STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

1 Struts 2: the modern web application framework

This chapter covers

- Building applications on the web
- Using web frameworks
- Exploring the Struts 2 framework
- Introducing interceptors and the `ValueStack`

Modern web applications are situated in a complex technological context. Some books that you read might be about a single subject, such as the Java language, or a specific API or library. This book is about Struts 2, a full-featured web application framework for the Java EE platform. As such, this book must take into account the vast array of technologies that converge in the space of the Java EE.

In response to this complexity, we'll start by outlining some of the most important technologies that Struts 2 depends on. Struts 2 provides some powerful boosts to production through convention over configuration, and automates many tasks

that were previously accomplished only by the sweat of the developer. But we think true efficiency comes through understanding the underlying technological context, particularly as these technologies become more and more obscured by the opacity of scaffolding and the like. That said, the first half of this chapter provides a primer on the Struts 2 environment. If you're comfortable with this stuff, feel free to skim or skip these sections entirely.

After sketching the important figures of the landscape, we'll move into a high-level overview of Struts 2 itself. We'll introduce how the Model-View-Controller (MVC) fits into the Struts 2 architecture. After that, we'll go through a more detailed account of what happens when the framework processes a request. When we finish up, you'll be fully ready for chapter 2's HelloWorld application.

Let's get going!

1.1 Web applications: a quick study

This section provides a rough primer on the technological context of a web application. We'll cover the technology stack upon which web applications sit, and take a quick survey of common tasks that all web applications must routinely accomplish as they service their requests. If you're quite familiar with this information, you could skip ahead to the Struts 2 architectural overview in section 1.3, but a quick study of the following sections would still provide an orientation on how we, the authors, view the web application domain.

1.1.1 Using the Web to build applications

While many Java developers today may have worked on web applications for most of their careers, it's always beneficial to revisit the foundations of the domain in which one is working. A solid understanding of the context in which a framework such as Struts 2 is situated provides an intuitive understanding of the architectural decisions made by the framework. Also, establishing a common vocabulary for our discussions will make everything easier throughout the book.

A web application is simply, or not so simply, an application that runs over the Web. With rapid improvements in Internet speed, connectivity, and client/server technologies, the Web has become an increasingly powerful platform for building all classes of applications, from standard business-oriented enterprise solutions to personal software. The latest iterations of web applications must be as full featured and easy to use as traditional desktop applications. Yet, in spite of the increasing variety in applications built on the web platform, the core workflow of these applications remains markedly consistent, a perfect opportunity for reuse. Frameworks such as Struts 2 strive to release 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'll now take a quick look at two of the main components in the technology stack upon which a web application is built. In one 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 context in which Struts 2 is used.

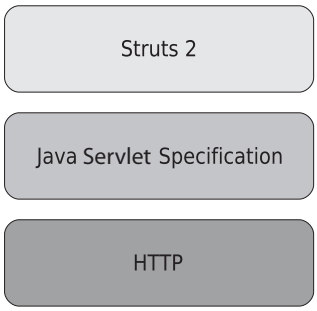

Figure 1.1 The Java Servlet API exposes the HTTP client/server protocol to the Java platform. Struts 2 is built on top of that.

As depicted in figure 1.1, Struts 2 sits on top of two important technologies. At the heart of all Struts 2 applications lie the client/server exchanges of the HTTP protocol. The Java Servlet API exposes these low-level HTTP communications to the Java language. Although it's possible to write web applications by directly coding against the Servlet API, this is generally not considered a good practice. Basically, Struts 2 uses the Servlet API so that you don't have to. But while it's a good idea to keep the Servlet API out of your Struts 2 code, it seems 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 or application server. The client initiates communication by sending a request for a specific resource. The resource 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'll limit ourselves to the most important implications as seen from the perspective of a web application. We can start by noting that HTTP was not originally designed to serve in the capacity that web application developers demand of it. It was meant for requesting and serving static HTML documents. All web applications built on HTTP must address this discrepancy.

