STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

Chapter 8. Results in detail. Struts 2 in Action, ISBN: 9781933988078

8 Results in detail

This chapter covers

- Working with results
- Dispatching and redirecting requests
- Building custom results
- Using results with Velocity or FreeMarker

This chapter wraps up part 3. If you recall from the early chapters, the *result* is the MVC view component of the Struts 2 framework. As the central figure of the view, you might be wondering why we started with two chapters on the Struts 2 tag API and left the result for last. Easy. For common development practice, you don't need to know much about the result component itself. In fact, if you use JSP pages for your results, you don't even need to know that the framework supports many different types of results, because the default result type supports JSPs. But the framework comes with support for many different kinds of results, and you can write your own results as well. This chapter explores the details of this important Struts 2 component.

In order to make sure you know what they are and how they work, we start by building a custom result that demonstrates a technique for developing Ajax applications on the Struts 2 platform. Seeing how the framework can easily be adapted to return nontraditional results, such as those required by Ajax clients, serves as a perfect demonstration of the flexibility of the result component. It both teaches

Chapter 8. Results in detail. Struts 2 in Action, ISBN: 9781933988078

you the internals of results and gives you an example of Struts 2 Ajax development. After that, we go on to tour the built-in results that the framework provides for your convenience. These include the default result that supports JSP pages, as well as alternative page-rendering options such as Velocity or FreeMarker templates.

Let's start by refreshing ourselves on the Struts 2 architecture and the role played by results in that architecture.

8.1 Life after the action

To quickly refresh ourselves, a Struts 2 action conducts the work associated with a given request from the client. This work generally consists of a set of calls to business logic and the data tier. We've seen how actions expose JavaBeans properties or model objects for carrying domain data. And we've seen how they provide an `execute()` method as the entry point into their business logic. When the framework determines that a given request should be handled by a given action, the action receives the request data, runs its business logic, and leaves the resulting state of domain data exposed on the `ValueStack`. The last thing the action does is return a control string that tells the framework which of the available results should render the view. Typically, the chosen result uses the data on the `ValueStack` to render some sort of dynamic response to the client.

We've seen this in action throughout the book—in chapter 3 in particular—but we still don't know much about results. So what exactly is a result? In the introductory section of this book, we described the result as the encapsulation of the MVC view concerns of the framework.

DEFINITION A *classic web application architecture* uses the HTTP request and response cycle to submit requests to the server, and receive dynamically created HTML page responses back from that server. This architecture can most notably be distinguished from Ajax web applications that generally receive HTML fragments, XML, or JSON responses from the server, rather than full HTML pages.

In a classic web application, these view concerns are generally equivalent to creating an HTML page that's sent back to the client. The intelligent defaults of the framework are in perfect tune with this usage. By default, the framework uses a result type that works with JSPs to render these response pages. This result, the `dispatcher` result, makes all the `ValueStack` data available to the executing JSP page. With access to that data, the JSP can render a dynamic HTML page.

Thanks to the intelligent defaults, we've been happily using JSPs from our earliest HelloWorld example, all the while oblivious to the existence of a variety of result types. The following snippet shows how easy it is to use JSPs under the default settings of the framework:

```
<action name="PortfolioHomePage" class=". . . PortfolioHomePage">
  <result>/chapterEight/PortfolioHomePage.jsp</result>
</action>
```

Chapter 8. Results in detail. Struts 2 in Action, ISBN: 9781933988078

As the snippet demonstrates, you can get your JSPs up and running without knowing anything about what a result is. All you need to know is that a result is the element into which you stuff the location of your JSP page. And that's about all you need to know as long as your project stays the JSP course.

But what if you want to use Velocity or FreeMarker templates to render your HTML pages? Or what if you want to redirect to another URL rather than rendering a page for the client? These alternatives, which also follow the general request and response patterns of a classic web application, are completely supported by the built-in result types that come with the framework. Starting in section 8.2, we'll provide a tour of these commonly used results. If all you want to do is switch from JSPs to FreeMarker or Velocity, or redirect to another URL instead of rendering the HTML page yourself, you can skip ahead to the reference portion of this chapter.

If you want, however, to see how you can adapt results to nonstandard patterns of usage, such as the nonclassic patterns of Ajax applications, then stick around.

8.1.1 Beyond the page: how to use custom results to build Ajax applications with Struts 2

In the rest of this section, we're going to build a custom result that can return a response suitable for consumption by an Ajax client. This example, while focusing on an Ajax use case, is meant, first and foremost, to serve as a thorough introduction to results. But we've chosen Ajax as our example because we know that all new web application frameworks will have to support Ajax. In short, we feel that Struts 2 provides a strong platform for building Ajax applications, and we believe that the flexibility of the result component is a cornerstone of this strength. Thus, it's only natural to use an Ajax example as our case study of a custom result.

