STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

2 Saying hello to Struts 2

This chapter covers

- Declaring your architecture
- Deploying a HelloWorld application
- Building an XML-based application
- Using Struts annotations

In the first chapter, we acquainted ourselves with the web application domain, learned how design patterns and frameworks help developers do their jobs, and conducted a quick survey of the Struts 2 architecture and request-processing pipeline. With that, we've now finished the abstract portion of the book. This chapter, which concludes the introductory section of the book, provides the practical and concrete details to bring the theoretical concepts from the first chapter down to earth. In particular, this chapter will demonstrate the basic Struts 2 architectural components with the HelloWorld sample application. This application isn't intended to demonstrate the full complexity of the framework. As we've said, we'll 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'll introduce at a type of configuration known as *declarative architecture*.

2.1 Declarative architecture

In this book, we use the phrase *declarative architecture* to refer to a type of configuration that allows developers to describe their application architecture at a higher level than direct programmatic manipulation. Similar to how an HTML document simply describes its components and leaves the creation of their runtime instances to the browser, Struts 2 allows you to describe your architectural components through its high-level declarative architecture facility and leave the runtime creation of your application to the framework. In this section, we'll see how this works.

2.1.1 Two kinds of configuration

First, we need to clarify some terminology. In the 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, it can be confusing. Hidden beneath the far-reaching concept of configuration, we can distinguish between two distinct sets of activity that occur in a Struts 2 project. One of these, the declarative architecture, is more central to actually building Struts 2 applications, while the other is more administrative in nature. Conceptually, it's important to distinguish between the two.

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 is flexible, it allows you to tweak its behavior in many areas. If you want to change the URL extension by which the framework recognizes which requests it should handle, you can configure the framework to recognize any extension that you like. By default, Struts 2 looks for URLs ending in `.action`, but you could configure the framework to look for `.do` (the Struts 1.x default extension) or even no extension at all. Other examples of configurable parameters include maximum file upload size and the development mode flag. Due to its administrative nature, we'll explain this type of configuration as we come to topics in the book that can be configured.

For now, we'll focus on how to build web applications.

DECLARING YOUR APPLICATION'S ARCHITECTURE

The more important type of configuration, which we'll refer to as declarative architecture, involves defining the Struts 2 components that your application will use and linking them together—or *wiring* them—to form your required workflow paths. By *workflow path*, we mean which action fires when a particular URL is hit, and which results might be chosen by that action to complete processing.

DEFINITION Declarative architecture is a specialized type of configuration that allows developers 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 runtime 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 primarily consists of specifying which Java class will implement 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'll 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'll look at the nuts and bolts of declaring your architecture. There are two ways to do this: through XML-based configuration files or through Java annotations. Figure 2.1 demonstrates the dual interface to the declarative architecture.

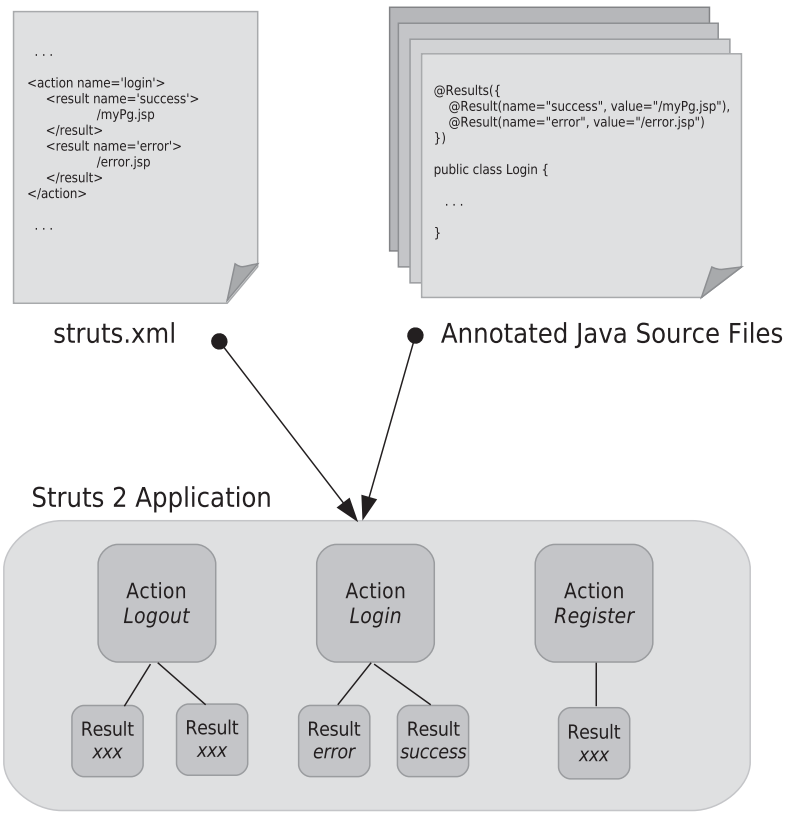

Figure 2.1 Declaring your Struts 2 application architecture with XML or annotations

