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

complex domain objects. Finally, we saw that you can use an entirely different method to expose your complex domain objects by implementing ModelDriven actions. The choice of which method to use will largely depend upon the requirements of your project and the action at hand. Ultimately, it's powerful just to have such a choice.

We rounded off the chapter with a case study. We looked at using one of the framework's built-in interceptors to add a file upload action to our sample application. While we obviously saw that uploading files with Struts 2 can be quite easy, we also tried to point out some important lessons that this example demonstrates about the framework itself. In particular, we have tried to show what we mean when we say the framework tries to provide a clean implementation of MVC. In particular, the file upload example shows how proper cooperation between interceptors and actions can provide a web application we re-useable, and flexible encapsulations of crosscutting tasks, and super clean actions.

Well, we know you know what an action is by now. Next up, a detailed look at interceptors.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4 Adding Workflow with Interceptors

In the previous chapter, we learned a great deal about the action component of the Struts 2 framework. From a developer's daily working perspective, the action component may very well be the heart and soul of the framework. But working silently in the background are the true heroes of the hour, the interceptors. In truth, the interceptors are responsible for most of the processing done by the framework. Built-in interceptors handle most of the fundamental tasks ranging from data transfer and validation to exception handling. Due to the rich set of the built-in interceptors, you might not need to develop your own interceptor for quite some time. Nonetheless, the importance of these core Struts 2 components can not be underestimated. We can honestly say that without an understanding of how interceptors work, you will never truly understand Struts 2.

After such a bold statement, we have no choice but to back it up with a detailed explanation of interceptors and the role they play in the framework. This chapter will begin by clarifying that architectural role with a brief conceptual discussion. We will then dissect a couple of simple interceptors from the default stack (just so we can see what's inside!), provide a user guide level discussion of all the built-in interceptors, and end by creating a custom interceptor to provide an authentication service to our Struts 2 Portfolio sample application.

Incidentally, if you are one of those who would prefer to see a working code sample before hearing the explanation, feel free to skip ahead to the last section of this chapter, where we build a custom interceptor for the Struts 2 Portfolio application. After seeing one in action, you can always come back here for the theory. Just don't forget to come back!

4.1 Why intercept requests?

Earlier in this book we described the Struts 2 framework as a second generation MVC framework. We said that this new framework leveraged the lessons learned by the first generation of MVC based frameworks to implement a super clean architecture. The interceptors play a very important role in allowing the framework to achieve such a high level of separation of concerns. In this section, we will take a closer look at the way in which interceptors provide a powerful tool for encapsulating the kinds of tasks that have traditionally been an architectural thorn in the developer's side.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.1.1 Cleaning up the MVC

From an architectural point of view, interceptors have revolutionized the level of separation we can achieve when trying to isolate the various concerns of our web applications. In particular, the interceptor serves to remove crosscutting tasks from our action components. When we try to describe the kinds of tasks that interceptors implement, we usually say something like crosscutting, or preprocessing and post-processing. These terms may sound a bit vague now, but they won't by the time we finish this chapter.

Logging is a very typical crosscutting concern. In the past, you might have had a logging statement in each of your actions. While this seemed a natural place for placing a logging statement, it's really not a part of the action's interaction with the model. In reality, logging is just some administrative stuff that we want done for every request that the system processes. We call this crosscutting because this task is not specific to a single action. It cuts across a whole range of actions. As software engineers, we should instantly see this as an opportunity to raise the task to a higher layer that can sit above, or in front of if you will, any number of requests that require logging. The bottom line is that we have the opportunity to remove the logging from the action, thus creating cleaner separation of concerns.

Some of the tasks undertaken by interceptors are more easily understood as being preprocessing and post-processing tasks. These tasks are still technically crosscutting, and we recommend not worrying about the semantics of these terms. We present these new terms mostly to give you some ideas about the specific types of tasks handled by interceptors. A good example of a preprocessing task would be the data transfer that we have already become familiar with. This task is achieved by the `ParametersInterceptor`. Nearly every action will need to have some data transferred from the request parameters onto its domain specific properties. This must be done before the action fires. This task can be seen as merely being a preparation for the actual work of the action. From this perspective we can call it a preprocessing task. This is a perfect task for the interceptor. Again, this increases the purity of the action component by removing code that can not be strictly seen as part of a specific action's core work.

No matter whether we call the task crosscutting or preprocessing, the conceptual mechanics of the interceptor are quite clear. Instead of having a simple controller invoking an action directly, we now have a component that sits between the controller and the action. In Struts 2, no action is invoked in isolation. The invocation of an action is a layered process that always includes the execution of a stack of interceptors prior to and after the actual execution of the action itself. Rather than invoke the action's `execute` method directly, the framework creates an object called an `ActionInvocation` which encapsulates the action and all of the interceptors that have been configured to fire before and after that action actually executes. Figure 4.1 illustrates the encapsulation of the entire action execution process in the `ActionInvocation` class.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

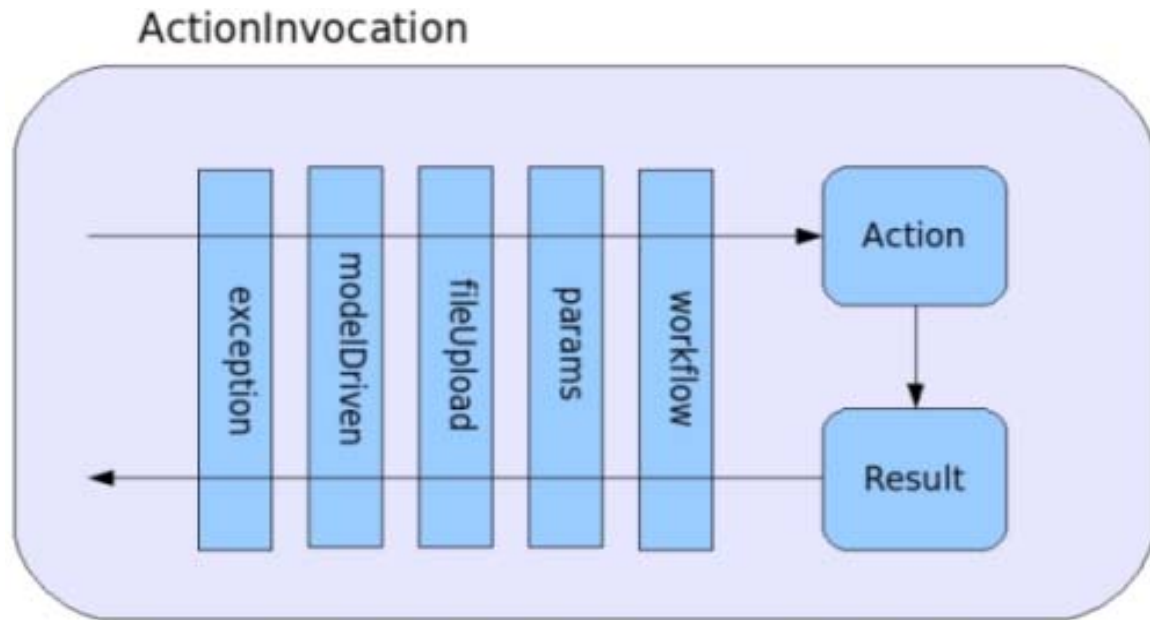


Figure 4.1 The ActionInvocation Encapsulates the Execution of an Action with its Associated Interceptors and Results

As you can see in Figure 4.1, the invocation of an action must first travel through the stack of interceptors associated with that action. Here we have presented a simplified version of the `defaultStack`. The `defaultStack` includes such tasks as the transfer of request params onto our action and file uploading. Figure 4.1 represents the normal workflow; none of the interceptors have diverted the invocation. This action will ultimately execute and return a control string that selects the appropriate result. After the result executes, each of the interceptors, in reverse order, gets a chance to do some post-processing work. As we will see, the interceptors have access to the action and other contextual values. This allows them to be aware of what is happening in the processing. For instance, they can examine the control string returned from the action to see what result was chosen.

One of the powerful functional aspects of interceptors is their ability to alter the workflow of the invocation. As we noted, Figure 4.1 depicts an instance where none of the interceptors have intervened in the workflow, thus allowing the action to execute and determine itself which result should render the view. Sometimes, however, one of the interceptors will determine that the action shouldn't execute. In these cases, the interceptor can halt the workflow by itself returning a control string. Take the `workflow` interceptor for example. As we have seen, the `workflow` interceptor does two things. First it invokes the `validate()` method on the action, if the action has implemented the `Validateable` interface that is. Next, it checks for the presence of error messages on the action. If errors are present, it returns a control string itself and, thus, stops further execution. The action will never fire. The next interceptor in the stack will not even be invoked. By returning the control string itself, the interceptor causes control to return back up the chain, giving each interceptor above the chance to do some post processing. Finally, the result that matches the returned control string will render the view. In the case of the `workflow` interceptor that has found error messages on the action, the control string is "input" which typically maps back to the form page that submitted the invalid data.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

As you might suspect, the details of this invocation process are a bit thorny. In fact, they involve a bit of recursion. As with all recursion, it will seem harmless once we see the details. And we will look at them shortly. But first we need to talk about the benefits we gain from using interceptors.

4.1.2 Reaping the benefits

While layering always makes our software cleaner, which helps with readability and testing, it also provides flexibility. Once we have broken these crosscutting, preprocessing and post-processing tasks into manageable units, we can really start to do some cool stuff with them. The two primary benefits we gain from this flexibility are re-use and configuration.

Everyone wants to re-use software. Perhaps this is the number one goal of all software engineering. Re-use is a bottom line issue from both business and engineering perspectives. Re-use means time, money, and maintainability. It makes everyone happy. And, actually, achieving re-use is kind of simple. We just need to isolate the logic that we want to re-use in a cleanly separated unit. Once we have isolated the logic in an interceptor, we can drop it in anywhere we like, easily applying it to whole classes of actions. This is much more exciting than clean architectural lines, but really it's the same thing. We've already been benefiting from code re-use by inheriting the `defaultStack`. Using the `defaultStack` allows us to re-use the data transfer code written by the Struts 2 developers, along with their validation code, their internationalization code, and so forth.

In addition to the benefits of code re-use, the layering power of interceptors gives us another very important benefit. Once we have these tasks cleanly encapsulated in interceptors, we can, in addition to merely reusing them, easily reconfigure their order and number. While the `defaultStack` provides a common set of interceptors, arranged in a common sequence, to serve the common functional needs of most requests, we can easily re-arrange them to meet varying requirements. We can even remove and add particular interceptors as we like. We can even do this on a per action basis, but this is seldom necessary. In our Struts 2 Portfolio application, we will develop an authentication interceptor and add it to the stack of interceptors which fires when the actions in our secure package are invoked. The flexible nature of interceptors allows us to easily customize request processing for the specific needs of certain requests all while still taking advantage of code re-use.

Warning: Struts 2 is extremely flexible. This strength is clearly what separates it from many of its competitors. But, as we have mentioned, this can also be very confusing when you first begin to use the framework. Thankfully, Struts 2 provides a strong set of intelligent defaults that allow developers to build most standard functionality without needing to think about the many ways in which they can modify the framework and its core components. In the case of interceptors, one of the framework's most flexible components, the `defaultStack` should serve in the vast majority of cases.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.1.3 Developing interceptors?

Despite their importance, many developers just won't write very many interceptors. In fact, most of the common tasks of the web application domain have already been written into the built-in Struts 2 interceptors. Even if you never write an interceptor yourself, it's still very important to understand what they are and how they do what they do. If this chapter weren't core to understanding the framework, we would have placed it at the end of the book. We put this material here because we believe that understanding interceptors is absolutely necessary to successfully leveraging the power of the framework. First of all, you need to be familiar with the built-in interceptors and you need to know how to arrange them to your liking. Secondly, debugging the framework can truly be confusing if you don't understand how the requests are processed. We think that interceptors ultimately provide a simpler architecture that can be more easily debugged and understood. However, many developers may find them counter intuitive at first.

With that said, when you do find yourself writing your own custom interceptors, you will truly begin to enjoy the Struts 2 framework. As you develop your actions, keep your eyes out for any tasks that can be moved out to the interceptors. As soon as you do, you'll be hooked for life. But first, we should see how they actually work.

4.2 Interceptors in action

Now we will take a quick look at how the interceptors actually run. We'll have a look at the interceptor interface, and learn the mysterious process by which a interceptor is fired. Along the way will meet the boss man, the `ActionInvocation` in which the framework encapsulates the entire processing of a request. We'll also take the time to look inside the actual code of two of the built-in Struts 2 interceptors, just to keep it real. But first let's start with the boss man.

4.2.1 The guy in charge: ActionInvocation

A few paragraphs back we introduced the `ActionInvocation`. While you will almost certainly never have to work directly with this class, a high level understanding of it is key to understanding interceptors. In fact, knowing what `ActionInvocation` does is equivalent to knowing how Struts 2 handles requests; it's very important! As we said before, the `ActionInvocation` encapsulates all the processing details associated with the execution of a particular action. When the framework receives a request it first must decide to which action the URL maps. An instance of this action is added to a newly created instance of `ActionInvocation`. Next, the framework consults the declarative architecture, as created by the application's XML or Java annotations, to discover which interceptors, and in what sequence, should be used with this action. References to these interceptors are added to the `ActionInvocation`. In addition to these central elements, the `ActionInvocation` also holds references to other important information like the Servlet request objects and a map of the results available to the action. Now, let's look at how the process of invoking an action occurs.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.2.2 *How the interceptors fire*

Now that the `ActionInvocation` has been created and populated with all the objects and information it needs, we can start the invocation. The `ActionInvocation` exposes the `invoke()` method which is called by the framework to start the execution of the action. When the framework calls this method, the `ActionInvocation` starts the invocation process by executing the first interceptor in the stack. Note, the `invoke()` method doesn't necessarily map to the first interceptor. It is the responsibility of the `ActionInvocation` itself to keep track of what stage the invocation process has reached, and pass control to the appropriate interceptor. It does this by calling that interceptor's `intercept()` method. Simple enough.

But now for the tricky part. Continued execution of the subsequent interceptors, and ultimately the action, occurs through recursive calls to the `ActionInvocation`'s `invoke()` method. Each time `invoke()` is called, `ActionInvocation` consults its state and executes whichever interceptor comes next. When all of the interceptors have been invoked, the `invoke()` method will cause the action itself to be executed. If this is cooking your noodle, hang in there. It will clear up momentarily.

Why do we call it a recursive process? Let's have a look. The framework itself starts the process by making the first call to the `ActionInvocation` object's `invoke()`. `ActionInvocation` hands control over to the first interceptor in the stack by calling that interceptor's `intercept()` method. Importantly, the `intercept()` method takes the `ActionInvocation` instance itself as a parameter. During its own processing, the interceptor will call `invoke()` on this object to continue the recursive process of invoking successive interceptors. Thus, in normal execution, the invocation process tunnels down through all of the interceptors until the action is finally executed. Again, the `ActionInvocation` itself maintains the state of this process internally so it always knows which interceptor comes next.

Now, let's look at what an interceptor can do. When an interceptor fires, it has a three-stage, conditional execution cycle.

Do some preprocessing.

Pass control on to successive interceptors, and ultimately the action, by calling `invoke()`, or divert execution by itself returning a control string.

Do some post-processing.

Looking at some code will make these three stages more concrete. The following code snippet shows the `intercept()` method of the `TimerInterceptor`, one of the built-in interceptors.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

public String intercept(ActionInvocation invocation) throws Exception {

    long startTime = System.currentTimeMillis();
|#1

    String result = invocation.invoke();
| #2

    long executionTime = System.currentTimeMillis() - startTime;
|#3

    ... log the time ...
|#3

    return result;
|#3
}

```

(annotation) <#1 Do Some Pre-processing >
 (annotation) <#2 Pass Control to the Rest of the Action Invocation Process>
 (annotation) <#3 Do Some Post-processing >

The `TimerInterceptor` times the execution of an action. The code is quite simple. The `intercept()` method, defined by the `Interceptor` interface, is the entry point into a interceptor's execution. Note that the `intercept` method received the `ActionInvocation` instance. When the `intercept` method is called, the interceptor's preprocessing phase consists of recording the start time. Next, the interceptor must decide whether it will pass control on to the rest of the interceptors and action. Since this interceptor has no reason to halt the execution process, it always calls `invoke()`, thus passing control to whoever comes next in the chain.

After calling `invoke()`, the interceptor waits wait for the return of this method. `Invoke()` returns a result string that indicates what result was rendered. Note, while this string tells the interceptor which result was rendered, it doesn't indicate whether the action itself fired or not. It's entirely possible that one of the deeper interceptors altered workflow by returning a control string itself without calling `invoke()`. Either way, when `invoke()` returns, a result has already been rendered. In other words, the response page has already been sent back to the client. An interceptor could implement some conditional post-processing logic that uses the result string to make some decision, but it can't stop or alter the response at this point. In the case of the `TimerInterceptor`, we don't care what happened during processing, so we don't look at the string.

What kind of post-processing does the `TimerInterceptor` do? It calculates the time that has passed during the execution of the action. It does this simply by taking the current time and subtracting the previously recorded start time. This is a simple task. When finished, it must finally return the control string that it received from `invoke()`. Doing this causes the recursion to travel back up the chain of interceptors. These outer interceptors will then have the opportunity to conduct any post-processing they might be interested in.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Oh, my! Provided all of that sunk in, you're noodle should definitely be cooked by now. But hopefully in a good way -- by the vast possibilities that such an architecture allows! When contemplating the wide range of tasks that can be implemented in the reusable and modular interceptor, consider this short sampling of the available opportunities:

During the **preprocessing** phase, the interceptor can be used to prepare, filter, alter, or otherwise manipulate any of the important data available to it. This data includes all of the key objects and data, including the action itself, that pertain to the current request.

Call `invoke()` or divert workflow. If an interceptor determines that the request processing should not continue, it can return a control string rather than call the `invoke()` method on the `ActionInvocation`. In this manner, it can stop execution and determine itself which result will render.

Even after the `invoke()` method returns a control string, any of the returning interceptors can arbitrarily decide to alter any of the objects or data available to them as part of their **post-processing**. Note, however, that at this point the result has already been rendered.

As we have said, interceptors can be a bit confusing at first. Furthermore, you can probably avoid implementing them yourselves. However, we encourage you to reread these pages until you feel comfortable with the interceptor. Even if you never make one yourself, a solid grasp of the interceptor in action will ease all aspects of your development. Now, let's move on to something simpler – the user guide section of this chapter, wherein we will tell you all about the built-in interceptors that you can easily leverage when building your own applications.

4.3 Surveying the built-in Struts 2 interceptors

Struts 2 comes with a powerful set of built-in interceptors that provide most of the functionality you'll ever want from a web framework. In the introductory portion of this book, we said that a good framework should automate most of the routine tasks of the web application domain. The built-in interceptors provide this automation. We have already seen several of these and used them in our Struts 2 Portfolio sample application. The ones we have used have all been from the `defaultStack` that we have inherited by extending the `struts-default` package. While this default stack is quite useful, the framework actually comes with more interceptors and pre-configured stacks than just that one. In this section we will give you a introduction to the most commonly used built-in interceptors. In the next section we will show you how to declare which of these interceptors should fire for your actions, and even how to arrange their order. But first, let's explore the offerings.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.3.1 Utility interceptors

First, we will look at some utility interceptors. These interceptors provide simple utilities to aid in development, tuning, and troubleshooting.

Timer

This simple interceptor merely records the duration of an execution. Note, position in the interceptor stack determines what this is actually timing. If you place this interceptor at the heart of your stack, just before the action, then it will time the action's execution itself. If you place it at the outer most layer of the stack, it will be timing the execution of the entire stack, as well as the action. Here's the output.

```
INFO: Executed action [/chapterFour/secure/ImageUpload!execute]
took 123 ms.
```

Logger

This interceptor provides a simple logging mechanism that logs a statement entry statement during preprocessing and an exit statement during post-processing.

```
INFO: Starting execution stack for action
/chapterFour/secure/ImageUpload

INFO: Finishing execution stack for action
/chapterFour/secure/ImageUpload
```

This can be useful for debugging. Again, note that the position in which you put this in the stack can change the nature of the information you learn from these simple statements. This interceptor serves as a good demonstration of an interceptor that does processing both before and after the action executes.

4.3.2 Data transfer interceptors

As we have already seen, interceptors can be used to handle data transfer. In particular, we have already seen that the params interceptor from the defaultStack moves the request parameters onto the JavaBeans properties we expose on our action objects. There are actually several other interceptors that can move data onto our actions. These other interceptors can move data from other locations, such as from parameters defined in the XML configuration files.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Params (defaultStack)

This familiar interceptor provides one of the most integral functions of the framework. It transfers the request parameters onto properties exposed by the `ValueStack`. We have also discussed how the framework uses OGNL expressions, embedded in the name attributes of your form's fields, to map this data transfer onto those properties. In the action chapter, we explored techniques for having this transfer move data onto properties exposed directly on our actions as well as on domain model objects with `ModelDriven` actions. The `params` interceptor doesn't know where the data is ultimately going, it just moves it to the first matching property it can find on the `ValueStack`. So, how do the right objects get onto the `ValueStack` in time to receive the data transfer? As we learned in the last chapter, the action is always put on the `ValueStack` at the start of a request processing cycle. The model, as exposed by the `ModelDriven` interface, is moved onto the `ValueStack` by the `modelDriven` interceptor, discussed below.

We will fully cover the enigmatic `ValueStack`, and the equally enigmatic OGNL, in the next chapter when we delve into the details of data transfer and type conversion.

Static-params (defaultStack)

This interceptor also moves parameters onto properties exposed on the `ValueStack`. The difference is the origin of the parameters. The parameters that this interceptor moves are defined in the action elements of the declarative architecture. For example, suppose you have an action defined like this in one of your declarative architecture XML files:

```
<action name="exampleAction" class="example.ExampleAction">
  <param name="firstName">John</param>
  <param name="lastName">Doe</param>
</action>
```

The `static-params` interceptor is called with these two name-value pairs. These parameters are moved onto the `ValueStack` properties just as in the `params` interceptor. Note, again order matters. In the default stack, the `static-params` interceptor fires before the `params` interceptor. This means that the request parameters will override values from the XML `param` element. You could, of course, change the order of these interceptors.

Autowiring

This interceptor provides an integration point for using Spring to manage your application resources. We list it here because it is technically another way to set properties on your action. Since this use of Spring is such an important topic, we save it for a fuller treatment in Chapter 10 which covers integration with such important technologies.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Servlet-config (defaultStack)

The `Servlet-config` interceptor provides a clean way of injecting various objects from the Servlet API into your actions. This interceptor works by setting the various objects on setter method exposed by interfaces that the action must implement. The following interfaces are available for retrieving various objects related to the Servlet environment. Your action can implement any number of these.

`ServletContextAware` – sets the `ServletContext`
`ServletRequestAware` – sets the `HttpServletRequest`
`ServletResponseAware` – sets the `HttpServletResponse`
`ParameterAware` – sets a map of the request parameters
`RequestAware` – sets a map of the request attributes
`SessionAware` – sets a map of the session attributes
`ApplicationAware` – sets a map of application scope properties
`PrincipalAware` – sets the `Principal` object (security)

Each of these interfaces contains one method — a setter — for the resource in question. These interfaces are found in the Struts 2 distribution's `org.apache.struts2.interceptor` package. As with all of the data injecting interceptors that we've seen, the `Servlet-config` interceptor will put these objects on your action during the preprocessing phase. Thus, when your action executes, the resource will be available. We will demonstrate using this injection later in this chapter when we build our custom authentication interceptor; the `Login` action that will work with the authentication interceptor will implement the `SessionAware` interface. We should note that best practices recommend avoiding use of these Servlet API objects as they bind your action code to the Servlet API. After all the work the framework has done to separate you from the Servlet environment, you would probably be well served by this advice. Nonetheless, you'll sometimes want to get your hands on these important Servlet objects. Don't worry, it's a natural urge.

FileUploads (defaultStack)

We covered the `fileUpload` interceptor in depth in the previous chapter. We note it briefly here for completeness. The `fileUpload` interceptor transforms the files and meta-data from multi-part requests into regular request parameters so that they can be set on the action just like normal parameters.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.3.3 Workflow interceptors

Some interceptors provide functionality to alter the workflow of the action's execution. By workflow, we mean the logic of the processing. In normal workflow, all interceptors will fire, doing a variety of preprocessing chores, then the action will conduct its business via the `execute()` method. Workflow interceptors somehow intervene and alter this normal path. Sometimes only slightly, and sometimes quite drastically.

Workflow (defaultStack)

We have already used the workflow interceptor. This interceptor works with our actions to provide data validation and the subsequent workflow alteration if a validation error occurs. Since we have already used this interceptor, we will leverage our familiarity to learn more about how interceptors work by taking a look at the code that alters execution workflow. Listing 4.1 shows the code from this important interceptor.

Listing 4.1 Altering Workflow from within an Interceptor

```
public String intercept(ActionInvocation invocation)
    throws Exception {

    Action action = invocation.getAction();           | #1
    if (action instanceof Validateable) {             | #1   | #2
        Validateable validateable = (Validateable) action; | #1   | #2
        validateable.validate();                      | #1   | #1
    | #2
    }
    if (action instanceof ValidationAware) {          | #1   | #3
        ValidationAware validationAwareAction =       | #1   | #3
        ValidationAware) action;                     | #1   | #1
    | #3
        if (validationAwareAction.hasErrors()) {      | #1   | #3
            return Action.INPUT;                     | #1   | #3
        | #4
        }
    }
    | #1
    return invocation.invoke();                      | #5
}

(annotation) <#1 Pre-processing >
(annotation) <#2 If Action Implements Validateable, Call Validate()>
(annotation) <#3 If Action is ValidationAware, Check for Errors >
(annotation) <#4 If Errors Exist, Return to Input Page>
(annotation) <#5 If No Errors Exist, Continue Invocation of Action>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

If you recall, the actions of our Struts 2 Portfolio use a form of validation implemented in a couple of interfaces that cooperate with the workflow interceptor. The `Validateable` interface exposes the `validate()` method which contains our validation logic and is invoked by the workflow interceptor as seen in Listing 4.1. Next, the `ValidationAware` interface provides methods to store the error messages that are produced when something fails validation. After calling `validate()`, the workflow interceptor checks for error messages by invoking the `hasErrors()` method. If some are present, the workflow interceptor takes the rather drastic step of completely halting execution of the action. Acute workflow alteration if you will. It does this, as you can see, by returning its own `INPUT` control string. Further execution stops immediately. The `INPUT` result is rendered, and post-processing occurs as control climbs back out of the interceptor stack.