For web applications, HTTP has two hurdles to get over. It's stateless, and it's text based. Stateless protocols don't keep track of the relationships among the various requests they receive. 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 be unaware of this client's repeated visits.

But if we are trying to build more complex web applications with more complicated use cases, this won't 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 with all other requests coming from that client during that user session. Without the ability to keep track of relationships among 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, HTTP also is text based. Mating a text-based technology to a strongly typed technology such as 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 objects. A servlet is a singleton Java object whose whole purpose is to receive requests and return responses 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, and then writes and returns the response to the client.

ESSENTIAL KNOWLEDGE

You should know Sun and the Servlet Specification. If you're 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 that includes a variety of interested parties, not the least of which is Sun itself. The specification details the obligations and contracts that the API must honor; actual implementations are provided by various third-party vendors. In the case of the Servlet Specification, the implementations are *servlet containers*. These containers can be standalone 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're unfamiliar with the Servlet Specification, we recommend reading it. It's short, to the point, and well written.

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 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 its clients' requests. A web application's servlets and resources are packaged together in a specific directory structure and zipped up in an archive file with a .war extension. A WAR file is a specialized version of the Java JAR file. The letters stand for *web application archive*. When we discuss chapter 2's HelloWorld application, we'll see exactly how to lay out a Struts 2 application to these standards.

Once you've packaged the web application, you need to deploy it. Web applications are deployed in *servlet containers*. A servlet is a special kind of application known as a *managed life cycle application*. This 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 container determines which servlet should process a request, it invokes 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.

Figure 1.2 shows the relationship between the key players 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 application they ultimately target, must first be handled by the container; it's the server. 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

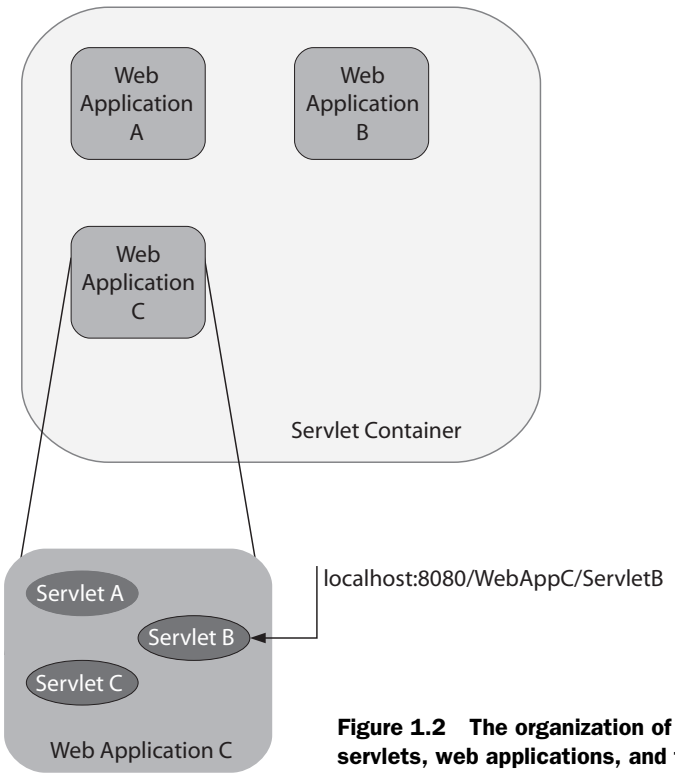

Figure 1.2 The organization of the Servlet API: servlets, web applications, and the servlet container

URL, both the web application and the individual servlet targeted therein can be determined. The full details of this parsing process aren't 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 listening for requests on the `localhost` network interface.