As you can see, whether your application's Struts 2 components are declared in XML or in annotations, the framework translates them into the same runtime components. In the case of the XML, we have the familiar XML configuration document with elements describing your application's actions, results, and interceptors. With annotations, the XML is gone. Now, the metadata is collected in Java annotations that reside directly within the Java source for the classes that implement your actions. Regardless of which method you use, the framework produces the same runtime 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. Which 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 are already 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 application's components. Listing 2.1 shows an example of XML elements that declare actions and results.

Listing 2.1 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>
```

We won't go into the details of these elements now. We just show this as an example of what 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 it's possible to declare all your components in struts.xml, developers more commonly use this file only to include secondary XML files in order to modularize their applications.

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

JAVA ANNOTATION–BASED DECLARATIVE ARCHITECTURE

A relatively new feature of the Java language, annotations allow you to add metadata directly to Java source files. One of the loftier goals of Java annotations is support for

tools that can read metadata 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. We'll explain how the framework finds the classes that it should scan for annotations when we demo the annotated version of HelloWorld, which we'll do shortly. Listing 2.2 shows what these annotations look like.

Listing 2.2 Using annotations for declarative architecture

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

public class Login implements Action {

   public String execute() {

      //Business logic for login

   }
}
```

Note that 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 metadata that the framework uses to create the runtime components of your application.

We'll fully explain this material throughout the course of the book, but 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 by many to be a more elegant solution than the XML mechanism. For one thing, the annotation mechanism is heavily combined with convention-based deduction of information. In other words, some of the information that must be explicitly specified in the XML elements can be deduced automatically from the Java package structure to which the annotated classes belong. For instance, you 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'll 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 a mechanism for declaring architecture is up to the developer. The most important thing is to understand the concepts of the Struts 2 declarative architecture. If you understand those, moving between the XML or Java annotation–based mechanisms should be quite trivial. This book will use XML in its sample applications. We do this for a couple of reasons. First, we think the XML version is better suited to learning the framework. The XML file is probably more familiar to many of

our readers, and, more importantly, it provides a more centralized notation of an application's components. This makes it easier to study the material when one is first learning the framework. Second, the annotations are a moving target at this point. The Struts 2 developers are ardently moving toward a zero-configuration system that uses convention over configuration, with annotations serving as an elegant override mechanism when conventions aren't followed. Many people are already using this system, but we think at this point it isn't the best approach to learning the framework. We do think that many of you will ultimately choose to use Java annotations to declare your application's components because of their elegance.

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 declaration style 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. 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 minimum development.

These predefined components are part of the Struts 2 intelligent defaults. In case you're interested, many of these components are declared in struts-default.xml, found in the struts2-core.jar. This file uses XML 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'll learn how to take advantage of the components offered in this default package.

2.2 A quick hello

Now we'll present two HelloWorld applications, one with XML and one with annotations, that will bring all this to life. First, we'll introduce the XML version of the application. We'll explore the use of the XML as well as discuss how the application demonstrates the Struts 2 architecture. We'll also introduce the basic layout of a Struts 2 application. Then we'll revisit the same application implemented with annotations, focusing on the annotations themselves. As we've said, the two styles of declaring your architecture are just two interfaces to the 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 Deploying the sample application

To deploy an application, you need a servlet container. This sounds simple, 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 container. It's your choice. Some books attempt to walk you through the installation details for a specific container. The problem with this is that it's never 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're experienced with Java web application development, you'll already have your own container preferences and know how to deploy a web application in your chosen container. If you're new to Java web application development, you can probably expect to spend a few hours reading some online documentation and working through the installation process. Deploying a web application on a running container is typically point-and-click simple. Choosing a servlet container can be overwhelming, but for newbies we recommend Apache Tomcat. It's arguably the most popular open source implementation of the Servlet Specification. It's both easy to obtain 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 sample 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 much of the Java open source community, has strongly adopted Maven 2 as their build/project management tool. If you plan to have more than a fleeting relationship with the Struts 2 source code, 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 in a single Struts 2 web application. This application is packaged in the Struts2InAction.war file. Once you've deployed this web application to your container, point your browser to http://localhost:8080/Struts2InAction/Menu.action to see the main menu for the sample application. Note that this 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 the menu.