As we have seen with our Struts 2 Portfolio actions, we can get built in implementations of these two interfaces by extending the `ActionSupport` convenience class.

The workflow interceptor also introduces another important interceptor concept. Using params to tweak the execution of the interceptor. After we finish covering the built-in interceptors we will cover the syntax of declaring your interceptors and interceptors stacks in the next section. At that time, we will learn all about setting and overriding parameters. For now, we'll just note the parameters that an interceptor can take. The workflow interceptor can take several parameters.

```
alwaysInvokeValidate ( true or false: defaults to true which means that validate()
will be invoked )
inputResultName ( name of the result to choose if validation fails: defaults to
Action.INPUT )
excludeMethods ( names of methods for which the workflow interceptor should not
execute )
```

These should all be pretty straightforward. Note, the workflow interceptor configured in the `defaultStack` is passed a list of `excludeMethods` parameters as seen in the following snippet from `struts-default.xml`.

```
<interceptor-ref name="workflow">
  <param name="excludeMethods">input,back,cancel,browse</param>
</interceptor-ref>
```

This list of exclude methods is meant to support actions that expose methods other than `execute()` for various processing tasks related to the same data object. For instance, imagine you want to use the same action to pre-populate as well as process a form. A common scenario is to combining the Create Read Update and Delete (CRUD) functions pertaining to a single data object into a single action. There are many reasons for wanting to do this, and we will present these in the best practices section of the book, Chapter 14. For now, note that the main benefit of such a strategy is the reuse of code across the various functions. One difficulty with this strategy is that the process that pre-populates the form can't be validated because there is no data yet. To

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

handle this problem, we can put the pre-population code into an input method and list this method as one that should be excluded from the workflow interceptor's validation measurements.

Several other interceptors take `excludeMethods` and `includeMethods` parameters to achieve similar filtering of their processing. We will note when these parameters are available for the interceptors that we cover in this book. In general though, you should be on the look out for such parameters any time you are dealing with an interceptor for which it seems logical that such a filtering would exist.

Validation (defaultStack)

Another important interceptor, which most applications will want to use, is the validation interceptor. Note, this interceptor has nothing to do with the `Validateable` interface and its `validate()` method. The `Validateable` interface provides a programmatic validation mechanism; you must write the validation Java code yourself in you `validate()` method, as we have seen. The validation interceptor, on the other hand, is part of the Struts 2 Validation Framework and provides a declarative means to validate your data. The Validation Framework provided by Struts 2 allows you to use both XML files and Java annotations to describe the validation rules for your data. Since the Validation Framework is such a rich topic, Chapter 11 is dedicated to it.

For now we should note that the validation interceptor, like the `Validateable` interface, works in tandem with the workflow interceptor. Recall that the workflow interceptor called the `validate()` method of the `Validateable` interface to cause execution of validation code before it checked for validation errors. The process is the same. In this case, the validation interceptor runs before the workflow interceptor. The validation interceptor is the entry point into the Validation Framework's processing. When the Validation Framework does its work, it will store validation errors using the same `ValidationAware` methods that your own `validate()` code does. When the workflow interceptor checks for error messages, it doesn't know whether they were created by the Validation Framework or the validation code invoked through the `Validateable` interface. In fact, it doesn't matter. You could even use both methods of validation if you liked. Either way, if errors are found, the workflow interceptor will divert workflow back to the input page.

Prepare (defaultStack)

The prepare interceptor provides a generic entry point for arbitrary workflow processing that you might want to add to your actions. The concept is quite simple. When the prepare interceptor executes, it looks for a `prepare()` method on your action. Actually, it checks to see if your action implements the `Preparable` interface, which defines the `prepare()` method. If your action is `Preparable`, the `prepare()` method is invoked. This allows for any sort of preprocessing to occur. Note, while the prepare interceptor has a specific place in the default stack, you can define your own stack if you need to move the prepare code to a different location in the sequence.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

The prepare interceptor is somewhat flexible as well. For instance, you can define special prepare methods for the different execution methods on a single action. As we said above, sometimes you will want to define more than one execution entry point on your action. In addition to the `execute()` method, you might define an `input()` method and an `update()` method. In this case, you might want to define specific preparation logic for each of these methods. If you have implemented the `Preparable` interface, you can also define preparation methods, named according to the following conventions, for each of your action's execution methods.

Action Method Name	Prepare Method 1	Prepare Method 2
<code>input()</code>	<code>prepareInput()</code>	<code>prepareDoInput()</code>
<code>update()</code>	<code>prepareUpdate()</code>	<code>prepareDoUpdate()</code>

Note, two naming conventions are provided. You can use either one. The use case is simple. If you `input()` method is being invoked, the `prepareInput()` will be called by the prepare interceptor, thus giving you an opportunity to execute some preparation code specific to the input processing. Note, the prepare method itself will always be called by the prepare interceptor regardless of the action method being invoked. Its execution comes after the specialized prepare method. If you like, you can turn off the prepare method invocation with a parameter passed to the prepare interceptor:

`alwaysInvokePrepare` - Default to `true`.

The `Preparable` interface can be helpful for setting up resources or values before your action is executed. For instance, if you have a drop-down list of available values that you look up in the database, you may want to do this in the `prepare()` method so that the values will be populated for rendering to the page even if the action isn't executed because, for instance, the workflow interceptor found error messages.

Model-driven (defaultStack)

We've probably already covered the model-driven interceptor enough for one book. We'll just make a couple of brief notes here for the sake of consistency. The model-driven interceptor is considered a workflow interceptor because it alters the workflow of the execution by invoking `getModel()`, if present, and setting the model object on the top of the `ValueStack` where it will receive the parameters from the request. This alters workflow because the transfer of the parameters, by the `params` interceptor, would otherwise be directed onto the action object itself. By placing the model over the action in the `ValueStack`, the model-driven interceptor thus alters workflow. This concept of creating a interceptor that can conditionally alter the effective functionality of another interceptor without direct programmatic intervention demonstrates the power of the layered interceptor architecture. When thinking of ways to add power to your own applications by writing custom interceptors, this is a good model to follow.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.3.4 Miscellaneous interceptors

A few interceptors don't quite fit into any specific classification, but are important or useful nonetheless. The following interceptors range from the core interceptors from the defaultStack to wonderful built-in interceptors that provide cool functionality of the bells-and-whistles sort.

Exception (defaultStack)

This important interceptor lays the foundation for rich exception handling in your applications. The exception interceptor comes first in the defaultStack, and should probably come first in any custom stacks you might create yourself. The exception interceptor will catch exceptions and map them, by type, to user defined error pages. Its position at the top of the stack guarantees that it will be able to catch all exceptions that may be generated during all phases of the action invocation.

The Struts 2 Portfolio uses the exception interceptor to route all exceptions of type java.lang.Exception to a single, somewhat unpolished, error message page. We've implemented this in the chapterFourPublic package. The following snippet shows the code from the chapterFour.xml file that sets up the exception handling.

```
<global-results>
  <result name="error">/chapterFour/Error.jsp</result>
</global-results>

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

First, we define a global result. We need to do this because this error page is not specific to one action and global results are available to all actions in the package. The exception mapping element tells the exception interceptor which result to render for a given exception. When the exception interceptor executes during its post-processing phase, it will catch any exception that has been thrown and map it to a result. Before yielding control to the result, the exception interceptor will create an ExceptionHolder object and place it on top of the ValueStack. The ExceptionHolder is a wrapper around an exception that exposes the stack trace and the exception as JavaBeans properties that you can easily access from a tag in your error page. The following snippet shows our error JSP page.

```
<p><h4>Exception Name: </h4><s:property value="exception" /></p>
<p><h4>What you did wrong:</h4> <s:property value="exceptionStack"
/></p>

<p><h5>Also, please confirm that your Internet is working before
actually
  contacting us.</h4></p>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

We have created an `ErrorProne` action, that automatically throws an exception, so that we can see all of this in action. For a guaranteed application failure, hit the error prone link from the home page of the Chapter Four version of the application. Note, you don't have to have one page catch all exceptions; you can have as many exception mappings as you like, mapping specific exception types to a variety of specific results.

Token and token-session

The token and token-session interceptors can be used as part of a system to prevent duplicate form submissions. Duplicate form posts can occur when users click the Back button to go back to a previously submitted form and then click the button again, or when they click the button more than once while waiting for a response. The token interceptors work by passing a token in with the request which is checked by the interceptor. If the unique token comes to the interceptor a second time, the request is considered as a duplicate. These two interceptors both do the same thing, differing only in the richness of their handling of the duplicate request. You can either show an error page or save the original result to be re-rendered for the user. We will implement this functionality for the Struts 2 Portfolio application in Chapter 13.

Scoped-Model-Driven (defaultStack)

This nice interceptor supports wizard like persistence across requests for your action's model object. This one adds to the functionality of the model-driven interceptor by allowing you to store your model object in, for instance, session scope. This works great for implementing wizards that need to work with a data object across a series of requests. We will also implement this for the Struts 2 Portfolio application in Chapter 13.

ExecAndWait

When a request takes a long time to execute, it's really nice to give the user some feedback. It's impatient users after all that make all of those duplicate requests. While the token interceptors discussed above can technically solve this problem, we should still do something for the user. The `execAndWait` interceptor helps prevent your users from getting antsy. Again, we will go into detail with a example implementation of this functionality in Chapter 13.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.3.5 Built-in stacks

The Struts 2 framework comes with many built-in stacks that provide convenient arrangements of the built-in interceptors. We have been using one of these, the `defaultStack`, for all of our Struts 2 Portfolio packages. We inherit all of this, and other built-in stacks, just by having our packages extend the `struts-default` package defined in `struts-default.xml`. To make things simple, we recommend that you also use the `defaultStack` unless provoked by some clear imperative. Most of the other built-in stacks that you could use are just pared down versions of the `defaultStack`. This paring down is not so much to make more efficient versions of the stacks by eliminating unnecessary interceptors. Rather, the smaller stacks are meant to be modular building blocks for the larger ones. If you find yourself building your own stacks, try to use these modular pieces to help simplify your task. Still, you might ask, Why do we need the `scoped-model-driven` interceptor in the stack if we aren't using it? Valid question, but it turns out the it doesn't really impact performance to have these non-working interceptors in the stack. Messing with the interceptors can be the fastest way to introduce debugging complexity. Ultimately, we always recommend using the built-in path of least resistance as long as possible. While Struts 2 is super flexible, it is also meant to be highly useful right out of the box.

4.4 Declaring interceptors

We can't go much further without learning how to set up our interceptors with the declarative architecture. In this section we will cover the details of declaring interceptors, building stacks, and passing parameters to interceptors. Since most of the interceptors that you will typically need are provided by the `struts-default` package, we will spend a fair bit of time perusing the interceptor declarations made in the `struts-default.xml` file. After we look at the interceptors and stacks from the `struts-default` package, we'll also show how you can specify the interceptors that fire for a given action. As it turns out, you can do this at varying levels of granularity starting with the broad scope of the framework's intelligent defaults and narrowing down to a per action specification of interceptors.

We should also note that, at this point, XML is your only option for declaring your interceptors; the annotations mechanism doesn't yet support declaring interceptors.

4.4.1 Declaring individual interceptors and interceptor stacks

Basically, interceptor declarations consist of declaring the interceptors that are available and then associating them with the actions for which they should fire. The only complication to this process is the creation of stacks which allow you to reference groups of interceptors all at once. Interceptor declarations, like declarations of all framework components, must be contained in a package element. Listing 4.2 shows the individual interceptor declarations from the `struts-default` package of the `struts-default.xml` file.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Listing 4.2 Interceptor Declarations from the struts-default Package

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

    . . .

    <interceptors>                                     | #1
        <interceptor name="execAndWait"
class="ExecuteAndWaitInterceptor" />                | #2
        <interceptor name="exception"
class="ExceptionHandlerInterceptor" />
        <interceptor name="fileUpload" class="FileUploadInterceptor" />
        <interceptor name="i18n" class="I18NInterceptor" />
        <interceptor name="logger" class="LoggerInterceptor" />
        <interceptor name="model-driven" class="ModelDrivenInterceptor" />
        <interceptor name="scoped-model-driven"
class="ScopedModelDrivenInterceptor" />
        <interceptor name="params" class="ParametersInterceptor" />
        <interceptor name="prepare" class="PrepareInterceptor" />
        <interceptor name="static-params"
class="StaticParametersInterceptor" />
        <interceptor name="servlet-config"
class="ServletConfigInterceptor" />
        <interceptor name="sessionAutowiring"
class="SessionContextAutowiringInterceptor" />
        <interceptor name="timer" class="TimerInterceptor" />
        <interceptor name="token" class="TokenInterceptor" />
        <interceptor name="token-session"
class="TokenSessionStoreInterceptor" />
        <interceptor name="validation"
class="AnnotationValidatorInterceptor" />
        <interceptor name="workflow" class="DefaultWorkflowInterceptor" />

    . . .

    <interceptor-stack name="defaultStack">           | #3
        <interceptor-ref name="exception" />         | #4
        <interceptor-ref name="alias" />
        <interceptor-ref name="servlet-config" />
        <interceptor-ref name="prepare" />
        <interceptor-ref name="i18n" />
        <interceptor-ref name="chain" />
        <interceptor-ref name="debugging" />
        <interceptor-ref name="profiling" />
        <interceptor-ref name="scoped-model-driven" />
        <interceptor-ref name="model-driven" />
        <interceptor-ref name="fileUpload" />
        <interceptor-ref name="checkbox" />
        <interceptor-ref name="static-params" />
        <interceptor-ref name="params" />
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

        <param name="excl udeParams">doj o\ . . *</param> | #5
    </i nterceptor-ref>
    <i nterceptor-ref name="conversi onError"/>
    <i nterceptor-ref name="val i dati on">
        <param
name="excl udeMethods">i nput, back, cancel , browse</param>
    </i nterceptor-ref>
    <i nterceptor-ref name="workfl ow">
        <param
name="excl udeMethods">i nput, back, cancel , browse</param>
    </i nterceptor-ref>
    </i nterceptor-stack>

</i nterceptors>

    <defaul t-i nterceptor-ref name="defaul tStack"/> | #6

</package>

```

(annotation) <#1 The interceptors Element >
 (annotation) <#2 All of the interceptor Elements>
 (annotation) <#3 Declaring a Stack>
 (annotation) <#4 Interceptor References >
 (annotation) <#5 Configure Interceptor with Parameters>
 (annotation) <#6 Default Interceptor Reference for all Actions in this Package>

#1 The `interceptors` element contains all of the interceptors declarations of the package. This includes declarations of interceptors that can be used as well as definitions of stacks of interceptors.

#2 The `interceptor` elements each declare an interceptor that can be used in the package. Note, this just maps an interceptor implementation class to a logical name, such as `com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor` to the name `workflow`. (In Listing 4.2, we have snipped the package names to make the listing more readable.) Note, these declarations don't actually create an interceptor or associate that interceptor with any actions.

#3 Now we can define some stacks of interceptors and associate them with logical names. Since most actions will use the same groups of interceptors, arranged in the same sequence, it is common practice to define these re-usable stacks. The `struts-default` package declares several stacks, most importantly the `defaultStack`.

#4 The contents of the `interceptor-stack` element are a sequence of `interceptor-ref` elements. These references must all point to one of the logical names created by the `interceptor` elements. Creating your own stacks, as we will see when we build a custom interceptor later in this chapter, is just as easy.

#5 The `interceptor-ref` elements can pass in parameters to configure the instance of the interceptor that is created by the reference.

#6 Finally, a package can declare a default set of interceptors. This set will be associated with all actions in the package that don't explicitly declare their own interceptors. The `default-interceptor-ref` element simply points to a logical name, in this case the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

`defaultStack`. This important line is what allows our actions to inherit default set of interceptors when we extend the `struts-default` package.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

While this example is from the `struts-default` package, you can do the same thing yourself, in your own packages, when you need to change the interceptors that fire for your actions. This can be dangerous for the uninitiated. Since most of the framework's core functionality exists in the default stack of interceptors defined in `struts-default`, you probably won't want to mess with those for a while. However, we will show how to safely modify this stack when we build our custom authentication interceptor in a few pages.

XML Document Structure

Before moving on to show how you specify the interceptors that will fire for your specific actions, we should make a point about the sequence of elements within the XML documents we use for declarative architecture. These XML documents must conform to certain rules of ordering. An XML document is structured data after all. For instance, each package element contains precisely one interceptors element and that element must come in a specific position in the document. We include the complete DTD, `struts-2.0.dtd`, for your convenience in Appendix E. For now, note the following snippet from the DTD which pertains to the structure of Listing 4.2:

```
<!ELEMENT struts (package|include|bean|constant)*>

<!ELEMENT package (result-types?, interceptors?, default-interceptor-
ref?, default-action-ref?, global-results?, global-exception-mappings?,
action*)>
```

The first element definition specified the contents of the `struts` element. The `struts` element is the root element of a XML file used for the declarative architecture. As you can see in Listing 4.2, the `struts-default.xml` document certainly starts with the `struts` element. Moving on, this root element can contain zero or more instances each of four different element types. For now, we are only concerned with the `package` element. The contents of a `package` element, unlike the `struts` element, must follow a specific sequence. Furthermore, all of the elements contained in a `package` element, except for the actions, can occur only once. From this snippet, we glean the important information that our `interceptors` element must occur just once, or not at all, and must come after the `result-types` element and before the `default-interceptor-ref` element. The documents in the Struts 2 portfolio application will demonstrate the correct ordering of elements, but if you ever have questions, consult the DTD.

4.4.2 Mapping interceptors to actions

Much of the time, your actions will belong to packages that extend `struts-default`, and you will be content to let them use the `defaultStack` of interceptors they inherit from that package. Eventually, you will probably want to modify, change, or perhaps just augment that default set of interceptors. To do this you have know how to map interceptors to your actions. The association of an interceptor to an action is done with an `interceptor-ref` element. The following code snippet shows how to associate a set of interceptors with a specific action.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```
<action name="MyAction" class="org.actions.myactions.MyAction">
  <interceptor-ref name="timer"/>
  <interceptor-ref name="logger"/>
  <result>Success.jsp</result>
</action>
```

This snippet associates two interceptors with the action. They will fire in the order they are listed. Of course, you already know enough about how Struts 2 applications work to know that this action, with just the `timer` and `logger` interceptors, would not be able to accomplish much. It would not have access to any request data because the `params` interceptor is not there. Even if it could get the data from the request, it wouldn't have any validation. In reality, most of the functionality of the framework is provided by interceptors. You could define the whole set of them here, but that would be tedious, especially as you would end up repeating the same definitions across most of your actions.

Stacks address this very situation. As it turns out, you can combine references to stacks and individual interceptors. The following snippet shows a revision of the previous action element that still makes use of the `defaultStack` while adding the other two interceptors it needs.

```
<action name="MyAction" class="org.actions.myactions.MyAction">
  <interceptor-ref name="timer"/>
  <interceptor-ref name="logger"/>
  <interceptor-ref name="defaultStack"/>
  <result>Success.jsp</result>
</action>
```

We should note a couple of important things here. First, this action names interceptors, not to mention the `defaultStack`, which are declared in the `struts-default` package. Because of this, it must be in a package that extends `struts-default`. Next, while actions that don't define any `interceptor-ref`'s themselves will inherit the default interceptors, as soon as an action declares its own interceptors, it loses that automatic default and must explicitly name the `defaultStack` in order to use it.

As we have seen, if an action doesn't declare its own interceptors, it inherits the default interceptor reference of the package. The following snippet shows the line from `struts-default.xml` that declares the default interceptor reference for the `struts-default` package.

```
<default-interceptor-ref name="defaultStack"/>
```

When you create your own packages, you can make default references for those packages. We will do just this when we create the authentication interceptor in a few pages.

Now, let's see how to pass parameters into interceptors that permit such modifications of their behavior.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.4.3 Setting and overriding parameters

Many interceptors can be parameterized. If an interceptor accepts parameters, the `interceptor-ref` element is the place to pass them in. We can see that the workflow interceptor in the `defaultStack` is parameterized to ignore requests to certain action method names, as specified in the `excludeMethods` parameter element.

```
<interceptor-ref name="workflow">
  <param name="excludeMethods">input, back, cancel, browse</param>
</interceptor-ref>
```

Passing parameters into interceptors is as simple as this. With the method above, you pass the parameters in when you create the `interceptor-ref`. This one is a part of a stack. What if we wanted to reuse the `defaultStack` from which this reference is taken, but we wanted to change the values of the `excludeMethods` parameter. This is easy enough also, as demonstrated in the following snippet.

```
<action name="YourAction" class="org.actions.youractions.YourAction">
  <interceptor-ref name="defaultStack">
    <param name="workflow.excludeMethods">doSomething</param>
  </interceptor-ref>
  <result>Success.jsp</result>
</action>
```

First, we assume that this action belongs to a package that inherits the `defaultStack`. This action names the `defaultStack` as its interceptor reference but overrides the workflow interceptor's `excludeMethods` interceptor. This allows you to reuse existing stacks while still being able to customize the parameters. Very convenient.