In addition to exposing HTTP to the Java language, the Servlet API provides other important high-level functions, such as a session mechanism that allows us to correlate groups of requests from a given client. As we explained earlier, HTTP doesn't provide a good sense of state across a set of requests, regardless of whether they all came from the same client. This is perhaps the most important benefit, in terms of higher-level functionality, that we receive from servlets. 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 lot of higher-level functionality. It directly encapsulates the details of the client/sever exchange in a set of object-oriented abstractions. This means that we don't have to parse the incoming HTTP request ourselves. Instead, we receive a tidy request object, already wrapped in Java. We say this to make the point that, ultimately, the Servlet API is an infrastructure-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 doesn't attempt to provide solutions for such things. Now that we know what servlets can do, let's look at what they leave undone. These common tasks of the domain are what 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 can now 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

- Binding request parameters to Java types
- Validating data
- Making calls to business logic
- Making calls to the data layer
- Rendering presentation layer (HTML, and so on)
- Providing internationalization and localization

We'll examine each of the concerns briefly in the following paragraphs.

REQUEST PARAMETER BINDING AND DATA VALIDATION

Being a text-based protocol, HTTP must represent its request parameters in a text encoding. When these parameters enter our application, they must be converted to the appropriate native data type. The Servlet API doesn't do this for us. The parameters, as retrieved from the servlet request objects, are still represented as strings. Converting these strings to Java data types is easy enough but 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 that 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; for example, a ZIP code should not have any letters in it.

Then, after the value has been successfully bound to a Java type, the data must be validated against higher-level logic, such as whether a provided ZIP code is valid. An application must determine whether the value itself is within the acceptable range of values according to the business rules of the application. In addition to checking ZIP code validity, you might verify that an email address has the valid structure. Spending too many hours writing this kind of code can certainly make Java Jack a dull boy.

CALLS TO BUSINESS LOGIC AND THE DATA LAYER

Once inside the application, most requests involve calls to business logic and the data layer. While the specifics of these calls vary from application to application, a couple of generalizations can be drawn. First, despite variance in the details of these calls, they form a consistent pattern of workflow. At its core, the processing of each request consists of a sequence of work that must be done. This work is the *action* of an action-oriented framework. Second, the logic and function of this work represents a clear step outside of the web-related domain. If you look back to our list of the common tasks that a web application must do while processing its requests, you'll see that these calls to business logic and the data layer 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 them is that they are outside the specific workflow concerns of a web application.

PRESENTATION RENDERING AND INTERNATIONALIZATION

It could be said that 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 that no longer accurate. At the same time that front-end user interface technology is increasing in complexity, there's an increasing demand for internationalization. Internationalization allows us to build a single web application that can discover the locality of each user and provide locale-specific language and formatting of date, time, and currency. Whether an application returns a simple page of static text or a Gmail-esque super client, the rendering of the presentation layer is a core domain task of all web applications.

We've outlined the domain tasks that all web applications must address. What now? These tasks, by virtue of being common to the processing of nearly every request that comes to a web application, are perfect candidates for reuse. We'd hope that a web application framework would provide reusable solutions to such common tasks. Let's look at how frameworks can help.

1.2 Frameworks for web applications

Now that we've 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. Unless you want to spend hours upon hours solving the tasks outlined in the previous section by hand, you must use a framework, and there are a lot of them. Let's start with a fundamental question.

1.2.1 What's a framework?

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 workflow of a specific domain. The framework then attempts to provide a platform upon which applications of that domain can be more quickly built. The framework does this primarily in two ways. First, the framework tries to automate all the tedious tasks of the domain. 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 a built-in architectural solution that can be easily inherited by applications implemented on the framework.

A FRAMEWORK AUTOMATES COMMON TASKS

Don't reinvent the wheel. Any good framework will provide mechanisms for convenient and perhaps automatic solutions to the common tasks of the domain, saving developers the effort of reinventing 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 tasks such as converting data from HTTP string representation to Java data types, data validation, separation of business and data layer 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.