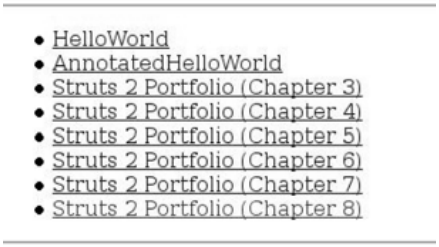

Figure 2.2 The sample application is organized into several mini-applications.

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 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, and then provide a full-featured version for later chapters.

Extra! Extra! Independent HelloWorld WAR

As we go to press, we've responded to feedback from our Manning Early Access Program (MEAP) readers by adding a separate WAR file version of the HelloWorld sample application. Many of our readers wanted to see HelloWorld in the simplest packaging possible. Other readers wanted to see a minimal Struts 2 web application. To fulfill both these requests, we've broken the XML-based HelloWorld out into a standalone web application. Accordingly, you'll find HelloWorld.war also bundled with the downloadable code on the Manning site. This allows you to deploy the HelloWorld application without configuring the database, or other resources, that the full application depends upon. Also, it provides you with a perfect skeleton application for jump-starting your own projects.

Note: We didn't remove HelloWorld from the main sample web application (Struts2InAction.war); we just created the bonus application for your convenience. You can work from either version as you read through this chapter, though we'll assume you're using the full Struts2InAction web application.

THE LAYOUT OF A STRUTS 2 WEB APPLICATION

The entire Struts2InAction.war file can be used as a template for understanding what's required of a Struts 2 web application. Most of the requirements for the application structure come from the requirements put on all web applications by the Servlet API. Figure 2.3 shows the exploded directory structure of the Struts2InAction.war file.

Again, if you aren't familiar with the Servlet Specification, a quick read might be worth your while. For our purposes, we'll 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'll do, but those resources can also load from JAR files on the classpath. In the sample application, we've organized our JSPs according to the chapter 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 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 that there's also a directory called src, but that's our project source code. This is not a required part of the web application directory structure. We've put it here for convenience. You could put it anywhere, depending on the details of your build process. Ultimately, you'll most likely not want source code in a production-ready web application. We've done it this way as a convenient, build–agnostic alternative.

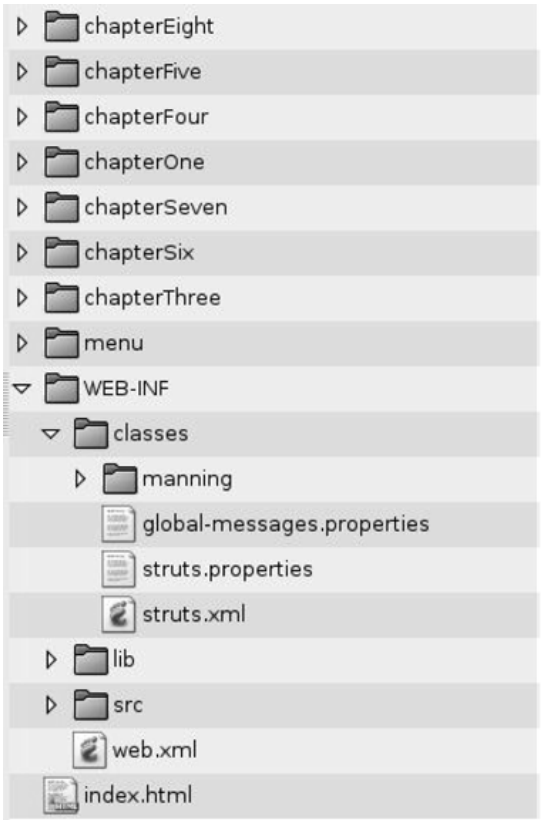

Figure 2.3 The exploded directory structure of our Struts 2 sample application

As for the other two directories, they're essential. The lib directory holds all of the JAR file dependencies that your application needs. In the sample application, we've placed all of the JARs commonly used by Struts 2 web applications. Note that Struts 2 is flexible. If you add some features that we don't use in this book, you might need to add additional JARs. Also note that, if you want to see the absolute minimum set of JARs, you should check out HelloWorld.war, referenced in an earlier sidebar. 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 JARs. 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've 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 application.