Next up, rolling your own authentication interceptor!

4.5 Building your own interceptor

We have said several times that you probably won't need to build your own interceptor. On the other hand, we hope that we have sold the power of interceptors well enough to make you itch to start rolling your own. Apart from the care needed when sequencing the stack, and learning to account for this sequencing in your debugging, interceptors can be quite simple to write. We round out the chapter by creating an authentication interceptor what we can use to provide application based security for our Struts 2 Portfolio application. We'll start by looking at the technical details of implementing an interceptor.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

4.5.1 Implementing the *Interceptor* interface

When you write an interceptor you will implement the `com.opensymphony.xwork2.interceptor.Interceptor` interface.

```
public interface Interceptor extends Serializable {
    void destroy();
    void init();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

As you can see, this simple interceptor defines only three methods. The first two methods are typical lifecycle methods giving you a chance to initialize and clean up resources as necessary. The real business occurs in the `intercept` method. As we have already seen, this method is called by the recursive `ActionInvocation.invoke()` method. If you don't recall the details you might want to re-read Section 4.2, which describes this interceptor execution process in detail.

We will directly implement the `Interceptor` interface when we write our authentication interceptor. Sometimes you can take advantage of a convenience class provided with the distribution that provides support for method filtering. We saw parameter-based method filtering when we looked at the workflow interceptor. Such interceptors accept a parameter that defines methods for which the interceptor will not fire. This type of parameterized behavior is so common that an abstract implementation of the `Interceptor` interface has already taken care of the functionality involved in such method filtering. If you want to write an interceptor that has this type of parameterization, you can extend the `com.opensymphony.xwork2.interceptor.MethodFilterInterceptor` rather than directly implementing the `Interceptor` interface. Since our authentication interceptor doesn't need to filter methods, we'll stick to the direct implementation.

4.5.1 Building the *AuthenticationInterceptor*

The authentication interceptor will be quite simple. If you recall the three phases of interceptor processing – preprocessing, calling `ActionInvocation.invoke()`, post-processing – you can anticipate how our `AuthenticationInterceptor` will function. When a request comes to one of our secure actions, we will want to check whether the request is coming from an authenticated user. This check is made during preprocessing. If the user has been authenticated, then the interceptor will call `invoke()` and, thus, allow the action invocation to proceed. If the user has not been authenticated, the interceptor will return a control string itself, thus barring further execution. The control string will route the user to the login page.

You can see this in action by visiting the Chapter Four version of the Struts 2 Portfolio application. On the home page, there is a link to add an image without having logged in. The add image action is a secure action. Try hitting the link without having logged in. You will be automatically taken to the login page. If you want to login and try the same link again. The application comes with a default user, username = “Arty” and password = “password”. This time you are allowed to access the secure add image action. This is done by a custom interceptor that we have placed in front of all of our secure actions. Let's see how it works.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

First, we should clear up some roles. The `AuthenticationInterceptor` does not do the authentication, it just bars access to secure actions by unauthenticated users. Authentication itself is done by the login action. The login action checks to see if the username and password are valid. If they are valid, the user object is stored in a session scoped map. When the `AuthenticationInterceptor` fires, it checks to see if the user object is present in the session. If it is, it lets the action fire as usual. If it isn't, it diverts workflow by forwarding to the login page.

We should take a quick look at the `manning.chapterFour.Login` action on our way to inspecting the `AuthenticationInterceptor`. Listing 4.3 shows the execute code from the `Login` action. Note, we have trimmed extraneous code, such as validation and JavaBeans properties, from the listing.

Listing 4.3 The Login Action Authenticates the User and Stores the User in Session Scope

```
public class Login extends ActionSupport implements SessionAware {    |
#1

    public String execute() {

        User user = getPortfolioService().authenticateUser( getUsername(),
|#2
            getPassword() );
|#2

        if ( user == null )
|#3
        {
|#3
            return INPUT;
|#3
        }
|#3
        else{
|#4
            session.put( Struts2PortfolioConstants.USER, user );
|#4
        }
|#4

        return SUCCESS;
    }

    . . .

    public void setSession(Map session) {
|#5
        this.session = session;
    }
}
```

(annotation) <#1 Implement SessionAware to Receive the Session Map>

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

(annotation)<#2 Validate Username and Password>
(annotation) <#3 If Invalid, Return Input>
(annotation)<#4 If Valid, Store User in Session>
(annotation)<#4 Setter for Session Map Injection>

The first thing of interest is that our Login action uses the SessionAware interface to have the session scope map conveniently injected. This is one of the services provided by the ServletConfigInterceptor provided in the defaultStack. (See the section on that interceptor earlier in this chapter to find out all the other objects you can have injected through similar interfaces.) If the user is authenticated, we will store the user object in this session map. As far as the authentication itself goes, we simply hand the username and password, have been transferred and validated by the framework through the methods outlined in the previous chapters, to our service method. If the credentials are valid, we put the user object into the session map under a known constant key and return a successful control string. If the credentials are bogus, we return to the input form.

With the Login action in place, we can look at how the AuthenticationInterceptor protects secure actions from unauthenticated access. Listing 4.4 shows the full code.

Listing 4.4 Inspecting the Heart of the AuthenticationInterceptor

```
public class AuthenticationInterceptor implements Interceptor {           |#1

    public void destroy() {                                             |#2
    }                                                                    |#2
                                                                    |#2
    public void init() {                                               |#2
    }                                                                    |#2

    public String intercept( ActionInvocation actionInvocation ) throws
Exception{ |#3

        Map session =
actionInvocation.getInvocationContext().getSession(); |#4
        User user = (User) session.get( Struts2PortfolioConstants.USER );
| #4

        if (user == null) {
| #5
            return Action.LOGIN;
| #5
        }
| #5
        else {
| #6

| #6
            Action action = ( Action ) actionInvocation.getAction();
| #6

| #6
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

        if (action instanceof UserAware) {
| #6            ((UserAware) action).setUser(user);
| #6        }
| #6
        return actionInvocation.invoke();
| #7    }
    }
}

```

(annotation) <#1 Implements Interceptor>

(annotation) <#2 Empty Implementations of Life Cycle Methods >

(annotation) <#3 Intercept Method: Entry Point for Interceptor>

(annotation) <#5 Forward Unauthenticated Users to Login Page>

(annotation) <#7 Recursive Call to Continue Action Invocation Process>

#4 The main part of the interceptor starts inside the `intercept()` method. Here we can see that the interceptor uses the `ActionInvocation` object to obtain information pertaining to the request. In this case, we are getting the session map. With the session map in hand, we retrieve the user object stored under the known key.

#5 If the user object is null, then the user hasn't been authenticated through the login action. At this point we return a result string, without allowing the action to continue. This result string will return the user to the result page. If you consult the `chapterFour.xml` file, you will see that the `chapterFourSecure` package defines the login result as a global result, available to all actions in the secure package. In Chapter 9 Results in Detail, we will learn about configuring global results.

Insider Tip: If you consult the API, you will see that the `getInvocationContext()` method returns the `ActionContext` object associated with the request. As we learned earlier, the `ActionContext` contains many important data objects for the processing of the request, including the `ValueStack` and key objects from the Servlet API such as the session map that we are using here. If you recall, we can also access objects in this `ActionContext` from our view layer pages, i.e. JSP's, via OGNL expressions. In this interceptors, we utilize programmatic access to those objects. Note, while it's always possible to get your hands on the `ThreadLocal ActionContext`, it's not a good idea. We recommend confining programmatic access to the `ActionContext` to interceptors, and using the `ActionInvocation` object as a path to that access. This keeps your API's separated and lays the foundation for clean testing.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

#6 If the user object exists, then the user has already logged in. At this point, we get a reference to the current action from the `ActionInvocation` and check to see whether it implements the `UserAware` interface. We have created this interface to allow actions to have the user object automatically injected into a setter method. This technique, which we obviously copied from the framework's own interface based injection, is a very powerful way of making your action's cleaner and more efficient. Most secure actions will want to work with the user object. With this interceptor in the stack, they just need to implement the `UserAware` interface to have the user conveniently injected. You can check out any of the secure actions in the Struts 2 Portfolio's `chapterFour` package to see how they do this. With the business of authentication out of the way, the interceptor calls `invoke()` on the `ActionInvocation` object to pass control on to the rest of the interceptors and the action. And that's that. Pretty straight forward really.

We need to point out one important detail before moving on. Interceptor instances are shared among actions. While a new instance of an action is created for each request, interceptors are reused. This has one important implication. Interceptors are stateless. Don't try to store data related to the request being processed on the interceptor object. This is not the role of the interceptor anyway. The interceptor should just apply its processing logic to the data of the request, which is already conveniently stored in the various objects you can access through the `ActionInvocation`.

Now we need to look at how this interceptor can be applied to our secure actions. Since we put all of our secure actions into a single package, we can build a custom stack that includes our `AuthenticationInterceptor`, and then declare that as the default interceptor reference for the secure package. Turns out, that's exactly what we've done. Listing 4.5 shows the elements from `chapterFour.xml` that configure the `chapterFourSecure` package.

Listing 4.5 Declaring our `AuthenticationInterceptor` and Building a New Default Stack

```
<package name="chapterFourSecure" namespace="/chapterFour/secure"
        extends="struts-default">

    <interceptors>                                     | #1
        <interceptor name="authenticationInterceptor"   | #2
            class="manning.utils.AuthenticationInterceptor"/> | #2

        <interceptor-stack name="secureStack">         | #3

            <interceptor-ref name="authenticationInterceptor"/>
            <interceptor-ref name="defaultStack"/>
        </interceptor-stack>

    </interceptors>

    <default-interceptor-ref name="secureStack"/>      | #4

    . . .

</package>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

(annotation) <#1 interceptors Element for our Package>
(annotation) <#2 Map our Implementation Class to a Logical Name >
(annotation) <#3 Build a New Stack that Adds the AuthenticationInterceptor to the defaultStack>
(annotation) <#4 Make Our New secureStack the Default for the Package>

With all of our secure actions bundled in this package, we just need to make a stack that includes our `AuthenticationInterceptor`, and then declare it as the default one. You can see how easy this is. First, we have to map our Java class to a logical name. We've chose `authenticationInterceptor` as our name. Seems good enough. Next, we build a new stack that just takes the `defaultStack` and adds our new interceptor to the top of it. Note, we put it on top because we might as well stop an unauthenticated request as soon as possible. Finally, we declare our new `secureStack` as the default stack. Every action in this package will now have authentication with automatic routing back to the login page and injection of the user object for any action that implements the `UserAware` interface. It feels like we've accomplished something, no? And the best part is that our interceptor is completely separate from our action code and completely re-usable.

4.6 Summary

In this chapter we have met, perhaps, the most important component of the framework. Even though you can get away without developing interceptors for quite a while, a solid understanding of these important components is critical to understanding the framework in general. A solid grasp of interceptors will facilitate easier debugging and give you an all round sense of well being while working with the framework. We hope we have given you a solid leg up on the road to interceptor mastery.

By now, you should pretty much have come to grips with the role of the interceptor in the framework. To reiterate, the interceptor component provides a nice place to separate the logic of various crosscutting concerns into layered, reusable pieces. Tasks such as logging, exception handling, and dependency injections can all be encapsulated in interceptors. With the functionality of these common tasks thus modularized, we can easily use the declarative architecture to customize stacks of interceptors to meet the needs of our specific actions or packages of actions.

Perhaps the toughest thing to wrap your mind around, as far as interceptors go, is the recursive nature of their execution. Central to the entire execution model of the Struts 2 framework is the `ActionInvocation`. We learned how the `ActionInvocation` contains all the important data for the processing of the request including everything from the action and its interceptors to the `ActionContext`. On top of this, it actually manages the execution process. As we have seen, it exposes a single, recursive `invoke()` method as an entry point into the execution process. `ActionInvocation` keeps track of the state of the execution process and invokes the next interceptor in the stack each time `invoke()` is called until, finally, the action is executed.

Interceptors themselves are invoked via their `intercept()` method. The execution of an interceptor can be broken into three phases: preprocessing, passing control on to the rest of the action invocation by calling `invoke()`, and post-processing. Interceptors can also divert workflow by returning a control string themselves instead of calling `invoke()`. They also

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

have access to all key data via the `ActionInvocation` instance they receive. Ultimately, interceptors can do about anything.

We also reviewed the functionality of many of the built-in interceptors that come with the `struts-default` package. Familiarity with these is critical to saving yourself from repeating work already done for you. We highly recommend staying up to date on the current set of interceptors available from the Struts 2 folks. They may have already built something you need by the time this book makes it onto your shelf. A quick visit to the Struts 2 web site is always a good idea. Finally, we hope that our `AuthenticationInterceptor` has convinced you that its pretty easy to write your own interceptors. Again, we think the hardest part is understanding how interceptors work. Writing them is not so bad. We're confident that you will soon find yourself with your own ideas for custom interceptors.

Now that we have covered actions and interceptors, we should be ready to move on to the view layer and start exploring the rich options that the framework offers for rendering result pages. Before we do that, we have one more stop on our tour of the core components of framework. And most likely, its a stop you've been wondering about. Next up, Chapter 5 Data Transfer: OGNL and Type Conversion will work on dispelling that mysterious OGNL cloud.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5 Data Transfer: OGNL and Type conversion

Now that we've covered the action and interceptor components, you've got a pretty good idea of the how the major chunks of server side logic execute in the Struts 2 framework. We've conveniently avoided poking our noses into the details of two of the more important tasks that the framework helps us achieve: data transfer and type conversion. Actually, we've been able to avoid thinking about these important tasks because the framework automates them so well. This will continue to be true for large portions of your development practice. However, if we give a small portion of our energy to learning how the data transfer and type conversion actually works, we can squeeze a whole lot more power out of the framework's automation of these crucial tasks.

We've already seen part of the data transfer and type conversion in action. For instance, when we learned about actions, we saw that the framework automates the transfer of data from request parameters to properties exposed on the action. We didn't say much about it at the time, but the framework also automatically converts the data from the string representations found in request parameters to the strict types of the Java language. The framework comes with a very strong set of built-in type converters that support all common conversions, including complex types such as `Maps` and `Lists`. The central focus of this chapter will be explaining how to wire up your applications so that you can take advantage of the framework's ability to automatically move data from the incoming requests onto your Java side properties. At the end of the chapter, we will also show you how to extend the type conversion mechanism by developing custom converters.

This chapter also starts a two part formal coverage of OGNL. The type converters are actually a part of OGNL. Accordingly, this chapter will cover OGNL from the point of view of data transfer and type conversion. In the next chapter, which introduces the Struts 2 Tags, we will focus on the expression language of OGNL. This division of labor is based upon our treatment of type conversion and tags as the primary framework components; OGNL is the stuff that makes them all work. Thus, we think it makes sense to divide the coverage of OGNL between these chapters.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5.1 Data transfer and type conversion: common tasks of the web application domain

Earlier in this book we said that one of the common tasks of the web application domain was the movement and conversion of data from the string based form of HTTP to the various data types of the Java language. If you have worked with web applications for very many years, you will certainly be familiar with the tedious task of moving data from form beans to data beans. This boring task is complicated by the accompanying task of converting from strings to Java types. Parsing strings into doubles and floats, catching the exceptions that arise from bad data . . . no fun at all. Worse yet, these tasks amount to pure infrastructure. All you are doing is preparing for the real work.

Data transfer and type conversion actually happen on both ends of the request processing cycle. We've already seen that the framework moves the data from the string based HTTP request onto our JavaBeans properties, which are clearly Java types. Moreover, the same thing happens on the other end. When the result is rendered, we typically funnel some of the data from those JavaBeans properties back out into the resulting HTML page. Again, while we haven't given it much thought, this means that the data has been reconverted from the Java type back out to a string format.

This process is something that occurs with nearly every request in a web application. It's just an inherent part of the domain. No one will moan about handing this responsibility over to the framework. Nonetheless, there will be times when you want to extend or configure this automated support. The Struts 2 type conversion mechanisms are very powerful, and quite easily extended. We think you'll be quite excited when you see the possibilities for writing your own custom converters. But first, we need see who is actually responsible for all of this automated wizardry.

5.2 OGNL and Struts 2

We call it wizardry, but, as we all know, computers are rational machines. No magic involved whatsoever. Perhaps unsolved mystery is a more well turned phrase. What exactly are these unsolved mysteries then? To be specific, we don't yet know how exactly all of that data makes it from the HTML request to the Java language and back out to HTML through the JSP tags? The next section will clarify this mysterious process.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5.2.1 What OGNL does

Actually, it's not mysterious at all. In fact, OGNL is quite ordinary. It stands for the Object Graph Notation Language. Sounds perfectly harmless, right? No? Actually, I agree. It sounds horrifying, as if I should have studied harder in school. Apparently in an attempt to lessen the academic aura, the makers of OGNL suggest one of those cool acronym pronunciations. They suggest that it should be pronounced similarly to that last few syllables of 'orthogonal' as slurred by a drunken orator. Two impressions. First, I'd love to be at that party . . . NOT! Second, what does orthogonal mean?

But all joking aside, OGNL is a powerful glue technology that has been integrated into the Struts 2 framework to provide data transfer and type conversion. OGNL is the glue between the framework's string based HTTP input and output, and the Java based internal processing. Nothing more really. However it is quite powerful and, while it seems that you can use the framework without really knowing about OGNL, your development efforts will be made many times more efficient by spending a few moments of quality time with this oddly named power utility.

From the point of view of a developer building applications on the Struts 2 framework, OGNL pretty much consists of two things, an expression language and type converters.

Expression language

First, let's look at the expression language. We have been using OGNL's expression language in our form input field names and in our JSP tags. In both places, we've been using OGNL expressions to bind Java side data properties to strings in the text based view layers, commonly found in the name attributes of form input fields, or in various attributes of the Struts 2 tags. The simplicity of the expression language, in its common usage, makes for a ridiculously low angle learning curve. This has allowed us to get pretty deep into Struts 2 without specifically covering it. Let's review what we've already been doing.

The following code snippet, from our Struts 2 Portfolio application's `RegistrationSuccess.jsp`, shows a Struts 2 tag using the OGNL expression language.

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

The OGNL expression language is the bit inside the double quotes of the value attribute. This Struts 2 property tag takes a value from a property on one of our Java objects and writes it into the HTML in place of the tag. Super simple. This is the point of expression languages. Expression languages allow us to use a simplified syntax to reference objects that reside in the Java environment. The OGNL expression language can be much more complex than this single element expression, it even supports such advanced features as invoking method calls on the Java objects that it can access, but the whole idea of an expression language is to simplify access to data.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Heads-up: The integration of OGNL into the Struts 2 framework is tight. Pains have been taken to make the simplest uses cases just that . . . simple. With this in mind, many instances of OGNL expressions require no special escaping. While there is an OGNL escape sequence, `%{expression}`, that signals to the framework when to process the expression as an expression, rather than interpreting as a string literal, this is not often used. In the name of intelligent defaults, Struts 2 will automatically evaluate the string as an OGNL expression in all contexts that warrant such a default behaviour. In contexts where strings are mostly likely going to be strings, the framework will require the OGNL escape sequence. As we move along, we will specifically indicate which context is which.

The other side of the coin. While the property tag, which resides in a result page, reaches back into the Java environment to pull a value from the `portfolioName` property, we've also seen that OGNL expressions are used in HTML forms to target properties in the Java environment as destinations for the data transfer. In both cases, the role of the OGNL expression is to provide a simple syntax for targeting specific properties in groups of objects. In other words, OGNL expressions navigate object graphs to find the appropriate property. While tags then pull data from those properties, the params interceptor moves data from the request parameters into those properties. This should be clear by now. But how does the data get converted from the string to the native Java type of the destination property?

Type converters

In addition to the expression language, we've also been using OGNL type converters. Even in this simple case of the Struts 2 property tag, a conversion must be made from the Java type of the property referenced by the OGNL expression language to the string format of the HTML output. Of course, in the case of the `portfolioName`, the Java type is also a string. But this just means that the conversion is easy. Every time data moves to or from the Java environment, a translation must occur between the string version of that data that resides in the HTML and the appropriate Java data type. Thus far, we have been using simple data types for which the Struts 2 framework provides adequate built-in OGNL type converters. In fact, the framework provides built-in converters to handle much more than we have been asking of it. Shortly, we'll cover the built-in type converters and show you how to map your incoming form fields onto a wide variety of Java data types including all the primitives as well as a variety of collections. But first, let's take an architectural look at where OGNL fits in to the framework, just to be clear about things.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5.2.2 How OGNL fits into the framework

Figure 5.1 shows how OGNL has been incorporated into the Struts 2 framework.

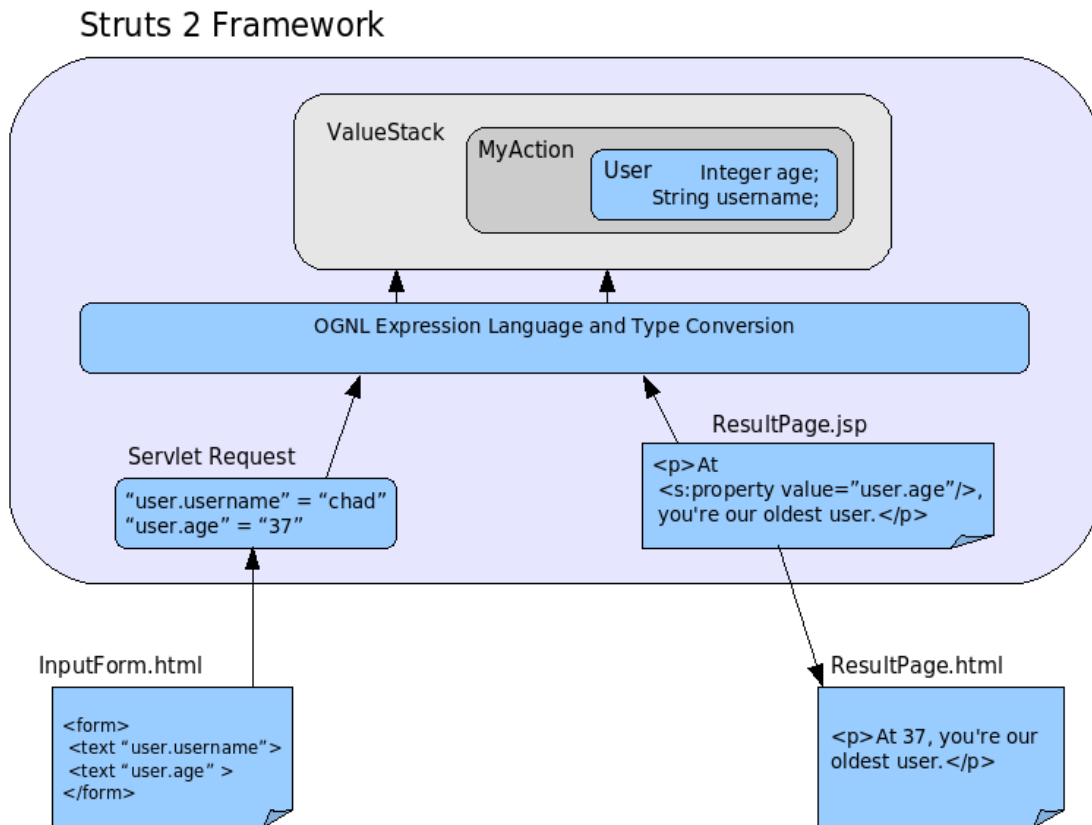


Figure 5.1 OGNL Provides the Framework's Mechanism for Transferring and Type Converting Data

Figure 5.1 shows the entire path of data into and out of the framework. Everything starts with the HTML form in the `InputForm.html` page from which the user will submit a request. And everything ends with the response that comes back to the user, represented in Figure 5.1 as the `ResultPage.html`. Now, let's follow the data in and out of the framework.