A FRAMEWORK PROVIDES AN 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 framework's structure 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 see the others in a few pages. When you build an application on a framework, you are buying into that framework's architecture. Sometimes you can fight against the architectural imperative of the framework, but a framework should offer its 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?

You don't have to use a framework. You 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 twenty-first 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 not a bad plan, but it assumes a couple of things. First, it assumes you have lots of smart developers. Second, it assumes 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 smart people, time, and money, there are still drawbacks. I've worked 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're 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-house frameworks seem to be subject to architectural erosion as the years pass, and too many extensions are inelegantly tacked on.

Ultimately, it's hard to imagine creating twenty-first 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, but of which framework offers the solutions you need. With that in mind, it's time to look at Struts 2 and see what kinds of modern conveniences it offers.

1.3 The Struts 2 framework

Apache Struts 2 is a brand-new, state-of-the-art web application framework. As we said earlier, Struts 2 isn't just a new release of the older Struts 1 framework. It is a completely new framework, based on the esteemed OpenSymphony WebWork framework. By now, you should be tuned in to what a web application framework should offer. In terms of the common domain tasks, Struts 2 covers the domain well. It handles all the tasks we've identified and more. Over the course of the book, you'll learn how to work with the features that address each of those tasks in turn. At this introductory stage, it makes more sense to focus on the architectural aspects of the framework. In this section, we'll see how Struts 2 structures the web application workflow. In the next few sections, we'll look at the 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 also true for the first version of Struts. In fact, one of the primary goals of the first Struts

was incorporating the MVC pattern from the desktop application world into a web application framework. The resulting pattern is occasionally called the *Model 2* pattern. This was a critical step in the evolution of well-designed web applications, as it provided the infrastructure for easily achieving the MVC separation of concerns. This allowed developers with few resources for such architectural niceties to tap into a ready-made best practice solution. Struts 1 can claim responsibility for many of the better-designed web applications of the last 10 years.

At some point, the Struts community became aware of the limitations in the first framework. With such an active community, identifying the weak and inflexible points in the framework wasn't 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 interceptors for layering cross-cutting concerns away from action logic; annotation-based configuration to reduce or eliminate XML configuration; a powerful expression language, Object-Graph Navigation Language (OGNL), that transverses the entire framework; and a mini-MVC–based tag API that supports modifiable and reusable UI components. At this point, it's impossible to do more than name drop. We'll have plenty of time to fully explore each of these features. We 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 how the parts of the framework work together when processing a request.

NOTE *Teaching old dogs new tricks, a.k.a. 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 some things to learn, interceptors and OGNL in particular. But while this is 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've worked with Struts 1, you already understand the MVC way of doing things and that knowledge will still be applicable to Struts 2. 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'll find themselves saying, "That's how it always should've been done!"

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 tell you 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 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 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.

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

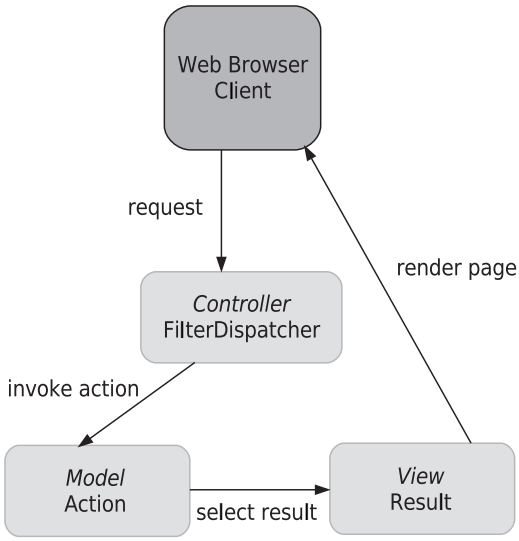

Figure 1.3 Struts 2 MVC is realized by three core framework components: actions, results, and the `FilterDispatcher`.