Listing 2.3 The web.xml deployment descriptor of our 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>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
    <init-param>
      <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>
     <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>
  </servlet-mapping>

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

</web-app>
```

FilterDispatcher: Struts 2 begins here

Tell Struts where to find annotations

A servlet outside of Struts

For a Struts 2 application, the most important elements in this deployment descriptor are the `filter` and `filter-mapping` elements 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 that target Struts 2 actions. Note the URL pattern to which this filter is mapped: "`/*`." This means the 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're 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've 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 won't have any other servlets in them. In fact, since Struts 2 uses a servlet filter rather than a servlet, many Struts 2 applications won't have any servlets in them—unless you count compiled JSPs. Since it's not uncommon to integrate other servlets with the framework, we've included another servlet, as seen in listing 2.3, in our web application. We'll demonstrate using this servlet later in the book.

Now you should know how to set up a skeletal Struts 2 application. Everything in our sample application is by the book except for the presence of the source directory in WEB-INF. As we said, this has been done as a convenience. You'll probably want to structure your build according to 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.

2.2.2 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 simple workflow. It'll collect a user's name from a form, use that to build a custom greeting for the user, and then present the user with a web page that displays the customized greeting. This workflow, while ultrasimple, will clearly demonstrate the function and usage of all the Struts 2 components, such as actions, results, and interceptors. Additionally, it'll demonstrate the mechanics of how data flows through the framework, including the `ValueStack` and OGNL. As Struts 2 is a sophisticated framework, we'll 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

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

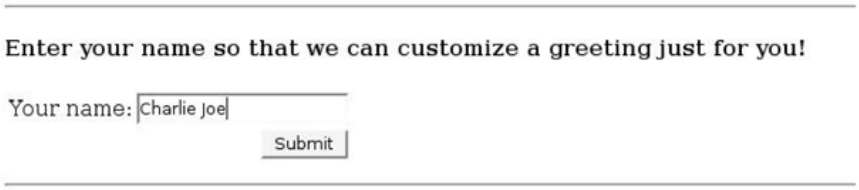

Figure 2.4 The first page collects the user's name.

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

Figure 2.5 The second page presents the customized greeting, built from the submitted name.

The 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've said, the entry point into the XML declarative architecture is the struts.xml file. We've also said that many developers use this root document to include other XML documents, allowing for

modularization. We've 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 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">

<struts>

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

    <package name="default" namespace="/" extends="struts-default">
      <action name="Menu">
        <result>/menu/Menu.jsp</result>
      </action>
    </package>

    <include file="manning/chapterTwo/chapterTwo.xml"/>
    <include file="manning/chapterThree/chapterThree.xml"/>

    . . .

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

</struts>
```

❶ Use constants to tweak Struts properties

Menu action belongs to a default package

Include modularized XML docs

Though off topic, we should note that the constant element ❶ 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'll 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 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 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.

Listing 2.5 Using XML for 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>

   <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>
       </action>

      </package>

</struts>
```

Note that the real file in the application contains detailed comments about the various elements. This is true for all the examples in this book, but when we print listings here we'll remove the comments for clarity. This simple application has only two actions, and one of them hardly does anything. Both the `Name` and the `HelloWorld` actions 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` 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'll be used when the framework maps URLs to these actions. Figure 2.6 shows how the namespace of the package is used to determine the URL that maps to our actions.

The mapping process is simple. The URL combines the servlet context with the package namespace and the action name. Note that the action name takes the .action extension. Chapter 3 will more fully cover the mechanics of namespaces. The first action, the `Name` action, doesn't do any real back-end processing. It merely forwards to the page that will present the user with a form to collect her name.

BEST PRACTICE

Use empty action components to forward to your results, even if they're simple JSPs that require no dynamic processing. This keeps the application's architecture consistent, prewires 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, a well-accepted best practice is to route these requests through actions regardless of their lack of

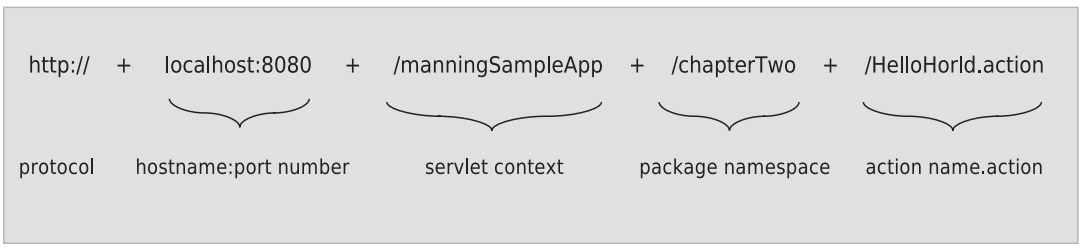

Figure 2.6 Anatomy of a URL: mapping a URL namespace to a Struts 2 action namespace

actual processing. As you can see, such pass-through actions do not specify an implementation class. They'll automatically forward to the result they declare. 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 Using Struts 2 UI component tags to render the form

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

<html>

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

   <body>

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

   </body>

</html>
```