Data in

We'll start with the `InputForm.html` shown in Figure 5.1. In this case, the form contains two text input fields. Note, in the interest of space, we have created a pseudo HTML markup for these fields; this certainly won't validate. The strings in the pseudo text input tags are the name attributes of the fields. Again, it's important to realize that these names are valid OGNL expressions. All that we need now is a user to enter two values for the fields and submit the form to the framework.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

When the request enters the framework, as we can see in Figure 5.1, it is exposed to the Java language as an `HttpServletRequest` object. As we learned earlier, Struts 2 is built on the Servlet API. The request parameters are stored as name and value pairs, and both name and value are `Strings`. As you can see in Figure 5.1, the request object just has a couple of name – value pairs where the names are the names of our form's text fields and the values are the values entered by the user when the form was submitted. Everything is still a string. This is where the framework picks up the ball.

We know the framework is going to handle the transfer and type conversion of the from these request parameters for us. First, let's look at where the data is going. In the chapter on actions, we saw that the framework can automatically transfer parameters onto our action object or a model object. Furthermore, we have learned that these objects are actually placed onto something called the `ValueStack`, and this `ValueStack` is actually the target of the data transfer. In Figure 5.1, we can see that our action object has been placed on the `ValueStack`. In the case represented by this figure, we are exposing our `User` object as a JavaBeans property on our action object. With our action object on the `ValueStack`, we are ready for the data transfer and conversion.

From our study of interceptors, we know that the `params` interceptor does the work of transferring this data from the request object to the `ValueStack`. The tricky part of the job is mapping the name of the parameter onto an actual property on the `ValueStack`. This is where `OGNL` comes in. The `params` interceptor interprets the parameter name as an `OGNL` expression to locate the correct destination property for the value. `OGNL` expressions are always interpreted against a specific object that serves as the entry point into resolving the property chain of the expression. In the Struts 2 environment, the default object against which `OGNL` resolves its expressions is a sort of virtual object created by the `ValueStack`.

Definition: The `ValueStack` is a Struts 2 construct that combines the properties of a stack of objects into a single virtual object exposing the combined property set of the stack. If duplicate properties exist, i.e. two objects in the stack both have a `name` property, then the property of the highest object in the stack will be the one exposed on the virtual object represented by the `ValueStack`. The `ValueStack` represents the data model exposed to the current request and is the default object against which all `OGNL` expressions are resolved.

The `ValueStack` is a virtual object? Sounds complicated but its not. The `ValueStack` holds a stack of objects. These objects all have properties. The magic of the `ValueStack` is that all of the properties of these objects appear as properties of the `ValueStack` itself. In our case, since the action object is on the `ValueStack`, all of its properties appear as properties of the `ValueStack`. The tricky part comes when more than one object is placed on the `ValueStack`. When this happens, we can have a contention of sorts between properties of those two objects. Let's say that two objects on the stack both have a `username` property. How does this get resolved? Simply. The `username` exposed by the `ValueStack` will always be that of the highest object in the stack. The properties of the higher objects in the stack cover up similarly named properties of objects lower in the stack. We'll cover this in more detail when we discuss the `OGNL` expression language in Chapter Six, Building a view: tags.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

For now, this should be enough to see how the request parameters find their way to their correct homes. In Figure 5.1, one of the request parameters is named `user.age`. If we resolve this as an OGNL expression against the `ValueStack`, we first ask, Does the `ValueStack` have `user` property? Since, as we have just learned, the `ValueStack` exposes the properties of the objects it contains, we know that, indeed, the `ValueStack` does have a `user` property. Next, does this `user` property have a `age` property? Of course it does. Obviously, we've found the right property. Now what?

Once the OGNL expression has been used to locate the destination property, the data can be moved on to that property by calling the property's setter with the correct value. But, at this point, the original value is still the string `'37'`. Here's where the OGNL type converters come into play. We need to convert the `String` to the Java type of the `age` property targeted by the OGNL expression, which is an `int`. OGNL will consult its set of available type converters to see if any of them can handle this particular conversion. Luckily, the Struts 2 framework provides a well outfitted set of type converters to handle all the normal conversions of the web application domain. Conversion between strings and integers is certainly provided for by the built-in type converters. The value is converted and set on the `user` object, just where we'll find it when we start our action logic after the rest of the interceptors have fired.

Data out

Now for the other half of the story. Actually, it's the same story, but in reverse. After the action has done its business, calling business logic, doing data operations, and so forth, we know that eventually a result will fire that will render a new view of the application to the user. Importantly, during the processing of the request, our data objects will remain on the `ValueStack`. The `ValueStack` acts a kind of place holder for viewing the data model throughout the various regions of the framework.

When the result starts its rendering process, it will also have access to the `ValueStack`, via the OGNL expression language in its tags. These tags will retrieve data from the `ValueStack` by referencing specific values with OGNL expressions. In Figure 5.1, the result is rendered by `ResultPage.jsp`. In this page, the age of the user is retrieved with the Struts 2 property tag, a tag that takes an OGNL expression to guide it to the value it should render. But once again, we must convert the value; this time we convert from the Java type of the property on the `ValueStack` to a string which can be written into the HTML page. In this case, the `Integer` object is converted back into a string for the rather depressing message that “At 37, you're our oldest user.” Must be a social networking site.

Now you know what OGNL does. This chapter is going to focus on the framework's data transfer and type conversion. We will give you just enough expression language to map your form field names to your Java properties. Moreover, we'll focus on the movement of data into the framework while learning the specifics of type conversion. With that said, you can save any expression language questions you have for Chapter Six, which introduces the Struts 2 tag libraries.

If you need a break before we go on to cover the built-in type converters, you could practice saying OGNL.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5.3 Built-in type converters

Now that we've seen how OGNL has been integrated into the framework to provide automatic data transfer and type conversion, it's time to see the nuts and bolts of working with the built-in type converters. Out of the box, the Struts 2 framework can handle almost any type conversions that you will require. These conversions are done by the built-in OGNL type converters that come with the framework. In this section, we'll show you what the framework will handle, give you plenty of examples, and show you how to write OGNL expressions for form field names that need to locate more complexly structured properties such as those backed by arrays, Maps and Lists.

5.3.1 Out of the box conversions

The Struts 2 framework comes with built-in support for converting between the HTTP native strings and the following list of Java types.

- `String` – sometimes a string is just a string.
- `boolean` / `Boolean` – true and false strings can be converted to both primitive and object versions of boolean
- `char` / `Character` – primitive or object
- `int` / `Integer`, `float` / `Float`, `long` / `Long`, `double` / `Double` – primitives or objects
- `Date` – string version will be in `SHORT` format of current `Locale` (i.e., “12/10/97”)
- Arrays – but each string element must be convertible to the array's type
- `Lists` – populated with `Strings` by default
- `Maps` – populated with `Strings` by default

When the framework locates the Java property targeted by a given OGNL expression, it will look for a converter for that type. If that type is listed above, you don't need to do anything but sit back and receive the data.

In order to utilize the built-in type conversion, you just need to build an OGNL expression that targets a property on the `ValueStack`. The framework provides built in support for automatic transfer and conversion to properties of all the types in the bullet list above. The OGNL expression will either be the name of your form field, under which the parameter will sent in the HTTP request, or it will be somewhere in your view layer tags, such as one of the Struts 2 JSP tags. Again, our current discussion will focus on the data's entry into the framework as request parameters. Chapter Six will focus on the tag point of view. However, this is mostly a matter of convenience; OGNL servers the same functional roles at both ends of the request processing – its expression language navigates our object graph to locate the specified property and its type converters manage the data type translations between the string based HTTP world and the strictly typed Java world. Data in, data out? It doesn't matter. The type conversion and OGNL will be the same.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

5.3.2 Mapping Form Field Names to Properties with OGNL Expressions

Hooking up your Java properties to your form field names, in order to facilitate the automatic transfer and conversion of request parameters, is a two step process. First, you need to write the OGNL expressions for the name attributes of your form fields. Second, you need to create the properties that will receive the data in the Java side. We'll go through each of the built-in conversions in the order from the bullet list above, showing how to set up both sides of the equation.

Primitives and Wrapper Classes

Since the built-in conversions to Java primitives and wrapper classes, such as Boolean and Double, are quite simple, we provide a single example to demonstrate them. We won't show every primitive or wrapper type because they all work the same way. First, let's see the OGNL expressions in the form fields. Listing 5.1 shows the Chapter Five version of our Struts 2 Portfolio's registration form, from `Registration.jsp`.

Listing 5.1 HTML Form Input Field Names are OGNL Expressions that Target Specific Properties on the ValueStack

```
<h4>Complete and submit the form to create your own portfolio.</h4>
<s:form action="Register">
  <s:textfield name="user.username" label="Username"/>
  <s:password name="user.password" label="Password"/>
  <s:textfield name="user.portfolioName" label="Enter a name for your
portfolio"/>
  <s:textfield name="user.age" label="Enter your age, with double
precision!"/>
  <s:textfield name="user.birthday" label="Enter your birthday.
(mm/dd/yy)"/>
  <s:submit/>
</s:form>
```

This is nothing new. But now that you know that each of the input field names is actually an OGNL expression, you will see a lot deeper into this apparently simple form markup. Recall, that our OGNL expressions resolve against the ValueStack, and that our action object will be automatically placed there when request processing starts. In this case our Register action uses a JavaBeans property, `user`, backed directly with our User domain object. The following snippet, from our Chapter Five version of `Register.java` shows the JavaBeans property that exposes our User object.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

private User user;

public User getUser() {
    return user;
}

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

```

If you want, you can see the full source for `Register.java` by looking at the sample application. But really, it's nothing new. The only thing important to our current discussion is the exposure of the user object as a JavaBeans property. Since the type of this property is our `User` class, let's look at that class to see what properties it exposes. Listing 5.2 shows the full listing of the `User` bean.

Listing 5.3 The JavaBeans Properties Targeted by the OGNL Expressions in Listing 5.2

```

public class User {

    private String username;
    private String password;
    private String portfolioName;
    private Double age;
    private Date birthday;

    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getPortfolioName() {
        return portfolioName;
    }
    public void setPortfolioName(String portfolioName) {
        this.portfolioName = portfolioName;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Double getAge() {
        return age;
    }
    public void setAge(Double age) {

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>


```

        this.age = age;
    }
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}

```

Go ahead and test the registration out if you like. Hit the Create an Account link from the Chapter Five version of the Struts 2 Portfolio application. The process is simple. The request comes into the framework with a map of name – value pairs that associate the name from the form input field with the string value entered. The name is an OGNL expression. This expression is used to locate the target property on the ValueStack. In the case of the Register action, the action itself is on top of the stack and an OGNL expression such as `user.birthday` finds the user property on the action, then finds the birthday property on that user object. In Java, this becomes the following snippet.

```

getUser().getBirthday();

```

OGNL sees that the birthday property is of Java type `Date`. It then locates the string to `Date` converter, converts the value and sets it on the property. All of the simple object types and primitives are just this easy. Note, if the incoming string value doesn't represent a valid instance of the primitive or type, then a conversion exception is thrown. Note that this conversion exception occurs before and distinct from validation code. Validation code is about validating the data as valid instances of the data types from the perspective of the business logic of the action. Conversion problems are problems that occur when just trying to bind the HTTP strings values to their Java types. In Chapter 13 Understanding Internationalization, we will learn how to customize the conversion exception handling and the error messages shown to the user when such conversion errors arise.

Struts 2 also provides rich support for transferring sets of data onto a variety of set data types on the Java side ranging from arrays to `Collections`. The following paragraphs show how to utilize these capabilities. Note, for each of these examples we will re-use a single action object, the `DataTransferTest`. From the perspective of data transfer and type conversion, action objects need only expose the properties that will receive the data. The `DataTransferTest` exposes all of the properties for the various examples in this chapter. We did this to consolidate the various permutations of data transfer into a convenient point of reference. Note, however, that each example is mapped in the `chapterFive.xml` as a distinct Struts 2 action, which is perfectly valid. In reality, we have a number of actions that simply use the same action class as their implementation.

These examples mean to demonstrate how to set up the data transfer and type conversion. Our action will do little more than serve as a data holder for these examples. Forms will submit request data, the framework will transfer that data to the properties exposed on the action, the action will do nothing be forward to the success result, and that result will just display the data by pulling it off of the action with Struts 2 tags. Hopefully, this will serve as a clean reference that will make all the variations on data transfer and type conversion crystal clear.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now, let's see how to have Struts 2 automatically transfer multiple values onto array backed properties.

Arrays

Struts 2 provides support for converting data to Java arrays. If you've worked with array backed properties, also known as indexed JavaBeans properties, you'll appreciate the ease with which Struts 2 handles these properties. Most of these improvements come from the OGNL expression language support for navigating to such properties. You can see the array data transfer in action by visiting the array data transfer link on the Chapter Five home page. Listing 5.4 shows the form from `ArraysDataTransferTest.jsp` that submits data targeted at Array properties.

Listing 5.4 Form that Targets Array Properties for Data Transfer: Each Input Field Name is an OGNL Expression

```
<s:form action="ArraysDataTransferTest">
  <s:textfield name="ages" label="Ages"/>           | #1
  <s:textfield name="ages" label="Ages"/>           | #1
  <s:textfield name="ages" label="Ages"/>           | #1

  <s:textfield name="names[0]" label="names"/>       | #2
  <s:textfield name="names[1]" label="names"/>       | #2
  <s:textfield name="names[2]" label="names"/>       | #2

  <s:submit/>
</s:form>
```

(annotation) <#1 These fields all target the 'ages' property>

(annotation) <#2 These fields all target the 'names' property>

On the OGNL expression side, you just have to know how to write an expression that can navigate to an array property on a Java object. The form shown in Listing 5.4 submits data to two different array properties. The first array property, named `ages`, will receive the data from the first three fields. The second array property, `names`, will receive the data from the second three fields. These properties, if the transfer is to work, must exist on the `ValueStack`. For this example, we'll expose the array properties on our action object, which will see momentarily.

This form demonstrates two syntaxes for targeting arrays with OGNL expressions. To understand what will happen, we need to refresh ourselves on the HTTP and Servlet API details that will occur as a result of this form being submitted. The first thing to remember is that the name of each input field, as far as HTTP and the Servlet API is concerned, is just a string name. These layers know nothing about OGNL. With that in mind, it's time for a pop quiz. How many request parameters will be submitted by this form? The correct answer is four. The first three fields all have the same name; this will result in a single request parameter with three values, perfectly valid in HTTP. On the other hand, the second set of fields will each come in as a

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

distinct parameter with a single value mapped to it. When this request hits the framework, four request parameters will exist as follows:

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Parameter Name	Parameter Value(s)
ages	12, 33, 102
names[0]	Chad
names[1]	Don
names[2]	Beth

Now, let's look at the implementation of the properties that will receive this data. These properties will be exposed on our action object. Listing 5.5 shows the target properties, each of an array type, from `DataTransferTest.java` source.

Listing 5.5 Array Properties Targeted by OGNL Input Field Names: They Don't Need Indexed Getters and Setters!

```
private Double[] ages ;

public Double[] getAges() {
    return ages;
}

public void setAges(Double[] ages) {
    this.ages = ages;
}

private String[] names = new String[10];

public String[] getNames() {
    return names;
}

public void setNames(String[] names) {
    this.names = names;
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

First, note that we don't need indexed getters and setters for these properties. OGNL handles all of the indexing details. We just need to expose the array itself via a getter and setter pair. Now, consider what happens when the framework transfers the 'ages' parameter. First, it resolves the property and finds the ages property on the action as seen in Listing 5.5. The value of the ages parameter in the request is an array of three age strings. Since the ages property on the action is also an array this makes the data transfer simple. OGNL creates a new array and sets it on the property. But OGNL does even more for us. In this case, the ages property is an array of element type Double. OGNL sees this and automatically runs its type conversion for each element of the array. Very nice! Also note, since the framework is creating the array for us in these cases, we don't need to initialize the array ourselves. For this example, we used multiple text input fields with the same name to submit multiple values under the ages parameter. In real applications, parameters with multiple values mapped to them are frequently the result of input fields that allow selection of multiple values, such as a select box.

Now, let's look at how the framework handles the three individual parameters with the names that look like array indexing. As far as the Servlet API is concerned, these are just three unique names. We can see that they seem to refer to a single array, and provide indexing into that single array, but the Servlet API sees only unique strings. However, when the framework hands these names to OGNL, they are accurately interpreted as references to specific elements in a specific array. These parameters, thus, get set, one at a time, into the elements of the names array. We should make a couple of comments before moving on. First, using this method requires initialization of the array. This is necessary because the OGNL expressions are targeting individual elements of an existing array; the previous method was setting the entire array so it didn't require an existing array. Second, we still don't need indexed getters and setters!

With all of this in place, the framework will automatically transfer and convert the request parameters on to our action's properties. The action, in these examples, does nothing but forward to the result page, ArraysDataTransferSuccess.jsp. The following snippet shows the code from this page.

```
<h5>Congratulations! You have transferred and converted data to and from  
  Arrays.</h5>  
<h3>Age number 3 = <s:property value="ages[2]" /> </h3>  
<h3>Name number 3 = <s:property value="names[2]" /> </h3>
```

We don't want to say too much about the Struts 2 tags now; that's the topic of the next chapter. But it should be easy enough to understand that this result page pulls some data off the action's array properties just to prove that everything is working. The rest of the examples in this chapter will follow a similar pattern of using a result page to pull the data off the action just to verify that the transfer and conversion is working. We may not show this code from the result pages every time though.

Many developers prefer to work with some of the more feature rich classes from the Java collections API. Next, we'll look at working with Lists.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Lists

In a similar fashion to array properties, Struts 2 supports automatic conversion of sets of request params into properties of various `Collection` types, such as `Lists`. Using `Lists` is almost like using arrays. The only difference being that `Lists`, prior to Java 5, don't support type specification. This typeless nature of `Lists` has an important consequence for the type conversion mechanisms of Struts 2. When the framework works with arrays, the type conversion can easily find the element type by inspecting the property itself as arrays are always typed in Java. With `Lists`, there is no way to automatically discover this.

We have two choices when working with `Lists`. Either specify the type for our elements or accept the default type. By default, the framework will convert request parameters into `Strings` and populate the `List` property with those `Strings`. Our first example will accept this default behaviour. The mechanics of using `Lists` is almost identical to using arrays. The following snippet shows the form field markup from `ListsDataTransferTest.jsp` that will target some `List` properties.

```
<s:textfield name="middleNames[0]" label="middleNames"/>           | #1
<s:textfield name="middleNames[1]" label="middleNames"/>           | #1
<s:textfield name="middleNames[2]" label="middleNames"/>           | #1

<s:textfield name="lastNames" label="lastNames"/>                   | #2
<s:textfield name="lastNames" label="lastNames"/>                   | #2
<s:textfield name="lastNames" label="lastNames"/>                   | #2

(annotation) <#1 These fields all target the 'middleNames' property>
(annotation) <#2 These fields all target the 'lastNames' property.>
```

As you can see, we once again show two different notations for referencing target properties with OGNL expressions. These are the exact same notations as used with Arrays. The only difference is in the Java side. In the Java, the `List` properties are also pretty much like the array properties except the type is different. Listing 5.6 shows the target `List` properties from the `DataTransferTest.java` source.

Listing 5.6 Using List Properties to Receive Data from the Automatic Transfer and Conversion Mechanisms of Struts 2

```
private List lastNames ;

public List getLastNames()
{
    return lastNames;
}
public void setLastNames ( List lastNames ) {
    this.lastNames=lastNames;
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```

private List middleNames ;

public List getMiddleNames()
{
    return middleNames;
}
public void setMiddleNames ( List middleNames ) {
    this.middleNames=middleNames;
}

```

These look pretty much like the array properties except the type is `List`. There are a couple of things to note. First, you don't have to pre-initialize any of the `Lists`, even the ones receive data from the indexed OGNL notation. Second, with out type specification, the elements of these `Lists` will all be `String` objects. If that works for your requirements, great. In our case, our data is first and last names so this is just fine. But if you name your field “birthdays”, don't expect the framework to convert to `Dates`. It will just make a `List` of `Strings` out of your incoming birthday strings. If you want to see this example in action, check out the List Data Transfer Test link on the Chapter Five home page. Again, the result page for this example will pull some values out of the `List` properties on the action just to prove everything is working as advertised. If you want to look at the JSP to see the tags, check out `ListsDataTransferSuccess.jsp`.

Sometimes, you'll want to specify a type for your `List` elements rather than just working with `Strings`. No problem. We just need to inform OGNL of the element type we want for a given property. This is done with a simple properties file. The OGNL type conversion uses properties files for several things. Later in the chapter, when we write our own type converters, we will see another use for these files. For now, we are just going to make a properties file that tells OGNL what type of element we want in our `List` property. In order to specify element types for properties on our action object, we create a file according to the following naming convention.

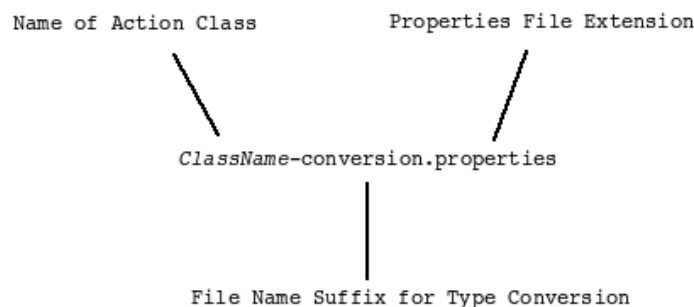


Figure 5.2 Naming convention for type conversion properties files

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

We then place this file next to the class in your Java package. In our case, we are going to create a file called `DataTransferTest-conversion.properties` and place it next to our `DataTransferTest.java` class in the `manning.chapterFive` package. If you check out the sample application, you'll see that this is the case. Figure 5.3 provides an anatomical dissection of the single property from that file.

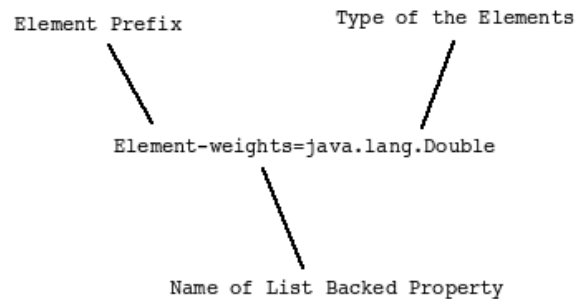


Figure 5.3 Specifying the Type for a List Backed Property

This brief line, `Element-weights=java.lang.Double`, is all the type conversion process needs to add typed elements to our `List`. Now, our `List` property will work just like the array property; each individual element will be converted to a `Double`. Here's the markup from `ListsDataTransferTest.jsp` for our new `weights` property.

```
<s:textfield name="weights[0]" label="weights"/>
<s:textfield name="weights[1]" label="weights"/>
<s:textfield name="weights[2]" label="weights"/>
```

Note, we could have used the non-indexed OGNL notation. You may have a preference but it doesn't matter to the framework. On the Java side, the property on the action object doesn't really change. Here's the implementation of the `List` property from the `DataTransferTest.java`.