CONTROLLER—FILTERDISPATCHER

We'll start with the controller. It seems to make more sense to start there when talking about web applications. In fact, the MVC variant used in Struts is often referred to as a *front controller* MVC. This 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 be thought of as commands that the user issues to the application. One of the fundamental tasks of a web application is routing these requests to the appropriate set of 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 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 handles all of the controller work for you. You just need to inform the framework which request URLs map to which of your actions. You can do this with XML-based configuration files or Java annotations. We'll demonstrate both of these methods in the next chapter.

NOTE Struts 2 goes a long way toward the goal of *zero-configuration* web applications. Zero-configuration aims at deriving all of an application's metadata, such as which URL maps to which action, from convention rather than configuration. The use of Java annotations plays an important role in this zero-configuration scheme. While zero-configuration has not quite been achieved, you can currently use annotations and conventions to drastically reduce XML-based configuration.

Chapter 2's HelloWorld application 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 find the model the most nebulous

of the MVC triad. In some ways, the model is a black box that contains the 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 merge together into the monolithic *state* of the application. For instance, if you are logging in to an application, 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 to form one of two states, "authenticated" or "unauthenticated." Neither 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 is too early to go into details, but we'll treat the topic 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 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'll be time to render a view back to the user who submitted the request. Toward this end, an action, upon completing its work, will forward the result to the Struts 2 view component. Let's consider the result now.

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 that 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 clear-cut: 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 groundwork for easily managing the most complex front end.

NOTE 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 handing off control of the processing to another object that will write the response to the client. This makes it easy for alternative responses, such as XML 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. 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 ensures that more result types can be built to handle new types of responses.

Since this favored MVC pattern has been around for decades, try visualizing what the MVC playing field would look like if the players were in fact nicely separated yet connectible. When I explain this to my students, I call it the Reese's peanut butter cup principle. Is this tasty treat chocolate or peanut butter? After your first bite, you discover it's both! How could you use this peanut butter if all you wanted was a PBJ sandwich? And so it goes with technology: how do you get all the richness you desire without actually "combining" the ingredients? Grab some sweets and continue reading to learn about Struts 2 and the framework request-processing factory.

1.3.3 How Struts 2 works

In this section, we'll detail processing a request within the framework. As you'll 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 processing every request. Chief among these are the interceptors, OGNL, and the `ValueStack`. We'll learn what each of these does in the following walkthrough of Struts 2 request processing. Figure 1.4 shows the request processing workflow.

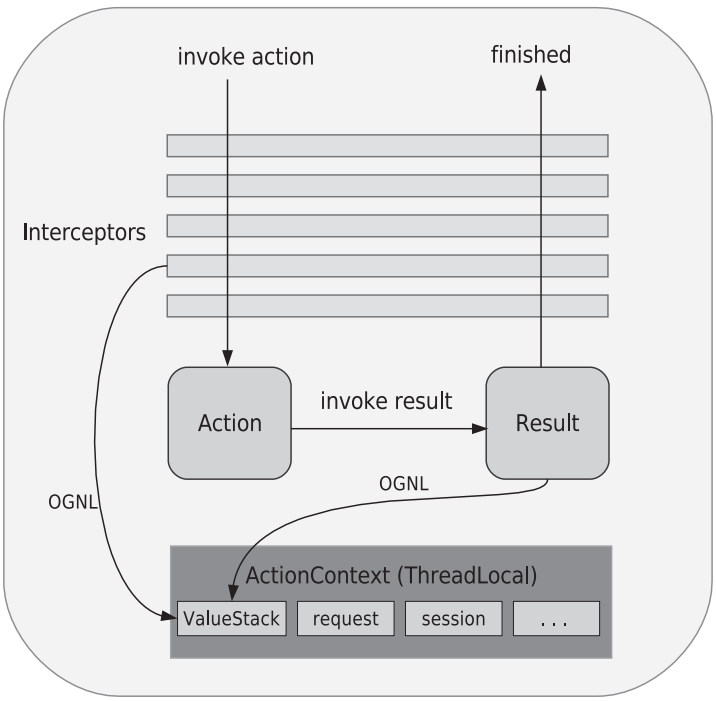

Figure 1.4 Struts 2 request processing uses interceptors that fire before and after the action and result.