Standard JSP directives

Struts 2 UI component tags

At this point, we provide this listing only for the sake of full disclosure. We'll cover the Struts 2 UI component tags fully in chapter 6. For now, just note that a tag or two will render a complete HTML form. And, as you'll 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 simple, it 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.

Listing 2.7 The `HelloWorld` action's `execute()` does the work

```
package manning.chapterTwo;

public class HelloWorld {

    private static final String GREETING = "Hello ";

    public String execute()  {

      setCustomGreeting( GREETING + getName() );
      return "SUCCESS";
    }

    private String name;
    private String customGreeting;

    public String getName() {
```

The action's business logic

Control string will select result

JavaBeans properties hold the data

```
    return name;
  }

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

  public String getCustomGreeting()
  {
    return customGreeting;
  }

  public void setCustomGreeting( String customGreeting ){
     this.customGreeting = customGreeting;
  }
}
```

As promised, there's not much to it. The `execute()` method contains the business logic, a simple concatenation of the submitted name with the greeting message. After this, the `execute()` method returns a control string that indicates which of its results should render the result page.

ACTION TIP While many Struts 2 actions will implement the `Action` interface, which we'll cover in chapter 3, they're only obligated to meet an informal contract. The `HelloWorld` action satisfies that contract by providing an `execute()` method that returns a string. It doesn't need to actually implement the `Action` interface to informally satisfy the contract.

The only other important thing to note is the presence of JavaBeans properties to hold the application domain data. For now, recall 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 way is using JavaBeans properties. The framework will set incoming data onto these properties when preparing the action for execution. Then, during execution, 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 these properties that the resulting JSP page will read the custom greeting.

Let's look at the JSP that renders the success result for the `HelloWorld` action. Listing 2.8 shows the HelloWorld.jsp file that does this rendering.

Listing 2.8 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>
    </body>
</html>
```

Pulls data from ValueStack

As you can see, this page is quite simple. The only thing to note is the 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.

You might still have questions about how the data gets from the front to the back of this process. 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 1, 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've just learned, the action itself holds the domain data. In the case of the HelloWorld action, that data is held on JavaBeans properties exposed on the action itself. So, what gives?

In short, both are true. The data is both stored in the action and in the `ValueStack`. Here's how. First, domain data is always stored in the action. We'll see variants on this, but it's 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 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.

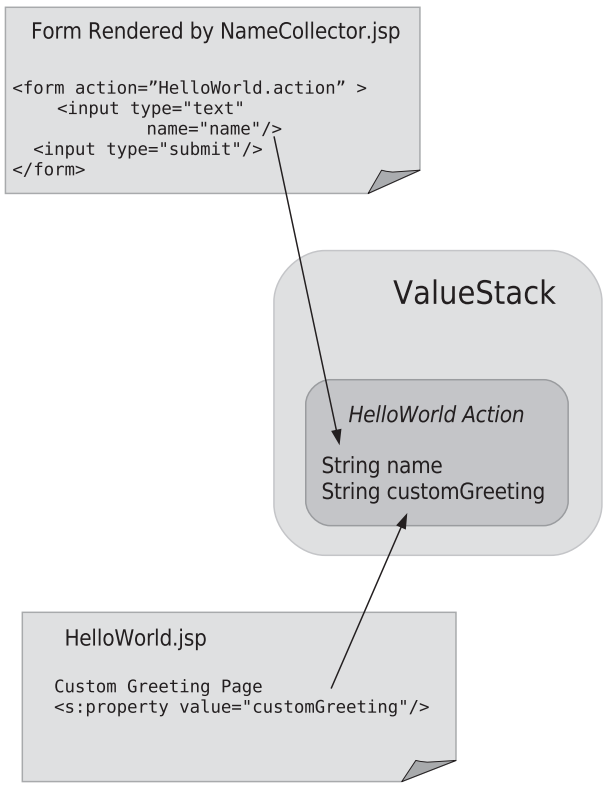

Figure 2.7 Every action is placed on the `ValueStack` so that its properties are exposed to framework-wide OGNL access.

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 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'll cover it fully, particularly in chapters 5 and 6.

That gives us as much as we need to know at this point. We've seen how to declare actions and results. We've also learned a bit about how the data moves through the framework. You might've noticed that we didn't declare any interceptors. Despite the importance of interceptors, the HelloWorld application declares none of them. It avoids declaring interceptors itself by using the default interceptor stack provided by the framework. This is 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.

2.3 HelloWorld using annotations

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

As we've said, the annotations are placed directly in the source code of the actions. 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>
```