```
private List weights;

public List getWeights() {
    return weights;
}

public void setWeights(List weight) {
    this.weights = weight;
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

You might have noticed that its absolutely no different from the previous untyped version. This shouldn't be too surprising though. Since the `List` is typeless from a Java point of view, we can't really see that the elements are `Doubles` unless we try to cast them at run time. Unless you want to modify the source code and test that the elements are actually `Doubles`, you'll just have to take our word for it. Of course, when you use this technique in a real application, whose business logic will depend on those elements being `Doubles`, you'll know soon enough that they are, in fact, `Doubles`. Note, if you want to see this in action, its right there on the same page as the previous example.

Before moving on, we need to make one warning regarding this type specification process. When specifying the type for `Lists` and other `Collections`, take care not to preinitialize your `List`. If you do you will get an error. While untyped `Lists` will work whether you initialize your `List` or not, with type specific `List` conversion, you can NOT preinitialize your `List`.

WARNING: Don't preinitialize your `List` when using the typed element conversion mechanism supported by the `ClassName-conversion.properties` file.

Now, we want to show a full power example. This example will use a `List` property that specifies the Struts 2 Portfolio's `User` class as its element type. This allows us to take advantage of the convenience of using `Lists` with the convenience of using our domain objects at the same time. Also, we just wanted to prove to you that the type specification was actually working. Here's the markup, again from `ListsDataTransferTest.jsp` that accesses our `List` of `Users`.

```
<s:textfield name="users[0].username" label="Usernames"/>
<s:textfield name="users[1].username" label="Usernames"/>
<s:textfield name="users[2].username" label="Usernames"/>
```

As you can see, these field names reference the `username` property on a `User` element in the `users` `List`, which is a property on our action. As before, from the property itself, which comes from the `DataTransferTest.java` source, we can discern nothing about the type of the elements it will contain. Here's the property itself from our `DataTransferTest` action class.

```
private List users ;

public List getUsers()
{
    return users;
}

public void setUsers ( List users ) {
    this.users=users;
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

In order to make the framework populate our `List` with actual `User` objects, we once again employ the aid of the conversion properties file. The following line in the `DataTransferTest-conversion.properties` file specifies that our `List` backed property will contain objects of type `User`.

```
Element_users=manning.utils.User
```

This is pretty cool stuff. In case you want to verify the type specific conversion, just check the tags in the `ListsDataTransferSuccess.jsp` page. They are reaching back into the `List` to retrieve usernames that simply wouldn't exist if the elements weren't actually of type `User`. If you've done much work with moving data around in older frameworks, we know you'll be able to appreciate the amount of work that something like this will save you.

Next, we'll point out a use case that you might not immediately infer from the above examples. Let's assume that you have a `List` for which you specify a `User` element type. Now, let's say that you expect an unpredictable amount of element data from the request. This could be due to something like a multiple select box. In this case, you simply combine the indexless naming convention with the deeper property reference. Note the following set of imaginary (they're not in the sample code) text fields.

```
<s:textfield name="users.username" label="Usernames"/>
<s:textfield name="users.username" label="Usernames"/>
<s:textfield name="users.username" label="Usernames"/>
```

This will submit a set of three username strings under the single parameter name of `users.username`. When OGNL resolves this expression, it will first locate the `users` property. Let's assume this is the same `users` property as in the previous example. This means that `users` is a `List` for which the element type has been specified as `User`. This information allows OGNL to use this single parameter name, with its multiple values, to create a `List` and create `User` elements for it, setting the `username` property on each of those `User` objects.

That should be about enough to keep you working with `Lists` for some time to come. Now we'll look at maps in case you need to use something other than a numeric index to reference your data.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Maps

The last out of the box conversion we will cover is conversion to Maps. Similarly to its support for Lists, Struts 2 also supports automatic conversion of a set of values from the HTTP request into a Map property. This conversion works similarly to the Lists conversion. Maps, however, associate their values with keys rather than indexes. This is just an inherent quality of a Java map. This keyed nature of Maps has a couple of implications for the Struts 2 type conversion process. First of all, the OGNL expression syntax for referencing them is a bit different than for Lists because it has to provide a key rather than a numeric index. The second implication involves specifying types for the Map properties. We have already seen that we can specify a type for our List elements. You can do this with Maps also. With Maps, however, you can also specify a type for the key object. As you might expect, both the Map element and key will default to a String if you don't specify a type. We will explore all of this in the examples of this section. Again, the examples can be seen in action on the Chapter Five home page.

We'll start with a simple version of using a Map property to receive your data from the request. Here's the form markup from `MapsDataTransferTest.jsp`.

```
<s:textfield name="maidenNames.mary" label="Maiden Name"/>      | #1
<s:textfield name="maidenNames.jane" label="Maiden Name"/>      | #1
<s:textfield name="maidenNames.hellen" label="Maiden Name"/>    | #1

<s:textfield      name="maidenNames['beth']"      label="Maiden      Name"/>
| #1
<s:textfield name="maidenNames['sharon']" label="Maiden Name"/>
| #1
<s:textfield name="maidenNames['martha']" label="Maiden Name"/> | #1
```

(annotation) <#1 These fields all target the Map backed 'maidenNames' property>

Again the main difference between this and the List property version is that we now need to specify a key value, a string in this case. In this case, we are using first names as keys to the incoming maiden names. As you can see, OGNL provides a couple of syntax options for specifying the key. First, you can use a simple, if somewhat misleading, property notation. Second, you can use a bracked syntax that makes the fact that the property is a map somewhat more evident. It doesn't matter which you use, though you will probably find one or the other more flexible in certain situations. Just to prove that the syntax doesn't matter, all of our fields in this example will submit to the very same property, a Map going by the name of `maidenNames`. Since we haven't specified a type for this map with a type conversion properties file, all of the values will be converted into elements of type `String`. Similarly, our key's will also be treated as `Strings`.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

With the OGNL expressions in place in the form elements, we just need a property on the Java side to receive the data. In this case, we have a Map backed property implemented on the `DataTransferTest.java` action. Here's the property that receives the data from the above form.

```
private Map maidenNames ;

public Map getMaidenNames()
{
    return maidenNames;
}

public void setMaidenNames ( Map maidenNames ) {
    this.maidenNames=maidenNames;
}
```

Nothing special. If you want to see it in action, hit Maps Data Transfer Test on the Chapter Five home page. You'll see that the `MapsDataTransferTest.jsp` page is successfully pulling some data out of the `maidenNames` property.

Now, let's see an example where we do specify the type for our Map elements. First, we just need to add a line to our `DataTransferTest-conversion.properties` file. Here's the line.

```
Element_myUsers=manning.utils.User
```

Again, this simple property simply specifies that the `myUsers` property, found on the `DataTransferTest` action, should be populated with elements of type `User`. How easy is that? Next, we need some form markup to submit our data to the `myUsers` property.

```
<s:textfield name="myUsers['chad'].username" label="Usernames"/>
<s:textfield name="myUsers['jimmy'].username" label="Usernames"/>
<s:textfield name="myUsers['elephant'].username" label="Usernames"/>

<s:textfield name="myUsers.chad.birthday" label="birthday"/>
<s:textfield name="myUsers.jimmy.birthday" label="birthday"/>
<s:textfield name="myUsers.elephant.birthday" label="birthday"/>
```

This form submits the data to a Map property named `myUsers`. Since we've specified an `User` element type for that map we can use OGNL syntax to navigate down to properties, such as `birthday`, that we know will be present on the map elements. Moreover, the automatic conversion of our birthday string to a Java `Date` type will occur automatically even at this depth.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Just to make sure, we'll check the `myUsers` property on the `DataTransferTest.java` source to make sure its still simple.

```
private Map myUsers ;

public Map getMyUsers()
{
    return myUsers;
}

public void setMyUsers ( Map myUsers ) {
    this.myUsers=myUsers;
}
```

Yeah, that's pretty simple. Again, the Java code is still type unaware. You'll still have to cast those elements to `User`'s if you access them in your action logic, but as far as the OGNL references are concerned, both from the input side form fields and from the result side tags, are concerned, you can take their type for granted. We should point out another cool feature while we're here. We've already noted that you don't have to initialize your `Maps` and `Lists`. The framework will create them for you. You might have also noticed, in this example, that the framework is also creating the `User` objects for you as well. Basically, the framework will create all objects it needs as it tunnels down to the level of the `birthday` property on the `User`.

Tip: The framework will automatically instantiate any intermediate properties in deep OGNL expressions if it finds them to be null when attempting to navigate to the target property. This ability to resolve null property access depends upon the existence of a no argument constructor for each of the properties. So, make sure that your classes have no argument constructors.

In addition to specifying a type for the elements, you can specify a type for the key objects when using `Map` properties. Java `Maps` support all objects as keys. Just as with your values, OGNL will treat the name of your parameter as a string that it should attempt to convert to the type you specify. Let's say we want to use `Integers` as the keys for the entries in our `Map` property, perhaps so we can order the values. Let's make a version of the `myUsers` that will use `Integers` as keys. We'll call it `myOrderedUsers`. First we add the following two lines to our `DataTransferTest-conversion.properties` file.

```
Key_myOrderedUsers=java.lang.Integer
Element_myOrderedUsers=manning.utils.User
```

These lines specify the key and element types for our `myOrderedUsers` `Map` property. As they say, we've been through most of this before so we'll go fast. Here's the form markup that submits the data.

```
<s:textfield name="myOrderedUsers['1'].birthday" label="birthday"/>
<s:textfield name="myOrderedUsers['2'].birthday" label="birthday"/>
<s:textfield name="myOrderedUsers['3'].birthday" label="birthday"/>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Here, we use a key value that is a valid `Integer`. If we didn't we'd get a conversion error on the key when the framework tries to turn the string into the `Integer` type. This is no different from the `myUsers` example except the keys are now `Integer` objects rather than `Strings`. Now, let's have a look at the property that receives this data.

```
private Map myOrderedUsers ;

public Map getMyOrderedUsers()
{
    return myOrderedUsers;
}
public void setMyOrderedUsers ( Map myOrderedUsers ) {
    this.myOrderedUsers=myOrderedUsers;
}
```

As you can see the Java property looks no different. It's still just a `Map`. The only important thing is that the name matches the name used in the OGNL expression.

That does it for the built-in type conversions. We've certainly seen a lot of ways to automatically transfer and convert your data. These methods provide a lot of flexibility. The variety of options can seem a bit overwhelming at first. In the end it's a simple process. You make a property to receive the data, then you write OGNL expressions that point to that property.

The next section takes on a bit of an advanced topic. In case you want the framework to convert to some type that it doesn't support out of the box, you can write your own custom converters. It's a simple process, as you'll soon see.

5.4 Customizing Type Conversion

While the built-in type conversions are quite powerful and full featured, sometimes you might want to write your own type converter. You can, if you desire, specify a conversion logic for translating any string to any object. The only thing you need to do is create the string syntax and the Java class, then link them together with a converter. The possibilities are quite limitless. This is a bit of advanced topic, but the implementation is quite simple and will provide insight that might help debugging even if you never need to write your own type converter.

In this section, we will implement a trivial type converter that converts between strings and a simple `Circle` class. This means that we will be able to specify a string syntax that represents a `Circle` object. The string syntax will represent the circles objects in the text based HTTP world, and the `Circle` class will represent the same objects in the Java world. Our convertor will automatically convert between the two just as the built-in converters handle changing the string `'123.4'` into a Java `Double`. The syntax you choose for your strings is entirely arbitrary. For our simple demonstration we will specify a string syntax as follows:

Syntax
`C:r10`

Example
`C:r10`

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

If a request parameter comes in with this syntax, the framework will automatically convert it to a `Circle` object. In this section we'll see how to implement the converter code and how to tell the framework to use our converter.

5.4.1 Implementing a type converter

As we have explained, type conversion is a part of OGNL. Due to this, all type converters must implement the `ognl.TypeConverter` interface. Generally, OGNL type converters can convert between any two data types. In the web application domain, we have a narrower set of requirements. All conversions are made between Java types and HTTP strings. For instance, we convert from `Strings` to `Doubles`, and from `Doubles` to `Strings`. Or, in our custom case, we'll convert from `Strings` to `Circles`, and from `Circles` to `Strings`.

Taking advantage of this narrowing of the conversion use case, Struts 2 provides a convenience base class for developers to use when writing their own type converters. The `org.apache.struts2.util.StrutsTypeConverter` is provided by the framework as a convenient extension point for custom type conversion. The following snippet lists the abstract methods of this class which you must implement.

```
public abstract Object convertFromString(Map context, String[] values,
    Class
        toClass);

public abstract String convertToString(Map context, Object o);
```

When you write a custom converter, as we will do shortly, you merely extend this base class and fill in these two methods with your own logic. This is a straightforward process, as we have noted. The only thing that might not be completely intuitive in the above signatures is the fact that the string that comes into your conversion is actually an array of strings. This is because all request parameter values are actually arrays of string values. It's possible to write converters that can work with multiple values, but, for the purposes of this book, we will stick to a simple case of a single parameter value.

5.4.2 Converting between String's and Circle's

The logic that we put in the conversion methods will largely consist of string parsing and object creation. Certainly not rocket science. As with many of the Struts 2 advanced features, the stroke of genius will be when you decide that a given use case can be handled elegantly by something like a custom type converter. The implementation itself will take much less brainpower. Listing 5.7 shows our `manning.utils.CircleTypeConverter.java` source.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Listing 5.7 The UserTypeConverter Provides Custom Type Conversion between the Java User Type and a String Representation of a User

```
public class CircleTypeConverter extends StrutsTypeConverter {      |#1

    public Object convertFromString(Map context, String[] values, Class
toClass) {

        String userString = values[0];                          |#2
        Circle newCircle = parseCircle ( userString );          |#2
        return newCircle;                                       |#2

    }

    public String convertToString(Map context, Object o) {

        Circle circle = (Circle) o;                             |#3
        String userString = "C:r" + circle.getRadius();         |#3
        return userString;                                       |#3

    }

    private Circle parseCircle( String userString ) throws
TypeConversionException
    {
        Circle circle = null;
        int radiusIndex = userString.indexOf('r') + 1;

        if (!userString.startsWith( "C:r" ) )
            throw new TypeConversionException ( "Invalid Syntax");
        int radius;
        try {
            radius = Integer.parseInt( userString.substring( radiusIndex )
);
        }catch ( NumberFormatException e ) {
            throw new TypeConversionException ( "Invalid Integer Value for
Radius"); }

        circle = new Circle();
        circle.setRadius( radius );
        return circle;

    }
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

```
}
```

```
(annotation) <#1 Implements the Convenience Class StrutsTypeConverter >  
(annotation) <#2 Logic to Convert from String to Circle Object>  
(annotation) <#3 Logic to Convert from Circle Object to String>
```

You should focus on the first two methods. We include the `parseCircle()` method here just to make sure you realize nothing sneaky is going on. The first method, `convertFromString()`, will be used to convert the request parameter into a `Circle` object. This is the exact same thing that is happening behind the scenes when we have been taking advantage of the built-in type conversions that come with Struts 2 by default. The only thing this method does is parse the string representation of a `Circle` and create an actual `Circle` object from it. So much for the mystique of data binding. Going back from a `Circle` object to a string is equally straightforward. We just take the bits of data from the `Circle` object and build the string according to the syntax we specified earlier.

5.4.2 *Configuring the framework to use our converter*

Now that we have our converter built, we have to let the framework know when and where it should be used. We have two choices here. We can configure our converter to be used local to a given action or globally. If we configure our converter local to an action, we just tell the framework to use the converter for when using OGNL to set or get a specific `Circle` property of a specific action. If we configure the converter to be used globally, it will be used every time a `Circle` property is set or retrieved through OGNL anywhere in the application. Let's look at how each of these configurations is handled.

Property Specific

The first choice is to specify that this converter should be used for conversions of a given property on a given action class. We have already worked with the configuration file used for configuring aspects of type conversion for a specific action. We used the `ActionName-conversion.properties` file when we specified types for our `Collection` property's elements in the earlier `Map` and `List` examples. Now we will use the same file to specify our custom converter for a specific `Circle` property.

We have created a specific action to demonstrate the custom type conversion. This action is `manning.chapterFive.CustomConverterTest`. The action does little. Its most important characteristic for our purposes is its exposure of a JavaBeans property with type `Circle`, as seen in the following snippet.

```
private Circle circle;  
  
public Circle getCircle() {  
    return circle;  
}  
  
public void setCircle(Circle circle) {  
    this.circle = circle;  
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

This action has almost no execution logic. It functions almost exclusively as a carrier of our untransformed data. For our purposes, we are only interested in the type conversion that will occur when a request parameter comes into the framework targeting this property. Here's the line from `CustomConverterTest-conversion.properties` that specifies our custom converter as the converter to use for this property.

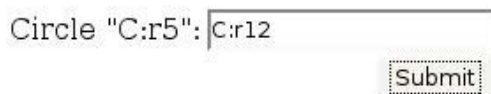
```
circle=manning.utils.CircleTypeConverter
```

This simple line associates a property name, 'circle', with a type converter. In this case, the type converter is our custom type converter. Now, when OGNL wants to set a value on our circle property, it will use our custom converter. Here's the form, from `CustomConverterTest.jsp`, that submits a request parameter targeting that property.

```
<s:form action="CustomConverterTest">
  <s:textfield name="circle" label='Circle' />
  <s:submit />
</s:form>
```

The name is an OGNL expression that will target our property. Since the action object is on top of the ValueStack, and the Circle property is a top level property on that action, this OGNL expression is quite simple. Now, let's try it out. Go to the Custom Converter Test link in the Chapter Five samples. Enter a valid Circle string in the form as shown in Figure 5.4.

Submit the form to test the flexible data transfer of the Struts 2 framework.



The screenshot shows a web form with the label "Circle". Inside the form, there is a text input field with the value "C:r12" and a "Submit" button.

Figure 5.4 Submitting a String that Our Custom CircleTypeConverter will Convert into a Java Circle Object

When this form is submitted, this string will go in as a request parameter targeted at our circle property. Since this property has our custom converter specified as its converter, this string will automatically be converted into a Circle object and set on the circle property targeted by the parameter name. Figure 5.5 shows the result page, `CustomConverterSuccess.jsp`, that confirms that everything has worked according to plan.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Congratulations! You have used a custom converter to create a Circle object from a string and back to a string.

You created a circle with radius equal to 12

Just to check the outgoing data conversion, here's the circle back in the string syntax C:r12

Figure 5.5 The Result Page from our Custom Conversion Test Pulls Data from the Circle Property to Verify that the Conversion Worked

Our result page verifies that the conversion has occurred by retrieving the first and last name from the `Circle` property, which holds the `Circle` object created by the converter. And, just for fun, the result page also tests the reverse conversion by printing the `circle` property as a string again.

While we are at it, try to enter a string that doesn't meet our syntax requirements. If you do, you will be automatically returned to the form input page with an error message informing you that the string you entered was invalid. This useful and powerful mechanism is just a part of the framework's conversion facilities. Tapping into it for your own custom type converters is quite easy. To access this functionality, we throw a `com.opensymphony.xwork2.util.TypeConversionException` when there's a problem with the conversion. In our case, this exception is thrown by our `parseCircle()` method. When we receive the input string value, we do some testing to make sure the string meets our syntax requirements for a valid representation of a `Circle`. If it doesn't, we throw the exception.

Global Type Converters

We just saw how to set up a type converter for use with a specific property of a specific action. We can also specify that our converter be used for all properties of type `Circle`. This can be quite useful. The process differs very little from our previous example and we'll go through it quite quickly without an example. The differences are so minute, you can alter the sample code yourself if you want some first hand proof. Instead of using the `ActionClassName-conversion.properties` file to configure your type converter, you will use the `xwork-conversion.properties` file. As you can probably tell, this is just a global version of the conversion properties. To this file, add a line such as follows:

```
manning.utils.Circle=manning.utils.CircleTypeConverter
```

Now, our custom type converter will run every time OGNL sets or gets a value from a property of the `Circle` type. Quite simple. By the way, the `xwork-conversion.properties` file goes on the classpath, such as in `WEB-INF/classes/`.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

That pretty much wraps up custom type converters. As we promised, the implementation is pretty easy. Now, the challenge, as with custom interceptors, is for you to find something cool to do with them. One of the problems is that most of the converters you will ever need have already been provided with the framework. But even if you never make a custom one, we're sure that knowing how they work will help in your daily development chores. If you do come up with a rad type converter though, don't keep it a secret. Go to the Struts 2 community and let them know. We're all anxious to benefit from your labor!

5.5 Summary

That does it for our treatment of the Struts 2 data transfer and type conversion mechanisms. While you can get away with minimal awareness of much of the information in this chapter, trying to do so will only leave you with frustration and lower productivity in the long run and, even, in the short run. Let's review some of the things we learned about how the framework moves data from one end of the request processing to the other end, all while transparently managing a wide range of type conversions.

The Object Graph Navigation Language (OGNL) is tightly integrated into Struts 2 to provide support for data transfer and type conversion. OGNL provides an expression language that allows developers to map form fields to Java side properties, and it also provides type converters that automatically convert from the strings of the request parameters to the Java types of your properties. The expression language and type converters also work on the other end of the framework when Struts 2 tags in the view layer pages, such as JSP's, pull data out of these Java properties to dynamically render the view.

We conducted an extensive review of the built-in type converters that come with the framework. We saw that they pretty much support all primitives and common wrapper types of the Java language. We also saw that they support a very flexible and rich set of conversions to and from Arrays and Collections. And if that's not enough, you can always build your own custom type converters. Thanks to a convenient base class provided by the framework, implementing a custom converter is pretty easy to do.

As we promised at the onset, this chapter started a two part introduction to OGNL. This chapter's efforts focused more on the OGNL type converters, explained in the context of incoming data. We showed enough OGNL expression language details to make full use of the built-in type conversions to such complex properties types as Maps and Lists. Now, we're ready to head to result side of the framework and see how data is pulled from the model, via tag libraries, and rendered in the view. The tags tend to take more advantage of the full expression language. Accordingly, in the next few chapter, which deal specifically with the tags, we will spend a lot more time on the OGNL expression language. On to Chapter Six Building a View: Tags!

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6 Building a view: tags

In this chapter we'll start looking at the Struts 2 tag library in detail. We will provide a good reference to the tags and clear examples of their usage. We will also finish our exploration of the Object Graph Navigation Language (OGNL). In Chapter 5 Data Transfer: OGNL and Type Conversion, we focuses on the type conversion aspects of OGNL in the context of data entering the framework. Now, we will focus on the OGNL expression language in the context of data exiting the framework through the Struts 2 Tag API. While the last chapter showed us how to map incoming request parameters onto Java properties exposed on the `ValueStack`, this chapter will show you how to pull data off of those properties for the rendering of result pages.

In this chapter, we will explore the syntax of the OGNL EL and we will study the locations in which the its target objects reside. In particular, we will look closely at the `ValueStack` and the `ActionContext`. These objects hold all of the data important to the processing of a given request, including your action object. While it may be impossible to blissfully ignore their existence during much of your development, you should welcome the opportunity to get to know them.

But that won't take very long, actually. And we will certainly make the most of the remainder of the chapter. After we demystify these two obscure repositories, we will provide a reference style catalog of the general use tags in the Struts 2 tag API. These new tags have lots of power and allow you to wield the OGNL expression language to feed them with values. But tags are tags, and they also won't take that long to cover. So, after covering the tags, if we have time, we will provide a concise primer to the advanced features of the OGNL expression language. In the end, you will wield your OGNL expressions confidently as you navigate through the densest of object graphs.