The first thing we should consider is that the workflow of figure 1.4 still obeys the simpler MVC view of the framework that we saw earlier. In the figure, the `FilterDispatcher` has already done its controller work by selecting the appropriate action to handle the request. The figure demonstrates what really happens when the action is invoked by the controller. As you can see, a few extra parts are added to the MVC basics. We'll explain in the next paragraphs how the interceptors and the `ActionContext` aid the action and result in their processing of the request.

Figure 1.4 introduces the following new Struts 2 components: `ActionContext`, interceptors, the `ValueStack`, and OGNL. This diagram goes a long way toward showing what really happens in Struts 2. You could say that everything we'll 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

You may have noticed, while studying figure 1.4, that there is a stack of interceptors in front of the action. The invocation of the action must travel through this stack. This is a key part of the Struts 2 framework. We'll devote an entire chapter to this important component later in the book. At this time, it is enough to understand that most every action will have a stack of interceptors associated with it. These interceptors are invoked both before and after the action, though we should note that they actually fire after the result has executed. Interceptors don't necessarily have to do something both times they fire, but they do have the opportunity. Some interceptors only do work before the action has been executed, and others only do work afterward. The important thing is that the interceptor allows common, cross-cutting tasks to be defined in clean, reusable 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 cross-cutting 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 should be done with the invocation of every action, but it probably shouldn't be put in the action itself. Why? Because it's not 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're not specifically related to the action logic of the request. Struts 2 uses interceptors to both separate and reuse these cross-cutting 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 find that many tasks are perfectly solved with custom interceptors. As we said, we'll devote all of chapter 4 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 a storage area that holds all of the data associated with the processing of a request. You could think of it as a piece of scratch paper where the framework does its work while solving the problems of request processing. Rather than passing the data around, Struts 2 keeps it in a convenient, central location—the `ValueStack`.

OGNL is the tool that allows us to access the data we put in that central repository. More specifically, it 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 anything else. If you're coming from Struts 1, you'll find that these are a couple of the more exotic features of the new framework. Due to this, and the sheer importance of this duo, we'll treat them carefully throughout the book. In particular, chapters 5 and 6 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 for request processing, it is manipulated there during action execution, and it is read from there when the results render their response pages.

The tricky, and powerful, thing about the `ValueStack` and OGNL is that they don't belong to any of the individual framework components. Looking back to 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` but also includes stuff the framework itself will use internally, such as the request, session, and application maps from the Servlet API. You can access these objects yourself if you like; 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 is 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'll 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'll use OGNL to bind HTML form fields to data objects on the `ValueStack` for data transfer, and you'll use OGNL to pull data into the rendering of your JSPs 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 had 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 in chapter 2. 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 good 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 2, 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 what we've learned.

We should probably spend a moment 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. The first responsibility of a framework is to provide an architectural foundation for web applications. We've seen how Struts 2 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 framework. We have also seen the specific framework components that implement the MVC pattern: the action component, the result component, and the `FilterDispatcher`.

The other responsibility of frameworks is 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'll see that the framework comes with many built-in interceptors to handle all the common tasks of the domain. You'll see that not only do they handle the bulk of the core framework functionality, they also can be just the thing to handle some of your own application-level needs. While you can probably avoid writing any interceptors yourself, we hope that the chapter on interceptors will inspire you to write your own.

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. By now you should have a decent grasp of what the framework can do. In the next chapter's HelloWorld application, you'll see a concrete example of the framework components in action. Once we get that behind us, we'll move on to explore the core components of the framework, starting with chapter 3's coverage of the Struts 2 action.