Scan manning package for annotations

Note that the value of the parameter is a Java package name. But just telling the system where our action classes are isn't enough. In addition, 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'll 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 annotation version of HelloWorld, one of our actions, the `AnnotatedNameCollector`, implements the `Action` interface to let the system know its true identity. The other action, `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'll refer back to their counterparts in the XML version of HelloWorld. The most notable thing is that several of the elements disappear entirely. We don't have to provide any metadata 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 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 isn't used. The framework only uses the portion of the package namespace beneath the package specified in the `actionPackages` 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 explicitly define the action, as we do with the XML method's `action` element. In fact, it's not possible to define the action ourselves. Since the annotations reside in the action class, the framework takes this as declaration of an action; this is convention over configuration at its finest. We don't need to inform Struts 2 about which class will provide the `action` implementation; it's 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` portion of the class name, if present, is dropped. Second, the first letter is dropped to lowercase. For instance, this version of HelloWorld uses the `AnnotatedHelloWorldAction` class. After removing the ending and changing the case of the first letter, we end up with the following: http://localhost:8080/manningHelloWorld/chapterTwo/annotatedHelloWorld.action

We'll look at the annotations in the `AnnotatedHelloWorldAction` class first, since it's the core 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, where 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 you're 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've been derived from the Java class itself. Finally, the result annotation resides nested within 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'd better say something about this other action. Do you recall that we wrapped NameCollector.jsp in an empty action element in the first example? We said that this use of a pass-through action, rather than hitting JSPs directly, was considered a best practice. We should try to do the same with our annotation version of the application. This is 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 isn't a problem.

If we look at AnnotatedNameCollector.java, we see that it is an empty class. This is shown in the following code snippet. It provides nothing but a container for the result annotation that points to our JSP page. This result annotation provides the 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 */
}
```

There's more going on than we currently need to know, but we'll give the short version for now. This class extends `ActionSupport`. `ActionSupport` is a framework-provided implementation of the `Action` interface we mentioned earlier. In terms of action logic, it does nothing but render the result we've defined. It does provide some support to help some common tasks, which we'll learn about in chapter 3. For now, just note that it's a convenient helper class. Also, 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 should make sure you're not confused. Really, declarative architecture is what we're trying to learn, and it's cleanly designed. If the two styles of metadata 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 and gotten something up and running. Not only have we finished our HelloWorld, we've completed part 1 of this book. 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 refer to this as configuration, we like to distinguish between configuration of the framework itself and configuration of your application's architecture. The former, probably more correctly labeled configuration, involves tweaking or tuning the behavior and performance of the framework. The latter plays a much more central role in the development of our web applications; it involves the nuts and bolts of defining your application's 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've seen, both are fairly simple. While annotations are considered by many to be more elegant than XML, we've opted to use XML because of its educational convenience. But we expect that many of you will ultimately choose annotations over XML. Once you've learned the framework, it'll be easy to start using annotations. As we've indicated, they're a part of a movement toward zero-configuration Struts 2 development. Please check the Struts 2 website for more information.

With the high-level overviews, architectural glosses, and the obligatory HelloWorld out of the way, we've officially completed the introductory portion of the book. Coming up is chapter 3, which kicks off part 2 by providing an in-depth discussion of Struts 2 actions. We'll also start developing our full-featured Struts 2 Portfolio sample application in chapter 3.