6.1 Getting started

Before we talk about how the details of how Struts 2 tags can help you dynamically pipe data into the rendering of your pages, let's talk about where that data comes from. While we focused on the data moving into the framework in the previous chapter, we now will focus on the data leaving the framework. When a request hits the framework, one of the first things Struts 2 does is create the objects that will store all of the important data for the request. If we are talking about your application's domain specific data, which is the data that you will most frequently access with your tags, it will be stored in a particular location known as the `ValueStack`. But processing a request requires more than just your application's domain data. Other, more infrastructural, data must be stored also. All of this data, along with the `ValueStack` itself, is all stored in something called the `ActionContext`.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.1.1 *The ActionContext and OGNL*

In the last chapter, we used OGNL expressions to bind form field names to specific property locations on objects such as our action object. Actually, we already know that our action object is placed on something called the `ValueStack` and that the OGNL expressions actually target properties on that stack object. In reality, the OGNL expressions can resolve against any of a set of objects. The `ValueStack` is just one of these objects, the default one as it turns out. This wider set of objects against which OGNL expressions can choose to resolve is called the `ActionContext`. We will now see how OGNL chooses which object to resolve against, as well as what other objects are available for accessing with OGNL.

The `ActionContext` is a key behind the scenes player in the Struts 2 framework. If you've worked with other web application frameworks, particularly Struts 1, then you might be asking, "Why do I need an `ActionContext`? Why have you made my life more complicated?" Well, we've been trying hard to emphasize that the Struts 2 framework wants to provide a very clean MVC implementation. The `ActionContext` helps clean things up by providing the notion of a context in which an action occurs. Basically, the `ActionContext` is a simple container for all the important data and resources that surround the execution of a given action. A good example of the type of data we are talking about is the map of parameters from the request, or a map of session attributes from the Servlet Container. In Struts 1, most of these resources were accessed via all of the Servlet stuff handed into the execution of every action. We've already seen how clean the Struts 2 action object has become; it has no parameters in its method signature to tie it to some API that has little to do with its task at hand. So, really your life is much less complicated, though at first it might not seem so.

Before we show you all of the specific things that the `ActionContext` holds, we need to discuss the OGNL integration. As we have seen, OGNL expressions target properties on specific objects. The resolution of each OGNL expression must begin with a single object against which to resolve its property references. Consider, the following OGNL expression:

```
user.account.balance
```

Here, we are targeting the `balance` property on the `account` object on the `user` object. But where is the `user` object located? We must define an initial object upon which we will locate the `user` object itself. Actually, every time you use an OGNL expression, you need to indicate which object the expression should start its resolution against. In Struts 2, each OGNL expression must choose its initial object from those contained in the `ActionContext`. Figure 6.1 shows the `ActionContext` and its most important contents.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

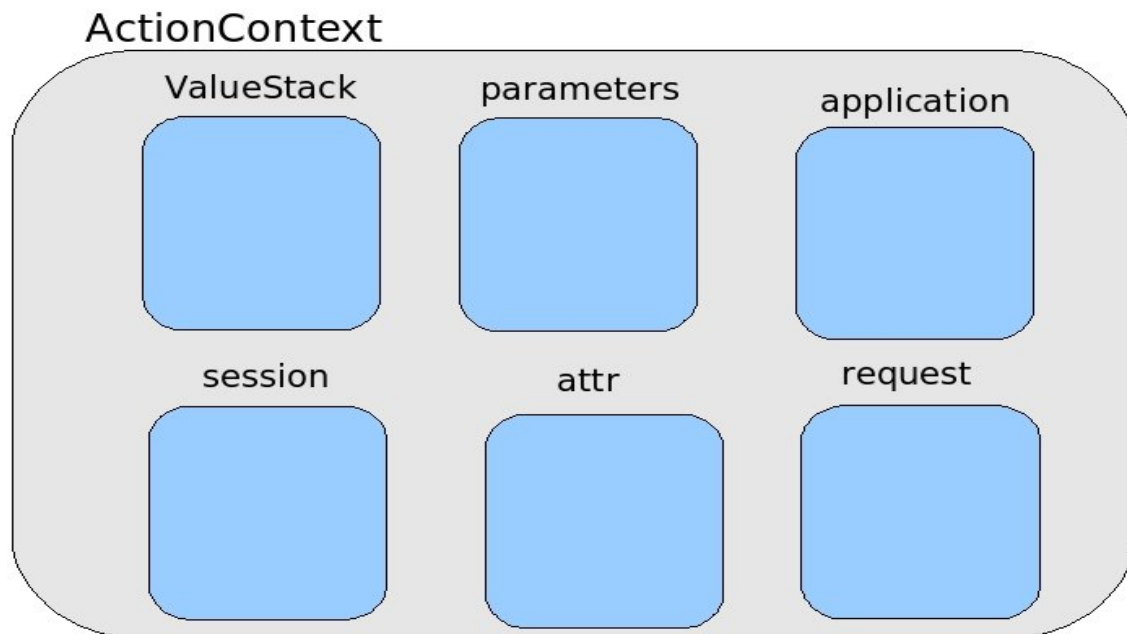


Figure 6.1 The ActionContext Holds All of the Important Data Objects Pertaining to a Given Action Invocation: OGNL Can Target Any of Them

As you can see, the `ActionContext` is full of juicy treasures. The most important of these treasures is the `ValueStack`. As we have said, the `ValueStack` holds your application's domain specific data for a given action invocation. For instance, if you are updating a student, you will expect to find that student data on the `ValueStack`. We will divulge more of the inner workings of the `ValueStack` in a moment. The other objects are all maps of important sets of data. Each of them has a name that indicates its purpose and should be familiar to seasoned Java web application developers as they correspond to specific concepts from the Servlet API. For more information on where the data in these sets comes from, we recommend the Java Servlet Specification, as always. The contents of each of these objects is summarized in Table 6.1.

Table 6.1 The Names and Contents of the Objects and Maps in the ActionContext

Name	Description
parameters	Map of request parameters for this request.
request	Map of request scope attributes.
session	Map of session scope attributes.
application	Map of application scope attributes.
Attr	Returns first occurrence of attribute occurring in page, request, session, or application scope, in that order.
ValueStack	Contains all the application domain specific data for the request.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

The `parameters` object is a map of the request parameters associated with the request being processed, the parameters submitted by the form in other words. The `application` object is a map of the application attributes. The `request` and `session` objects are also maps of request and session attributes. By attribute, we mean the Servlet API concept of an attribute. Attributes allow you to store arbitrary objects, associated with a name, in these respective scopes. Objects stored in application scope are accessible to all requests coming to the application. Objects stored in session scope are accessible to all requests of a particular session. And so forth. Common usages include such things as storing a user object in the session to indicate a logged in user across multiple requests. The `attr` object is a special map that looks for attributes in the following locations, in sequence: page, request, session, and application scope.

Up until now, we've hidden the fact that an OGNL expression must choose one of the objects in the `ActionContext` for its initial object. So, then, how does the framework choose which object to resolve a given OGNL expression against. We certainly did nothing about this while writing our simple input field names in the previous chapter, did we? No we didn't. As with all Struts 2 mysteries, this comes down to a case of intelligent defaults. The `ValueStack` is known as the root object and serves as the default initial object for resolution of all OGNL expressions that don't explicitly name an initial object. You almost don't even have to know that the `ValueStack` exists to use Struts 2. But, take our word, that's not an entirely blissful ignorance.

Though you haven't seen it yet, OGNL expressions can start with a special syntax that names the object from the context against which they should resolve. The following OGNL expression demonstrates this syntax.

```
#session['user']
```

This OGNL expression actively names the session map from the `ActionContext` via the `#` operator of the expression language. The `#` operator tells OGNL to use the named object, located in its context, as the initial object for resolution of the rest of the expression. With Struts 2, the OGNL context is the `ActionContext` and, thankfully, there is indeed a session object in that context. This expression then points to whatever object has been stored under the key `'user'` in the session object, which happens to be the session scope from the Servlet API. This could be, for instance, the user object that our Struts 2 Portfolio login action stores in the session.

As far as the full syntax of OGNL goes, we'll wait a bit on that. In the previous chapter, we saw as much of the OGNL syntax as we needed for writing our input field names that would target our properties. At the most complex, this included expressions that could reference map entries via string and object keys. But OGNL is much more. The OGNL expression language contains many powerful features including the ability to call methods on the objects you reference. While these advanced features give a Struts 2 developer a very high level set of flexible and powerful tools with which to solve the ad hoc thorn that you inevitably find stuck in your side late on a Friday afternoon, they certainly aren't necessary in the normal course of things. We will continue to delay full coverage of the OGNL expression language until the end of this chapter, preferring instead to only introduce as much as we need while demonstrating the tags.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.1.2 The ValueStack: a virtual object

Back to that root object of the `ActionContext`. Understanding the `ValueStack` is critical to understanding the data movement through the Struts 2 framework. By now, you've got most of what you need. We've certainly seen the `ValueStack` in action. When Struts 2 receives a request, it immediately creates an `ActionContext`, a `ValueStack`, and an action object. It then places the action object onto the `ValueStack`. We have seen that we can expose properties, such as username and password, directly on our action objects. These will then receive the data transfer from the request parameters automatically. As we saw in the interceptors chapter, this actually occurs because the `params` interceptor sets those parameters onto properties exposed on the `ValueStack`, upon which the action object sits. While other things, such as the model of the `ModelDriven` interface, may also be placed on the stack, the commonality of all data on the `ValueStack` is its specificity to the application's domain. In MVC terms, the `ValueStack` is the request's view of the application's model data. There are no infrastructural objects, such as Servlet API or Struts 2 objects, on the value stack. The action wouldn't even be there except for the fact that it serves as a locus of domain data in Struts 2. Its not there because of its action logic.

But there's one tricky bit about the `ValueStack`. The `ValueStack` actually pretends to be a single object when OGNL expressions are resolved against it. This virtual object contains all the properties of all the objects that have been placed on the stack. If multiple occurrences of the same property exist, those lowest down in the stack are hidden by the upper most occurrence of a similarly named property. Figure 6.2 shows a `ValueStack` with several objects on it.

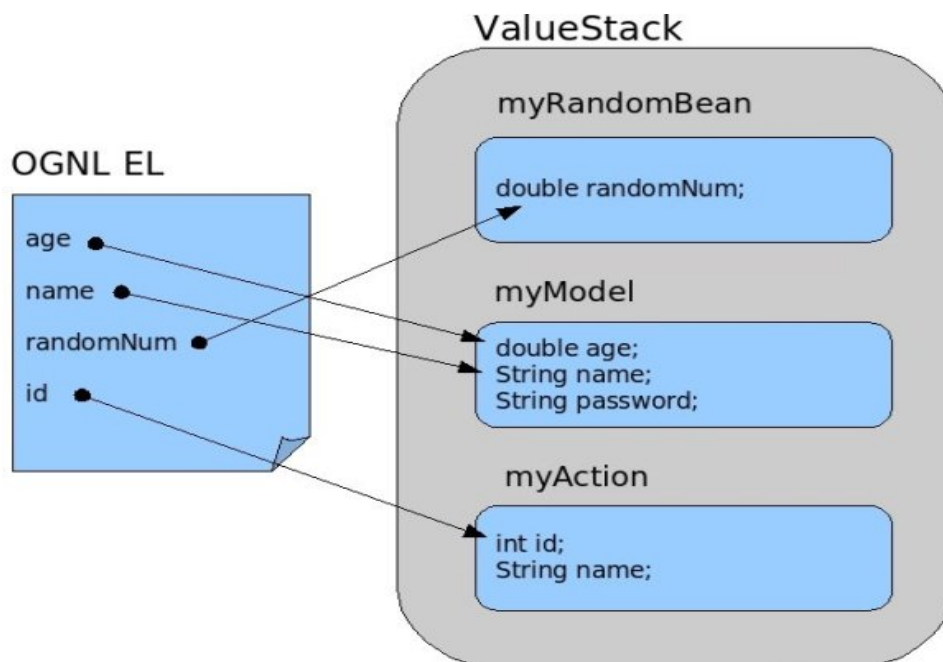


Figure 6.2 The `ValueStack` is the Default Object Against Which All OGNL Expressions are Resolved

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

As you can see in Figure 6.2, references to a given property resolve to the highest occurrence of that property in the stack. While this may seem complicated, it's actually not. As with most Struts 2 features, the flexibility and power to address complex use cases is there, but the common usage can remain ignorant of such details.

Let's examine Figure 6.2. As usual, the action object itself has been placed on the stack first. Then, a model object was added to the stack. This most likely has occurred because the action implements `ModelDriven`. Sometime after that, another object, apparently some sort of random number making bean, was added to the stack. By bean, we simply mean a Java object that either serves as a data carrier or as a utility providing object. In other words, it's usually just some object whose properties, either data or methods providing some sort of utility, you might want to access from your tags with OGNL expressions.

At present, we just want to see how the `ValueStack` appears as a single virtual object to the OGNL expressions resolving against it. In Figure 6.2, we have four very simple expressions. Each targets a top level property. Behind the scenes, OGNL will resolve each of these expressions by asking the `ValueStack` if it has a property such as, for instance, "name". The `ValueStack` will always return the highest level occurrence of the "name" property in its stack of objects. In this case, the action object has a name property, but that property will never be accessed as long as the model object's name property sits on top of it.

Definition: When Java developers talk about "beans" in the context of view technologies, such as JSP's, they frequently mean something slightly different than just a Java object that meets the JavaBeans standards. While these beans are most likely good JavaBeans as well, they don't have to be. The usage in this context more directly refers to the fact that the bean is a Java object that exposes data and / or utility methods for use in JSP tags and the like. Many developers call any object exposed like this a bean. This nomenclature is a historical artifact. In the past, expression languages used in tags couldn't call methods. Thus, they could only retrieve data from an object if it were exposed as a JavaBeans property. Since the OGNL expression language allows you to call methods directly, you could completely ignore JavaBeans conventions and still have data and utility methods exposed to your tags for use while rendering the page. However, in order to keep your JSP pages free of complexity, we strongly recommend following JavaBeans conventions and avoiding expression language method invocation as long as possible.

Just so you don't worry about it, we might as well discuss how that bean showed up on top of the stack. Prior to this point, we've just had stuff automatically placed on the `ValueStack` by the framework. So, how did the darn bean get there? There are many ways to add a bean to the stack. Many of the most common ways to add a bean to the stack occur within the tags that we will soon cover. For instance, the push tag lets you push any bean you like on to the stack. You might do such a thing just before you wanted to reference that bean's data or methods from following tags. We will, of course, demonstrate this with sample code when we cover those tags.

Now that you know where the data is and how to get to it, it's time to start examining the Struts 2 tags libraries that allow you to dynamically blend your data into the rendering of your HTML view pages.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.2 An overview of Struts tags

The Struts 2 tag API provides the functionality to dynamically create robust web pages by leveraging conditional rendering and integration of data from your application's domain model found on the `ValueStack`. Struts 2 comes with many different types of tags. For organizational purposes, they can be broken into four categories: data tags, control flow tags, UI tags, and miscellaneous tags. Since they are a complex topic all to themselves, we'll leave the UI tags for chapter 7. This chapter examines the other three categories.

Data tags focus on ways to extract data from the `ValueStack` and/or set values in the `ValueStack`. Control flow tags give you the tools to conditionally alter the flow of the rendering process. Miscellaneous tags describes a set of last, but not least tags that don't quite fit into the other categories. These left over tags include such useful functionality as management of URL rendering and internationalization of text. Before we get started, we need to make some general remarks about the conventions that are applied across the Struts 2 tag API.

6.2.1 The Struts 2 tag API syntax

The first issue to address is the multiple faces of the Struts 2 tag API. As we have mentioned earlier, Struts 2 tags are defined at a higher level than any specific view layer technology. Using a the tags is as simple as consulting the API. The tag API specifies the attributes and parameters exposed by the tag. Once you identify a tag that you want to use, you simply move on to your view technology of choice – JSP, Velocity or FreeMarker. Interfaces to the tag API have been implemented in all three technologies. The differences in usage amongst the three are so trivial that we will be able to cover them in the following sentences of this short subsection. After that, we present our functional reference of the tags, including a summary of their attributes and parameters. We also include examples of the tags in action. These examples are done in JSP, but we think you will soon see that taking your knowledge of the Struts 2 tags API to one of the other technologies will take approximately zero effort. Let's start with JSP's.

JSP Syntax

The JSP versions of the Struts 2 tags are just like any other JSP tags. The following property tag demonstrates the simple syntax.

```
<s:property value="name" />
```

The only other thing to note is that you must have the property taglib declaration at the top of your page before using the Struts 2 tags. This is standard JSP stuff and the following snippet from one of our Struts 2 Portfolio application's JSP's should show you what you need.

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

The second line, the taglib directive, declares the Struts 2 tag library and assigns them the 's' prefix by which they will be identified, as seen in the property tag above.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Velocity Syntax

You can also use Velocity templates for your view technology. All you need to do is specify your result type to the built-in velocity result type. We will see the details of declaring a result to use a velocity result type in Chapter 8 Results. For now, rest assured that using Velocity is something that the framework supports out of the box. Let's see how the Struts 2 tags are accessed from velocity. In Velocity, the Struts 2 tag API is actually implemented as Velocity macros. This doesn't matter though, the API is still the same. You just need to learn the macro syntax that specifies the same information. Here's the Velocity version of the property tag.

```
#sproperty( "value = name" )
```

Struts 2 tags that would require an end tag may require an #end statement in Velocity. Here's a JSP form tag from the Struts 2 Portfolio application that uses a closing tag.

```
<s:form action="Register">
  <s:textfield name="username" label="Username"/>
  <s:password name="password" label="Password"/>
  <s:textfield name="portfolioName" label="Enter a name"/>
  <s:submit value="Submit"/>
</s:form>
```

And here's the same tag as a Velocity macro.

```
#sform ("action=Register")
  #stextfield ("label=Username" "name=username")
  #spassword ("label=Password" "name=password")
  #ssubmit ("value=Submit")
#end
```

Again, it's the same tag, different syntax. Everything, as pertains to the API, is still the same.

FreeMarker Syntax

The framework also provides out of the box support for using FreeMarker templates as the view layer technology. We will also see how to declare results that use FreeMarker in Chapter 8. For now, here's the same property tag as it would appear in FreeMarker.

```
<@s.property value="name" />
```

As you can see, it's a bit more like the JSP tag syntax. In the end, it won't matter what view layer technology you choose. You can easily access all the same Struts 2 tag functionality from each technology.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.2.2 Using OGNL to set attributes on tags

The integration of OGNL into the tag API has provided a couple of convenient assumptions to make writing tags a little more intuitive. However, you need to know what these assumptions are so that you can navigate the exceptional cases.

In particular, we need to make a distinction between the types of values being passed in to the attributes. In case you haven't thought about it, the attributes to a tag are actually parameters being passed into to the underlying Java execution. As such, they ultimately have data types. In the Struts 2 tag library, a distinction is made in the handling of attributes whose type, in the underlying execution, will be strings and those whose type will be more complex, or non-string if you like. The API always specifies the type of the attribute.

String and non-String attributes

If an attribute is of `String` type, then the value written into the attribute is interpreted as a string literal. If an attribute is some non-`String` type, then the value written into the attribute is interpreted as an OGNL expression. This makes sense because the OGNL expressions point to typed Java properties, thus making a perfect tool for passing in typed parameters. The following property tags demonstrate the difference between string and non-string attribute types.

```
nonExistingProperty on the ValueStack = <s:property  
    value="nonExistingProperty" />  
  
nonExistingProperty on the ValueStack = <s:property  
    value="nonExistingProperty" default="doesNotExist" />
```

Here, we have two somewhat identical uses of the Struts 2 property tag. The use case of the property tag is to write a property, typically from the `ValueStack`, into the rendering page. This property might be of any Java type; the conversion to a string will be handled automatically by the OGNL type converters. The property tag's value attribute tells it the property to render to the page. In the case of these examples, both tags are looking for a property called `nonExistingProperty`.

The property tag will try to locate this property by resolving the value attribute as an OGNL expression. It will look on the `ValueStack`, since no specific object from the `ActionContext` is named with the `#` operator. As it turns out, the `nonExistingProperty` does not exist on the `ValueStack`. What then? A null value will be converted to an empty string. In the case of the first tag, as you can see in Figure 6.3, nothing will render.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

nonExistingProperty on the ValueStack =

nonExistingProperty on the ValueStack = doesNotExist

Figure 6.3 Output from Property Tags with No Value: One Specifies a Default Value While the Other Doesn't

But the second tag does write something, the string `doesNotExist`. The second property tag still tries to pull the `nonExistingProperty` from the `ValueStack`, which again comes up empty. However, it also specifies a default attribute that gives a value to use if the property doesn't exist. Since the purpose of the default attribute is to provide a default string for the tag to put in the page, its type is `String`. When the value given to an attribute is ultimately to be used as a string, it makes the most sense that the default behaviour is to interpret the attribute value as a string literal. Thus, the default value of `doesNotExist` is not used as an OGNL expression. As you can see in Figure 6.3, which shows the output of these two tags, the second tag uses `doesNotExist` as a string literal in its rendering.

Heads up: Attributes passed to Struts 2 tags are divided into two categories. Attributes that will be used by the tag as `String` values are referred to as `String` attributes. Attributes that point to some property on the `ValueStack`, or in the `ActionContext`, are referred to as `non-String` attributes. All `non-String` attributes will be interpreted as OGNL expressions and used to locate the property that will contain the value to be used in the tag processing. All `String` attributes will be taken literally as `Strings` and used as such in the tag processing. You can force a `String` attribute to be interpreted as an OGNL expression by using the `%{expression}` syntax.

Forcing OGNL Resolution

Let's say, assuming the previous example of the `nonExistingProperty`, that you wanted to use a `String` property from the `ValueStack` as your default attribute value. In this case, you wouldn't want the default attribute to be interpreted as a string literal. You'd want it to be interpreted as an OGNL expression pointing to your `String` property. If you want to force OGNL resolution of a `String` attribute, such as the default attribute of the property tag, then you need to use an OGNL escape sequence. This escape sequence is `%{expression}`. The following snippet revisits the scenario from the previous property tag example using the escape sequence to force the default attribute value to be interpreted as an OGNL expression.

```
nonExistingProperty on the ValueStack =  
<s:property value="nonExistingProperty"  
default="%{myDefaultString}" />
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now the default string value will be pulled from the `myDefaultString` property on the `ValueStack`.

Note the similarity between the bracket syntax used to force the `String` attribute to evaluate as an OGNL expression and the JSTL Expression Language syntax.

Struts 2 OGNL Syntax

`%{ expression }`

JSTL Expression Language

`${ expression }`

OGNL uses a `%` instead of a `$`. While this may seem confusing to some JSP veterans, in reality you don't actually use the OGNL escape sequence very often. Due to the intelligent default behaviour of the tags, you can almost always let the tags decide when to interpret your attributes as OGNL expressions and when to interpret them as string literals. This is just another way that the framework eases the common tasks.

We recognize that some of this may be a bit abstract at this point. But now we're going to look at the tags themselves, complete with plenty of sample code.

6.3 Data tags

The first tags we will look at are the Data tags. Data tags let you either get data out of the `ValueStack` or place variables and objects onto the `ValueStack`. In this section, we'll discuss the `property` tag, `set` tag, `push` tag, `bean` tag, and `action` tag. In this reference, our goal is to demo the common usages of these tags. Many of the tags have further functionality for special cases. To find out everything there is to know, consult the primary documentation on the Struts 2 website at <http://struts.apache.org/2.x/>.

All of the examples of tag usages are found in the Chapter Six version of the sample application. Most of these examples use the same action class implementation, `manning.chapterSix.TagDemo`. This simple action conducts no real business logic. It merely populates a couple of properties with data so that the tags have something with which to work. Two properties, `users` and `user`, are populated with a collection of all users in the system and with a selected one of those users, respectively. For the purposes of a tag reference, we will not try to integrate these tags into the core functionality of the Struts 2 Portfolio. We think the working of the tag is better illustrated in this fashion.

6.3.1 The property tag

The `property` tag provides a quick, convenient way of writing a property into the rendering HTML. Typically, these properties will be on the `ValueStack` or on some other object in the `ActionContext`. As these properties can be of any Java type, they must be converted to strings for rendering in the result page. This conversion is handled by the OGNL type converters. If a specific type has no converter, it will typically be treated as a string. In these cases, a sensible `toString()` method should be exposed on the class. Table 6.1 summarizes the most important attributes.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.1 Property tag attributes

Attribute	Required	Default	Type	Description
value	No	<top of stack>	Object	Value to be displayed
default	No		Boolean	Default value to be used if value is null
escape	No	True	Boolean	Whether to escape HTML

Now, let's take a quick look at a property tag in action. As with all the examples in this chapter, you can see this in action by hitting the property tag link on the Chapter Six home page of the sample application. The following tag accesses the user exposed on the ValueStack via our TagDemo action.

```
<h4>Property Tag</h4>
The current user is <s:property value="user.username"/>.
```

The output is quite as you would expect.

Property Tag

The current user is mary.

In this case the user property holds a user with the username of mary. When the property tag pulls the property out to render, it will be converted to a string based upon the appropriate type converters. While this property was a Java String, it still must be formally converted to a text string in the rendering page. See Chapter 5 for more on the type conversion process.

6.3.2 The set tag

In the terms of this tag, “setting” means assigning a property to another name. Various reasons for doing this exist. An obvious use case would be to take a property that needs a deep, complicated OGNL expression to reference it, and reassign, or set, it to a top level name for easier, faster access. This can make your JSP's faster and easier to read.

You can also specify the location of the new reference. By default the property becomes a named parameter in the ActionContext, equal to the ValueStack and session map objects. This means that you can then reference it as a top level named object with an OGNL expression such as #myObject. However, you can also specify that the new reference be kept in one of the scoped maps that are kept in the ActionContext. Table 6.2 provides the attributes for the set tag.

Table 6.2 set tag attributes

Attribute	Required	Type	Description
name	Yes	String	Reference name of the variable to be set in the specified scope
scope	No	String	application, session, request, page, or action. Defaults to action.
value	No	String	Expression of the value you wish to set

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

The example from the Chapter Six sample code is shown below.

```
<s:set name="username" value="user.username"/>
Your username is <s:property value="#username"/>.
```

In this case, we aren't really saving much by making a new reference to the username property. However, it illustrates the point. In this sample, the set tag sets the value from the `user.username` expression to the new reference specified by the name property. Since, we don't specify a scope, this new 'username' reference exists in the `ActionContext`. As you can see, we then reference it with the # operator. Here's the output.

Set Tag

Hello, mary. How are you?

And in case you were wondering what it would look like to set the new reference to a different scope, the following sets the new reference as an entry in the application scope map that is found in the `ActionContext`.

```
<s:set name="username" scope="application"
value="user.username"/>
Your username is <s:property value="#application['username']"/>.
```

Note that we have to use the OGNL map syntax to get at the property in this case. We certainly can't say that we've made any readability gains here, but we have managed to persist the data across the lifetime of the application by moving it to this map. Probably not a good idea to persist a user's username to the application scope, but it does serve to demonstrate the tag functionality;)

6.3.3 The push tag

Whereas the `set` tag allows you to create new references to values, the `push` tag allows you to push properties onto the `ValueStack`. This is useful when you wish to do a lot of work revolving around a single object. With the object on the top of the `ValueStack`, its properties become accessible with first level OGNL expressions. Any time you will access properties of an object more than a time or two, it will probably save a lot of work if you push that object onto the stack. Table 6.3 provides the attribute for the push tag.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.3 Push tag attributes

Attribute	Required	Type	Description
Value	Yes	String	Value to push onto the stack

Here's a the sample markup.

```
<s:push value="user">
  This is the "<s:property value="portfolioName"/>" portfolio,
  created by none other than <s:property value="username"/>
</s:push>
```

As you can see, the push tag has a start tag and close tag. Inside the body of the tag, we can, and do, reference the properties of the user object as top level properties on the ValueStack. The closing tag removes the user from the top of the ValueStack. If you want to see how it renders, you can check out the sample code. But we think you'll find no surprises.

Tip: The push tag, and even the set tag to a limited extent, can be powerful when trying to re-use view layer markup. Imagine you have a JSP template that you would like to re-use across several JSP result pages. Consider the namespace of the OGNL references in that JSP template. For instance, maybe the template's tags uses OGNL references that assume the existence of a User object exposed as a Model object, ala ModelDriven actions. In this case, the templates tags would omit the 'user' property, and refer directly to properties of the user, e.g. <s:property value="username"/>. If you try to include this template into the rendering of a result whose action exposed the user as a JavaBeans property, then this reference would be invalid. It would need to be <s: value="user.username"/>. Luckily, the push tag gives us the ability to push the user object itself to the top of the ValueStack, thus making the top level references of the template valid in the current action. In general, the push tag and the set tag can be used in this fashion.

6.3.4 The bean tag

The bean tag is a bit like a hybrid of the set and push tags. The main difference is that you don't need to work with an existing object. You can create an instance of an object and either push it onto the ValueStack or set a top level reference to it in the ActionContext. By default, the object will be pushed onto the ValueStack and will remain there for the duration of the tag. In other words, the bean will be on the ValueStack for the execution of all tags that occur in between the opening and closing tags of the bean tag. If you want to persist the bean longer than the body of the tag, you can specify a reference name for the bean with the var attribute. This reference will exist in the ActionContext as a named parameter accessible with the # operator.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

There are a few requirements on the object that can be used as a bean. As you might expect, the object must conform to JavaBeans standards by having a zero argument constructor and JavaBeans properties for any instance fields that you intend to initialize with param tags. We will demonstrate all of this shortly. First, Table 6.4 details the attributes for the bean tag.

Table 6.4 Bean tag attributes

Attribute	Required	Type	Description
name	Yes	String	Package and class name of the bean that is to be created
var	No	String	Variable name used if you want to reference the bean outside the scope of the closing bean tag

Our first example demonstrates how to create and store the bean as a named parameter in the `ActionContext`. In this case, we will create an instance of a utility bean that helps us simulate a for loop. This `counter` bean comes with Struts 2. For this example, we will create the bean and use the `var` attribute to store it in the `ActionContext` as a named parameter. The following markup shows how this is done.

```
<s:bean name="org.apache.struts2.util.Counter" var="counter">
  <s:param name="last" value="7"/>
</s:bean>

<s:iterator value="#counter">
  <li><s:property/></li>
</s:iterator>
```

And here's what this markup will render as in the result page.

Bean with Id

- 1
- 2
- 3
- 4
- 5
- 6
- 7

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

Now, let's look at how it works. The bean tag's name attribute points to the class that should be instantiated. The var attribute, repeating a common Struts 2 tag API pattern, specifies the name of the reference under which the bean will be stored in the `ActionContext`. In this case, we call the bean 'counter' and then refer to that bean instance, in the iterator tag's value attribute with the appropriate OGNL. Since the bean is in the `ActionContext`, rather than on the `ValueStack`, we need to use the `#` operator to name it, resulting in the OGNL expression `#counter`. The bean tag is the first of a few tags we'll explore that are parameterized. In the case of the counter, we can pass in a parameter that sets the number of elements it will contain, in effect setting the number of times the iterator tag will execute its body markup.

Now that the counter bean has been created and stored, we can make use of it from the Struts 2 iterator tag to create a simulation of a `for` loop style logic. The bean tag doesn't have to be used with the iterator tag; it's just in this example because the counter bean is meant to be used with the iterator tag. The counter bean works in combination with the iterator tag, which we will cover shortly, to provide a pseudo `for` loop functionality. Generally, the iterator tag iterates over a `Collection`, thus its number of iterations is based upon the number of elements in the `Collection`. For a `for` loop, we want to specify a number of iterations without necessarily providing a set of objects. We just want to iterate over our tag's body a certain number of times. The Counter bean serves as a fake `Collection` of a specified number of dummy objects that allows us to control the number of iterations. In the case of our example, we do nothing more than print a number to the result stream during each iteration.

Note: The bean tag just allows you to create any bean object that you might want to use in the page. In case you want to make your own bean to use with this tag, just remember that it needs to follow JavaBeans conventions on several important points. It has to have no argument constructor, and it must expose JavaBeans properties for any parameters it will receive with the param tag, such as the Counter bean's last param.

Now, let's look at how to use the bean tag to push a newly created bean onto the `ValueStack` rather than store it in the `ActionContext`. While we're at it, we'll further demonstrate the use of the param tag to pump parameters into our home roasted bean. This is all quite simple. To make use of the `ValueStack` as the temporary storage location for our bean, we just use an opening and closing style tag configuration. All tags inside the body of the bean tag will resolve against a `ValueStack` that has an instance of our `JokeBean` on its top. Here's the example.

```
<s:bean name="manning.utils.JokeBean" >
  <s:param name="jokeType">knockknock</s:param>
  <s:property value="startAJoke()" />
</s:bean>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

In this example, also from the Chapter Six sample code, we create an instance of a rather inane utility bean that helps us create jokes. If you look at the sample application, you'll see that this just outputs the first line of a joke -- "knock knock". Though inane, this bean does demonstrate the sense of utility beans. If you want to provide a canned joke component to drop into numerous pages, something that unfortunately does exist in the real world, you could embed that functionality into a utility bean and grab it with the `bean` tag whenever you liked. This keeps the joke logic out of the core logic of the action logic.

This markup demonstrates using the `bean` tag to push the bean onto the `ValueStack` rather than place it as a named reference in the `ActionContext`. Note, we no longer need to use the `var` attribute to specify the reference under which the bean will be stored. When its on top of the `ValueStack`, we don't need an id; we can just refer to its properties and methods directly. This makes our code very concise. Note, the bean is automatically popped from the stack at the close tag. Using the bean is quite easy. In this case, we make use of the OGNL method invocation operator. We do this just to demonstrate that the `bean` tag doesn't have to completely conform to JavaBeans standards – `startAJoke` is clearly not a proper getter. Nonetheless, OGNL has the power to use it.

Finally, note that we pass a parameter into our `JokeBean` that controls the type of joke told by the bean. This parameter is automatically received by our bean as long as the bean implements a JavaBeans property that matches the name of the parameter. If you look at the source code, you can see that we have done this. FYI: this joke bean also supports an 'adult' joke mode, but you'll probably be dissatisfied; its quite innocuous.

The `bean` tag is ultimately straight forward. What you want to be clear about is the difference between the use of the `var` attribute to create a named reference in the `ActionContext`, and the use of the opening and closing tags to work with the bean on the `ValueStack`. The real trick here is in understanding the `ValueStack`, `ActionContext` and how OGNL gets to them. If you are confused by this, you might want to reread the earlier sections of this chapter. With those fundamental concepts in place, the conventions of the `bean` tag, and other similar tags, should be straightforward enough. If you are straight on all of this, congratulate yourself on a mastery of what some consider to be the most Byzantine aspect of Struts 2.

6.3.5 The *action* tag

This tag allows us to invoke another action from our view layer. Use cases for this might not be obvious at first but you will probably find yourself want to invoke secondary actions from the result at some point. Such scenarios might range from integration of existing action components to some wise refactoring of action logic. The practical application of the `action` tag is quite simple. You simply specify another action that should be invoked. Some of the most important attributes of this tag include the `executeResult` attribute, which allows you to indicate whether the result for the secondary action should be written into the currently rendering page or not, and the `name` and `namespace` attributes, by which you identify the secondary action that should fire. By default the namespace of the current action is used.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.5 Action tag attributes

Attribute	Required	Type	Description
name	Yes	String	The action name
namespace	No	String	The action namespace; defaults to the current page namespace
Var	No	String	Reference name of the action bean for use later in the page
executeResult	No	Boolean	When set to true, executes the result of the action (default value: false)
Flush	No	Boolean	When set to true, the writer will be flushed upon end of action component tag (default value: true)
ignoreContextParams	No	Boolean	When set to true the request parameters are not included when the action is invoked (default value: false)

Here's an example that chooses to include the secondary action's result.

```
<h3>Action Tag</h3>
<h4>This line is from the ActionTag action's result.</h4>
<s:action name="TargetAction" executeResult="true"/>
```

Note, the default is to not include it, so we have to actively turn this by setting the `executeResult` attribute to true. Here's what the output looks like.

Action Tag

This line is from the ActionTag action's result.

But this line comes from the result of the TargetAction.

One thing to note is that the result of the secondary action should probably be an HTML fragment if you want it to functionality fit into the primary page.

Often, you might want to the secondary action to fire, but not write a result. One common scenario is that the secondary action can stash domain data into one of the scoped maps that would make it available to the first action. The following markup shows how to target an action in this fashion.

```
<h4>This line is before the ActionTag invokes the secondary
action.</h4>
<s:action name="TargetAction"/>
<h4>Secondary action has fired now.</h4>
<h5>Request attribute set by secondary action = </h5>
<pre> <s:property value="#request.dataFromSecondAction"/></pre>
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Note, the execution of the secondary action is a bit of a side effect unless we reach back to get something that it produced. We retrieve a property that was set, by the secondary action, into the request map, just to prove that the secondary action fired of course. You can check out the output yourself by visiting the Chapter Six sample code. Many times however, a side effect may be just what you want. Note also that the secondary action can receive, or not receive, the request parameters from the primary request, according to the `ignoreContextParams` attribute.

6.4 Control tags

Now that you know how to manipulate and display data, it's time to learn how to navigate around it as well. The Struts 2 tags have a set of tags that make it easy to control the flow of page execution. Using the `iterator` tag to loop over data and the `if/else/elseif` tags to make decisions, you can leverage the power of conditional rendering in your pages.

6.4.1 The iterator tag

Other than the `property` tag, the other most commonly used tag in Struts 2 is the `iterator` tag. The `iterator` tag allows you to loop over collections of objects easily. It's designed to know how to loop over any `Collection`, `Map`, `Enumeration`, `Iterator`, or array. It also provides the ability to define a variable in the `ActionContext`, the iterator status, that lets you determine certain basic information about the current loop state, such as whether you're looping over an odd or even row. Table 6.6 provides the attributes for the `iterator` tag.

Table 6.6 Iterator tag attributes

Attribute	Required	Type	Description
Value	Yes	String	The object to be looped over.
Status	No	String	If specified, an <code>IteratorStatus</code> object is placed in the action context under this name given as the value of this attribute.

We already saw the `iterator` tag in action when we looked at the `bean` tag. Now we'll take a closer look. The Chapter Six sample application includes an example that loops over a set of the Users of the Struts 2 Portfolio. Here's the markup from the result page.

```
<s:iterator value="users" status="itStatus">
  <li>
    <s:property value="#itStatus.count" />
    <s:property value="portfolioName"/>
  </li>
</s:iterator>
```

As you can see, it's pretty straightforward. The action object exposes a set of users and the `iterator` tag iterates over those users. During the body of the tag, each user is in turn placed on the top of the `ValueStack`, thus allowing for convenient access to the user's properties. Note that our iterator also declares an `IteratorStatus` object by specifying the `status` attribute. Whatever name you give this attribute will be the key for retrieving the iterator status object from the `ActionContext`, with an OGNL expression such as `#itStatus` of course.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

In this example, we use the iterator status's count property to sequence our users. Here's the output.

Existing User Portfolios:

- 1 Chad's Portfolio
- 2 Mary's Portfolio

We should probably take a minute to see what else the `IteratorStatus` can provide for us.

Using IteratorStatus

Sometimes it's desirable to know status information about the iteration that's taking place. This is where the `status` attribute steps in. The `status` attribute, when defined, provides an `IteratorStatus` object available in the `ActionContext` that can provide simple information such as the size, current index, and whether the current object is in the even or odd index in the list. The `IteratorStatus` object can be accessed through the name given to the `status` attribute. Table 6.7 summarizes the information that can be obtained from the `IteratorStatus` object.

Table 6.7 public methods of `IteratorStatus`

Method name	Return type
<code>getCount</code>	<code>int</code>
<code>getIndex</code>	<code>int</code>
<code>isEven</code>	<code>boolean</code>
<code>isFirst</code>	<code>boolean</code>
<code>isLast</code>	<code>boolean</code>
<code>isOdd</code>	<code>boolean</code>
<code>modulus(int operand)</code>	<code>int</code>

As you can see, this list provides a powerful set of just the kind of data that can sometimes be hard to come by when trying to produce various effects within JSP page iterations. Happy iterating!

6.4.2 *The if and else tags*

The tags provide the familiar control structures found in every language. Using them is as easy as you might suspect. As you can see, there's just one attribute, a boolean test.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.8 if and elseif tag attribute

Attribute	Required	Type	Description
Test	Yes	Boolean	Boolean expression that is evaluated and tested for true or false

Here's an example of using them. Note, you can put any OGNL expression you like in the test.

```
<s:if test="user.age > 35">This user is too old.</s:if>
<s:elseif test="user.age < 35">This user is too young</s:elseif>
<s:else>This user is just right</s:else>
```

Here we conduct a couple of tests on a user object exposed by our action and, ultimately, found on the ValueStack. The tests are simple boolean expressions and you can chain as many of the tests as you like.

That wraps up the control tags. They are as simple as they seem, and remain that way in use. We still have a few useful tags to cover, and we'll hit them in next section, Miscellaneous tags.

6.5 Miscellaneous tags

As we mentioned at the start of this chapter, Struts 2 includes a few different types of tags. You've already seen how the data tags and control tags work. Let's now look at the miscellaneous tags that, although very useful, can't be easily classified. In this section, we'll discuss the Struts 2 `include` tag (a slight variation of the `<jsp:include>` tag), the `URL` tag, and the `i18n` and `text` tags (both used for internationalization). Finally, we'll take another look at the `param` tag you've already seen, in the context of the `bean` tag, and show how it can be used to its full power.

6.5.1 The *include* tag

Whereas JSP has its own `include` tag, `<jsp:include>`, Struts 2 provides a version that integrates with Struts 2 better and provides more advanced features. In short, this tag allows you to execute a Servlet API style include. This just means that you can include the output of another web resource in the currently rendering page. One good thing about the Struts 2 `include` tag is that it allows you to pass along request parameters to the included resource.

This differs from the previously seen `action` tag in that the `include` tag can reference any servlet resource while the `action` tag can include only another Struts 2 action within the same Struts 2 application. This inclusion of an action stays completely within the Struts 2 architecture. The `include` tag can go outside of the Struts 2 architecture to retrieve any resource available to the web application in which the Struts 2 application is deployed. This generally means grabbing other servlets or JSP's. The `include` tag may not make a lot of sense unless you are pretty familiar with the Servlet API. Again, the Servlet Specification is recommended reading. <http://java.sun.com/products/servlet/download.html>

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.9 provides the attribute for the `include` tag.

Table 6.9 Include tag attribute

Attribute	Required	Type	Description
Value	Yes	String	Name of the page, action, servlet, or Any referenceable URL.

We won't show a specific example of the `include` tag, as its use is fairly straightforward. When using the `include` tag, you should keep in mind that you are including a JSP, servlet, or other web resource directly. The semantics of including another web resource come from the Servlet API. The `include` tag behaves very similarly to the JSP `include` tag. However, it's more useful when you're developing with Struts 2, for two reasons: It integrates better with the framework, and it provides native access to the `ValueStack` and a more extensible parameter model.

Let's start with the framework integration. For example, your tag can dynamically define the resource to be included by pulling a value from the `ValueStack` using the `%{ ... }` notation. (You have to force OGNL evaluation here as the value attribute is of `String` type and would normally be interpreted as a string literal.) Similarly, you can pass in parameters to the included page with the `<s:param>` tag (discussed in a moment). This tag can also pull values from the `ValueStack`. This tight integration with the framework makes the Struts 2 `include` tag a powerful choice.

The Struts 2 `include` tag is also a bit more user friendly. It will automatically rewrite relative URL's for you. If you wish to include the URL `../index.jsp`, you're free to do so even though some application servers don't support that type of URL when using the JSP `include` tag. The Struts 2 `include` tag will rewrite `../index.jsp` to an absolute URL based on the current URL where the JSP is located.

6.5.2 The URL tag

When you're building web applications, it's extremely common to create URLs that link your various pages together. Struts 2 provides a URL tag to help you do this. The tag supports everything you could want to do with a URL from controlling parameters to automatically persisting sessions in the absence of cookies. Table 6.10 lists its attributes.

Table 6.10 URL tag attributes

Attribute	Required	Type	Description
Value	No	String	The base URL; defaults to the current URL the page is rendering from
Id	No	String	If specified, the URL isn't written out but rather is saved in the action context for future use
includeParams	No	String	Selects parameters from all, get, or none; default is get
includeContext	No	Boolean	If true, then the URL that is generated will be prepended with the application's context; default is true
encode	No	Boolean	Adds the session ID to the URL if cookies aren't enabled for the visitor
scheme	No	String	Allows you to specify the protocol; defaults to the current scheme (http or https)

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Here's a couple of examples. First we look at a simple case.

```
URL =      <s:url value="IteratorTag.action"/>

<a href='<s:url value="IteratorTag.action" />'> Click Me </a>
```

And here's the output markup.

```
URL = IteratorTag.action

<a href='IteratorTag.action'> Click Me </a>
```

The URL tag just outputs the generated URL as a string. First we display it for reference. Then we use the same markup to generate the href attribute of a standard anchor tag. Note, we set the target of the URL with the value attribute. This means we must include the action extension ourselves. If we want to target an action, we should probably use the action attribute as seen in the next example.

```
URL =      <s:url action="IteratorTag" var="myUrl">
            <s:param name="id" value="2"/>
        </s:url>
        <a href='<s:property value="#myUrl" />'> Click Me </a>
```

Now, let's see the markup generated by these tags.

```
URL =

<a href='/manningHelloWorld/chapterSix/IteratorTag.action?id=2'>
Click
    Me      </a>
```

As you can see, the url tag did not generate any output in this example. This happened because we used the var attribute to assign the generated URL string to a reference in the ActionContext. This is useful to improve the readability of the code. In this example, our url tag, with its param tags, has become a bit unwieldy to embed directly in the anchor tag. Now we can just pull the URL from the ActionContext with a property tag and some OGNL. This is also useful when we need to put the URL in more than one place on the page.

The param tag used in this example specifies querystring parameters to be added to the generated URL. You can see generated querystring in the output. Note, you can use the includeParams attribute to specify whether parameters from the current request are carried over into the new URL. By default this attribute is set to 'get', which means only querystring params are carried over. You can also set it to 'post', which will cause the posted form params to also be carried over. Or you can specify 'none'.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.5.3 The *i18n* and *text* tags

Many applications need to work in multiple languages. The process of making this happen is called *internationalization*, or *i18n* for short (there are 18 letters between the *i* and the *n* of the word *internationalization*). Chapter 11 discusses Struts 2's internationalization support in detail, but we'd like to take a moment to detail the two tags that are central to this functionality: the *i18n* tag and the *text* tag.

The *text* tag is used to display language-specific text, such as English or Spanish, based on a key lookup into a set of text resources. These mappings are specified in *ResourceBundles*, a standard class in the Java language. We have already seen how Struts 2 uses a properties file version of these resources to implement locale specific error messages as supported by the basic validation provided by *ActionSupport*. In those examples, we saw how the locale can be used to choose messages from properties files that mapped to different languages. We will see these properties file resources again when looking at the *i18n* and *text* tags. Table 6.11 lists the attributes that the *text* tag supports.

Table 6.11 *Text* tag attributes

Attribute	Required	Type	Description
name	Yes	String	The key to look up in the <i>ResourceBundle(s)</i>
Var	No	String	If specified, the text is stored in the action context under this name

As we know from the *ActionSupport* examples, the framework will determine the locale on its own, based upon information from the browser. The framework uses this locale determination to determine the resource bundle from which to pull text messages. If you want to manually specify the resource bundle that should be used, you can use the *i18n* tag. Table 6.12 lists the attributes of the *I18n* tag.

Table 6.12 *i18n* tag attribute

Attribute	Required	Type	Description
name	Yes	String	The name of the resource bundle

Here's a quick example that shows how to set the resource bundle with the *i18n* tag and then extract a message text from it with the *text* tag.

```
<s:i18n name="manning.chapterSix.myResourceBundle_tr">
```

```
  In <s:text name="language"/>,  
  <s:text name="girl" id="foreignWord"/>
```

```
</s:i18n>
```

"<s:property value="#foreignWord"/>" means *girl*.

And here's the output.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Internationalization Tags

In Turkish, "kiz" means girl.

The `il8n` tag simply specifies a resource bundle to use. The bundle is only used during the body of the tag. However, as our example demonstrates you can “persist” a message from the bundle using the `id` attribute of the `text` tag to set the message to the `ActionContext` as a named reference. This usage of the `var` attribute should be becoming more familiar by now. The first `text` tag writes the message associated with the key 'language' directly to the output. The second `text` tag stores the message associated with the key 'girl' under the reference name 'foreignWord'.

These tags are quite simple in use. More importantly though, they play a key role in Struts 2's internationalization support, which we will cover in detail in Chapter 11 Understanding Internationalization.

6.5.4 The *param* tag

The last tag we'll discuss has already been used throughout this chapter. The `param` tag does nothing by itself, but at the same time it's one of the more important tags. It not only serves an important role in the usage of many of the tags covered in this chapter, it will also play a role in many of the UI Component tags, as you'll see in chapter 7. Table 6.13 lists the attributes you're now already familiar with.

Table 6.13 Param tag attributes

Attribute	Required	Type	Description
Name	No	String	The name of the parameter
Value	No	String	The value of the parameter

Usage of the `param` tag has already been established in this chapter. In particular, our coverage of the `bean` tag showed a couple of use cases for the `param` tag, including as a means for passing parameters into your own custom utility objects. As long as you have the general idea, it's just a matter of perusing the API's to see which tags can take parameters. Towards this end, it's always a good idea to consult the online documentation of the Struts 2 tags to see if a given tag can take a parameter of some sort. For one, this book doesn't attempt to be exhaustive in its coverage of the tags. Additionally, the tag API is always being improved and expanded. Struts 2 is a new framework and growth is rapid.

That covers all of the general use Struts 2 tags that we will cover. Chapter Seven will cover the UI Component tags that allow you to quickly develop rich user interfaces in your pages. But before that, we have a couple more topics to hit.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

6.6 Using JSTL and other native tags

What if you want to use the native tags and expression languages of your chosen view layer technology? Well, that's fine too. While the Struts 2 tags provide a high level tag API that can be used across all three view technologies that the framework supports out of the box, you can certainly use the tags and macros provided natively by each of those technologies if you wish. The JSTL, for instance, is still quite available in your JSP pages. We certainly don't cover the JSTL in this book, and assume that if you have enough reason to use JSTL instead of the built in Struts 2 tags, you probably already know quite a bit about the JSTL. We will just note that the result types that prepare the environment for JSP, Velocity, and FreeMarker rendering do make the `ValueStack` and other key Struts 2 data objects available to the native tags of each technology. Bear in mind that the exposure of the Struts 2 objects to those native tags and EL's may not be consistent. This is another example of the flexibility of the platform. Struts 2 will not tie your hands.

Next up, the OGNL expression language details we've been promising for oh so long.

6.7 A Brief Primer for the OGNL Expression Language

Throughout Struts 2 web applications, a need exists to link the Java run time with the text based world of HTML, HTTP, JSP, and other text based view rendering technologies. Somehow, there must be a way for these text based documents to reference run time data objects in the Java environment. A common solution to this problem is the use of expression languages. As we have seen, Struts 2 uses OGNL for this purpose. We now take the opportunity to cover the features of this expression language that you will most likely need to use in Struts 2 development.

Before we get too far into this, we should point out that this section could easily be skipped. For most use cases, you have probably already learned enough OGNL expression language to get by. You could treat this section as a rainy day reference if you like. On the other hand, you'll probably find some stuff that you can use immediately. It's your choice.

6.7.1 What is OGNL?

The Object Graph Navigation Language exists as a mature technology completely distinct from Struts 2. As such, it has purposes and features much larger than its use within Struts 2. OGNL defines itself as an expression and binding language. In Struts 2, we use the OGNL expression language to reference data properties in the Java environment, and we use OGNL type converters to manage the type conversion between HTTP string values and the typed Java values.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

In this last section, we will try to summarize the syntax and some of the more useful features of the OGNL expression language. First we will cover the syntax and features most commonly used in Struts 2 development. Then we will cover some of the other OGNL features that you might find very handy. OGNL has many of the features of a full programming language, so you can expect to find that most everything is possible. Also note that this section makes no intentions to being a complete reference to OGNL. If you want more OGNL power, visit their website at [www.ognl.org](http://www ognl.org) for more information.

Warning: Keep those JSP's clean! While OGNL has much of the power of a full featured language, you might want to think twice before squeezing the trigger. It's a very well established best practice that you should keep business logic out of your pages. If you find yourself reaching for the OGNL power tools, you might very well be pulling business logic into your view layer. We're not saying you can't do it, but we recommend giving a moment's thought before complicating your view pages with too much code style logic. If you are getting very complex in your OGNL , ask yourself if the thing you are doing should be done in the action or, at least, encapsulated in a helper bean that you can use in your page.

6.7.2 Expression language features commonly used in Struts 2

First we should just review the most common usages of the OGNL expression language in Struts 2 development. In this section we'll look at how the expression language serves its purpose in the most common cases of daily development. Basically, we use it to map the incoming data onto your `ValueStack` objects, and we use it in tags to pull the data off of the `ValueStack` while rendering the view. Let's look at the expression language features most commonly used in this work.

Referencing bean properties

First of all, we need to define what makes an expression. The OGNL expression language refers to something called a chain of properties. This concept is quite simple. Take the following expression:

```
person.father.father.firstName
```

This property chain consists of a chain of four properties. We can say that this chain references, or targets, the `firstName` property of the person's grandfather, if you will. You can use this same reference both for setting and getting the value of this property, depending upon your context.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Setting or getting?

When we use OGNL expressions to name our form input parameters, we are referring to a property that we would like to have set for us. The following code snippet shows the form from our Struts 2 Portfolio application's `Registration.jsp` page.

```
<s:form action="Register">
  <s:textfield name="username" label="Username"/>
  <s:password name="password" label="Password"/>
  <s:textfield name="portfolioName" label="Enter a portfolio
name."/>
  <s:submit/>
</s:form>
```

The name of each input field is an OGNL expression. These expressions refer to, for example, the `username` property exposed on the root OGNL object. As we have just learned, the root object is our `ValueStack`, which probably contains our action object and perhaps a model object. When the `params` interceptor fires, it will take this expression and use it to locate the property onto which it should set the value associated with this name. Of course, it will also use the OGNL type converters to convert the value from a string to the native type of the target property.

There is one common complication that arises when the framework moves data onto the properties targeted by the OGNL expressions. Take the deeper expression:

```
user.portfolio.name
```

If a request parameter targets this property, its value will be moved onto the `name` property of the `portfolio` object. One problem that can occur during run time is a `null` value for one of the intermediate properties in the expression chain. For instance, what if the user hadn't been created yet. If you recall, we have been omitting initialization for many of our properties in our sample code. Luckily, the framework handles this.

When the framework finds a `null` property in a chain that it needs to navigate, it will attempt to create a new instance of the appropriate type and set it onto the property. However, this requires two things on the developer's part. First, the type of the property must be a class that conforms to the JavaBeans specification in that it provides a no-argument constructor. Without this, the framework can't instantiate an object of the type. Next, the property must also conform to the JavaBeans specification by providing a setter method. Without this setter, the framework would have no way of injecting the new object into the property. Keep these two points in mind and you'll be good to go.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

In addition to targeting properties onto which the framework should move incoming data, we also use OGNL when the data leaves the framework. After the request is processed, we use the very same OGNL expression to target the same property from a Struts 2 tag. Recall that we said the domain model data stays on the `ValueStack` from start to finish. Thus, tags can read from the same location that the interceptors write. The following snippet from the `RegistrationSuccess.jsp` page shows the property tag doing just this.

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

In this snippet we see that the Struts 2 property tag takes an OGNL expression as its `value` attribute. This expression targets the property from which the property tag will pull the data for its rendering process, a quite simple process where it merely converts the property to a string and writes it into the page.

As you can see, OGNL expressions, as commonly used in Struts 2, serve as pointers to properties. Whether the use case is writing to or reading from that property is up to the context. Though not nearly as common, you can also use the fuller features of the OGNL expression language, operators in particular, to write self contained expressions that themselves, for instance, set the data on a property. But, as this is outside of the normal Struts 2 use case, we will only discuss such features in the advanced section.

Working with Java Collections

Java Collections are a mainstream of the Java web developers daily workload. While the JavaBeans specification has always supported indexed properties, working with actual Java Collections, while convenient in the Java side, has always been a bit of a hassle in contexts such as JSP tags. One of the great things about the OGNL expression language is its simplified handling of Collections. We have already seen this in action when we demonstrated the automatic type conversion to and from complex Collection properties in the last chapter. We will now summarize the OGNL syntax used to reference these properties.

Working with lists and arrays

References to lists and arrays share the same syntax in OGNL. Table 6.14 summarizes the basic syntax to access list or array properties. To see these in action, refer back to the code samples from Chapter 5 that demonstrated type conversion to and from lists and arrays.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.14 OGNL Expression Language Syntax for Referencing Elements of Array and List Properties

Java code	OGNL expression
<code>list.get(0)</code>	<code>list[0]</code>
<code>array[0]</code>	<code>array[0]</code>
<code>((User) list.get(0)).getName()</code>	<code>list[0].name</code>
<code>array.length</code>	<code>array.length</code>
<code>list.size()</code>	<code>list.size</code>
<code>list.isEmpty()</code>	<code>list.isEmpty()</code>

As Table 6.14 demonstrates, the syntax for referencing elements or properties of `Lists` and arrays is quite intuitive. Basically, OGNL uses array index syntax for `Lists` as well as arrays. This makes perfect sense due to the ordered, indexed nature of `Lists`.

A couple of things warrant remarks. First, the reference to the `name` property of a list element assumes something very important. As we know, Java `Lists` are type agnostic. In Java, we always have to cast the element to the appropriate type, in this case `User`, before we try to reference the `name` property. We can omit this in OGNL if we take the time to specify the `Collection` element type as we learned how to do in Chapter 6. This syntax assumes that has been done. We should also note that you can reference other properties, such as `length` and `size`, of arrays and `Lists`. In particular, note that OGNL makes the `List` class's non JavaBeans conformant `size` method answer to a simple property reference. This is just something nice that OGNL provides as free service to its valued customers!

OGNL also allows you to create `List` literals. This can be useful if you want to directly create a set of values to feed to something like a select box. Table 6.15 shows the syntax for creating these literals.

Table 6.15 Creating a List Dynamically in OGNL

Java Code	OGNL Expression
<code>List list = new ArrayList(3);</code>	<code>{1,3,5}</code>
<code>list.add(new Integer (1));</code>	
<code>list.add(new Integer (3));</code>	
<code>list.add(new Integer (5));</code>	
<code>return list;</code>	

You probably only want to do this with trivial data, since creation of complex data in the view layer would make a bit of a mess. Nonetheless, sometimes this will be the perfect tool for the job.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Working with maps

OGNL also makes referencing properties and elements of Maps a delightfully simple chore. Table 6.16 shows a variety of syntax idioms for referencing Map elements and properties.

Table 6.16 OGNL Expression Language Syntax for Referencing Map Properties

Java Code	OGNL Expression
<code>map.get("foo")</code>	<code>map['foo']</code>
<code>map.get(new Integer(1))</code>	<code>map[1]</code>
<code>User user = (User)map.get("userA"); return user.getName();</code>	<code>map['userA'].name</code>
<code>map.size()</code>	<code>map.size</code>
<code>map.isEmpty()</code>	<code>map.isEmpty</code>
<code>map.get("foo")</code>	<code>map.foo</code>

As you can see, you can do quite a lot with maps. The main difference here is that, unlike `Lists`, the value in the index box must be an object. If the value in the box is some sort of numeric data that would map to a Java primitive, such as an `int`, then OGNL automatically converts that to an appropriate wrapper type object, such as an `Integer`, to use as the key. If a string literal is placed in the box, that becomes a string object which will be used for the key. The last row in the table shows a special syntax for maps with strings as keys. If the key is a string, you may use this simpler, JavaBeans style property notation.

As for other object types that you might use as a key, you ultimately have the full power of OGNL to reference objects that might serve as the key. The possibilities are beyond the capacity of the table format. Note, as with the `Lists` syntax, the direct reference, in OGNL, to the name property on the un-cast map element depends upon the configuration of the OGNL type conversion to know the specific element type of the map, as we learned in Chapter 6.

And you can also create Maps on the fly with the OGNL map literal syntax. Table 6.17 demonstrates this flexible feature.

Table 6.17 Creating Maps Dynamically in OGNL

Java Code	OGNL Expression
<code>Map map = new HashMap(2);</code>	<code>#{ "foo" : "bar", "baz" : "whazzit" }</code>
<code>map.put("foo", "bar");</code>	
<code>map.put("baz", "whazzit");</code>	
<code>return map;</code>	
<code>Map map = new HashMap(3);</code>	<code>#{ 1 : "one", 2 : "two", 3 : "three" }</code>
<code>map.put(new Integer(1), "one");</code>	
<code>map.put(new Integer(2), "two");</code>	
<code>map.put(new Integer(3), "three");</code>	
<code>return map;</code>	

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

As you can see, the syntax for creating a Map literal is fairly similar to that for creating a List literal. The main difference is the use of the # sign before the leading brace.

Warning: OGNL uses the # sign in a few different ways. Each is distinct. The uses are completely orthogonal however, so you shouldn't be confused as long as you're alert to the fact that they are different use cases. In particular, this is not the same use of the # sign as we saw when specifying a non-root object from the `ActionContext` for an expression to resolve against. We will also see another use of the # sign in a few moments.

Dynamic maps are especially useful for radio groups and select tags. The Struts 2 tag libraries come with special tags for creating user interface components. These will be explored in Chapter 7 UI Component Tags. For now, just note that you can use literal maps to feed values into some UI components. If you wanted to offer a true/false selection that displays as a Yes/No choice, `{true : 'Yes', false : 'No'}` would be the value for the list attribute. The value for the value attribute would evaluate to either true or false.

Filtering and projecting collections

OGNL supports a couple of special operations that you can conduct on your `Collections`. Filtering allows you to take a collection of objects and filter them according to some rule. For instance, you could take set of users and filter them down to only the users who are more than 20 years old. Projection, on the other hand, allows you to transform a collection of objects according to some rule. For instance, you could take a set of user objects, having both first and last name properties, and transform it into a set of String objects that combined the first and last name of each user into a single string. To clarify, filtering takes a `Collection` of size N and produces a new collection containing a subset of those elements ranging from size 0 to size N. Projecting always produces a `Collection` with a set of elements numbering exactly the same as the original `Collection`; projecting produces a one for one result set.

The syntax for filtering is as follows.

```
collectionName.{? expression }
```

In the expression, you can use `#this` to refer to the object from the collection being evaluated. Note, this is another distinct use of the # sign. The syntax for projection is as follows.

```
collectionName.{ expression }
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Table 6.18 shows some examples of both of these useful operations in action.

Table 6.18 Producing New Collections by Filtering or Projecting Existing Collections

OGNL Expression	Description
users.{?#this.age > 30}	A filtering process that returns a new collection with only users who are older than 30.
users.{username}	A projection process that returns a new collection of username strings, one for each user.
users.{firstName + ' ' + lastName}	A projection process that returns a new collection of strings that represent the full name of each user.
users.{?#this.age > 30}.{username}	A projection process that returns the usernames of a filtered collection of users older than 30.

As you can see, each filtering or project simply returns a new collection for your use. This convenient notation can be used to get the most out of a single set of data. Note, that you can combine filtering and projection operations together. That about covers it for aspects of OGNL that are commonly used in Struts 2. In the next section we will cover some of the advanced features that might help you out in a pinch, but, still, we recommend keeping it simple unless you have no choice.

6.7.4 Advanced expression language features

As we have indicated, OGNL is a fairly full featured expression language. In fact, its features rival that of some full fledged programming languages. In this section, we give a brief summary of some of the advanced features that you might use in a pinch. Note, some of these things are basic features of OGNL, but advanced in the context of Struts 2 usage. Take our terminology with a grain of salt. Also, we will make almost no effort to introduce use cases for these features. We truly consider their usage as non-standard practice. With that said, we also know that these power tools can save the day on those certain occasions that always seem to occur.

Literals and operators

Like most languages, the OGNL expression language supports a wide array of literals. Table 6.19 summarizes these literals.

Table 6.19 Literals of the OGNL Expression Language

Literal type	Example
Char	'a'
String	'hello' "hello"
Boolean	true false
Int	123
Double	123.5

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

BigDecimal	123b
BigInteger	123h

The only thing out of the ordinary would be the usage of both single and double quotes for string literals. Note, however, that a string literal of a single character must use double quotes, or it will be interpreted as a char literal. Table 6.20 shows the operators.

Table 6.20 The Operators of the OGNL Expression Language

Operation	Example
add (+)	2 + 4 'hello' + 'world'
subtract (-)	5 - 3
multiply (*)	8 * 2
divide (/)	9 / 3
modulus (mod)	9 mod 2
increment (++)	++foo foo++
decrement (--)	bar-- --bar
equality (==)	foo == bar

As you can see, all the usual suspects are here. This would probably be a good time to note that the OGNL expression language also allows multiple common separated expressions to be linked in a single expression. The following snippet demonstrates this process.

```
user.age = 10, user.name = "chad", user.username
```

This relatively meaningless example simply demonstrates an expression that links three subexpressions. As with many languages, each of the first two expressions simple executes and passes control on to the next expression. The value returned by the last expression is the value returned for the entire expression. Now, we'll see how to invoke methods with OGNL.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Calling methods

One power that many a JSP developer has wished for is the ability to call methods from the expression language. Until recently, this was pretty rare. Actually, even the simplest property reference is calling a method. But those simple property references are able to invoke methods based upon JavaBeans conventions. If the method you want to invoke does not conform to the JavaBeans conventions, you'll probably need the OGNL method invocation syntax to get to that method. This is one of those things that can sometimes get you out of jam. It can also be quite useful in calling utility methods on helper beans. Table 6.21 shows how it works.

Table 6.21 Calling Methods from the OGNL Expression Language

Java Code	OGNL Expression
<code>utilityBean.makeRandomNumber()</code>	<code>makeRandomNumber()</code>
<code>utilityBean.getRandomNumberSeed()</code>	<code>getRandomNumberSeed()</code> <code>randomNumberSeed</code>

Note, in this table we assume that a random number generator bean, named `utilityBean`, has been pushed on to the `ValueStack` prior to the evaluation of these OGNL expressions. With this bean in place, you can omit the object name in the OGNL expression because it resolves to the `ValueStack` by default. First, we invoke the `makeRandomNumber()` method as you might expect. In the second example, we show that even a JavaBeans conformant property can be accessed with a full method invocation syntax with the same result as the simpler property notation. They are equivalent.

We should note that these method invocation features of the OGNL expression language are turned off during the incoming phase of Struts 2 data transfer. In other words, when the form input field names are evaluated by the `params` interceptor, method invocations, as well as some other security compromising features of the expression language, are completely ignored. Basically, when the `params` interceptor evaluates OGNL expressions it will only allow them to point to properties onto which it should inject the param values. Nothing else is permitted.

Accessing static methods and fields

In addition to accessing instance methods and properties, you can also access static methods and fields with the OGNL expression language. There are two ways of doing this. One requires specification of the fully qualified class name, while the other method resolves against the `ValueStack`. The syntax that takes the full class name is `@[fullClassName]@[property or methodCall]`. Here's are examples of using full class names to access both a static property and a static method.

```
@manning.utils.Struts2PortfolioConstants@USER
@manning.utils.PortfolioUtilityBean@startImageWrapper()
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

Besides that @ signs, these are really no different than the normal property specification or method invocation. As we said, you can forgo specification of the class name if, and only if, your property or method will resolve on the `ValueStack`. Here we have the same two examples, but they assume that some object on the `ValueStack` exposes what they need. The syntax replaces the class name with the `vs` symbol, which stands for `ValueStack`.

```
@vs@USER  
@vs@startImageWrapper()
```

That wraps up our coverage of some of the advanced features of OGNL. You will probably find yourself coming back to this quick reference in the future as you butt heads with some odd wall or two. Again, we recommend taking it easy on the OGNL power tools. However, we are compelled to tell you that OGNL contains even more power features than we've felt comfortable divulging. For the full details, we refer you directly to the primary OGNL documentation found at [www.ognl.org](http://www ognl.org).

6.8 Summary

Well, now, that was a long chapter. That should be about as long as we make them. I'm certainly worn out from writing it. To be fair, a large portion of the chapter was filled up with reference material, screen shots, tables, and a whole lot more. Let's take a moment to consider the range of information that this chapter covered.

This chapter started by trying to clarify the places that data is kept during the processing of a request. This key concept may be one of the most challenging parts of learning Struts 2. Not that it is all that complicated really, it's just a bit different than some frameworks you might have worked with in the past. As we noted, with the cleaned up action component, i.e. no heavy parameter list on the execute method signature, there's a strong need for a location in which to centralize all the data important to the execution of the action. This data makes up the context in which the action executes. Thus, the location in which most of the important data resides is known as the `ActionContext`.

The `ActionContext` contains all kinds of important data ranging from request, session, and application scoped maps to the all important `ValueStack` itself. We saw that we can access all of these data items via the OGNL expression language. In particular, we learned that, by default, OGNL will resolve against the `ValueStack`, but we can also specify a different object from the `ActionContext` for our OGNL expressions by using the # operator to name our initial object specifically. The `ValueStack`, in addition to being the default object for OGNL, is also distinguished by its special qualities. The most important quality of the `ValueStack` is that it presents a synthesis of the properties in the stack as if they were all properties on a single virtual object, with duplicate properties resolving to the instance of the property highest in the stack. Now that we are all clear on this, we should take a moment to celebrate. For many, understanding these sometimes mysterious data repositories can be the biggest hurdle in learning Struts 2.

With all that out of the way, we ran through the Struts 2 tag API at a gallop. The most important things to remember about the tag API are that it is implemented at a layer higher than the specifics of a given view layer technology. Everything we have learned about using the

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=351>

specific tags, though we demoed them in JSP's, can easily be transferred to Velocity or FreeMarker. Just consult the syntactical changes we specified in this chapter and go. The API's are all the same.

Finally, we wrapped up the chapter with a brief primer of the OGNL expression language. We hope you can already see how this powerful language, combined with newly cut set of tag functionality, can be a powerful tool for implementing complex view layer pages without getting snagged up by tag limitations. As we have said many times though, try not to use the OGNL power tools unless you don't have a choice. Often the tasks you will attempt to solve with these will be tasks better solved back in the Java code of the action or business tier objects. But, hey, if its Friday and it has to work or you don't go home . . .

Actually, we've just started our tour of the Struts 2 tag API. This chapter covered the general use tags. In Chapter 7, we will look at the UI component tags. These powerful tags will help us build rich user interfaces for our view layer. We're now deep into the view layer of the Struts 2 framework. But in many ways, its just getting interesting. Wait till you see how easy it is to make powerful forms with the Struts 2 UI tags.