As we mentioned, a classic web application returns full HTML page responses to the client. Figure 8.1 illustrates this pattern.

In figure 8.1, the client makes a request that maps to some action. This action, most likely, takes some piece of request data, conducts some business logic, then exposes the subsequent domain data on the `ValueStack`. The action then passes control to a result that renders a full HTML page, using the prepared data, to build the new HTML page. The key thing here is that the response is a full HTML page, which the client browser uses to rerender its entire window. The response sent back to the client in figure 8.1 is probably rendered by a JSP under the default dispatcher result type. As we've seen, the framework makes this classic pattern of usage easy.

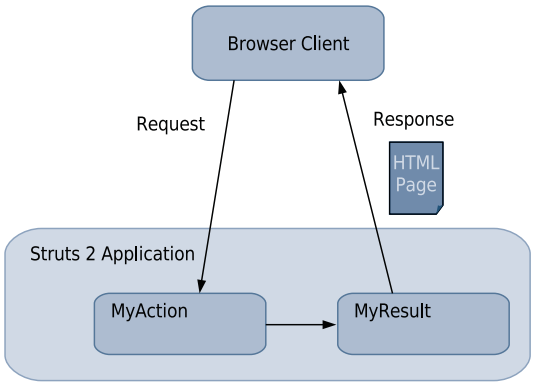

Figure 8.1 Classic web applications return full HTML page responses to the client.

On the other hand, Ajax applications do something entirely different. Instead of requesting full HTML pages, they only want data. This data can come in many forms. Some Ajax applications want HTML fragments as their responses. Some want XML or JSON responses. In short, the content of an Ajax response can be in a variety of formats. Regardless of their differences, they do share one distinct commonality: none of them want a full HTML page. Figure 8.2 illustrates a typical Ajax request and response cycle.

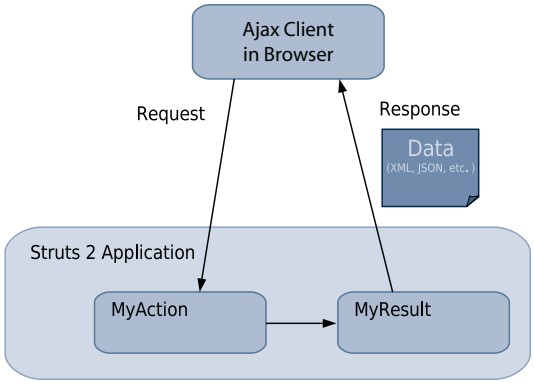

Figure 8.2 Ajax applications expect only data, such as JSON or XML, in the response.

When the Ajax client receives the response, it won't cause the browser to rerender the entire HTML page. On the contrary, it carefully examines the data serialized in the XML or JSON and uses that data to make targeted updates to the affected regions of the current browser page. This is a different kind of response. Luckily, Struts 2 can easily handle this with the flexibility of its result component.

8.1.2 Implementing a JSON result type

One of the most challenging things about developing a modern Ajax-based web application is that most of the web application frameworks weren't exactly designed for the innovative patterns of HTTP communication used by them. As we've indicated, the Struts 2 result component provides the adaptive power to make a variety of Ajax techniques fit the framework like a glove. In this section, we're going to develop a result type that can return JSON responses to an Ajax client.

JavaScript Object Notation (JSON) provides a succinct text-based serialization of data objects. JSON can be a powerfully succinct and lightweight means of communication between a web application server and an Ajax client. By using JSON, we can serialize our Java-side data objects and send that serialization in a response to the client. On the client side, we can easily deserialize this JSON into runtime JavaScript objects and make targeted updates to our client-side view without requiring a page refresh. Sounds pretty clean in theory, and it's even cleaner in practice.

HOT LINK If you want to learn more about JSON, visit the website at http://www.json.org/.

AN AJAX CLIENT TO DEMO OUR RESULT

If we're going to demonstrate a JSON result, we need to actually put some Ajax into our web app. For these purposes, we add an Ajax artist browser to the Struts 2 Portfolio page. This browser provides the visitor with a means of browsing the various artists who're currently hosting portfolios on the site. The visitor can peruse the list of artists and see such details as each artist's full name and the set of portfolios that artist has on the site. From

the chapter 8 home page of the sample application, enter the site as a visitor and follow the link to the artist browser. Figure 8.3 shows the Ajax artist browser.

Figure 8.3 Our artist browser submits Ajax requests to the web application and receives JSON responses via our custom `JSONResult`.

The artist browser is simple. The visitor can peruse some basic information about the selected artist. This information is displayed in the blue box and is updated when the visitor selects a different artist in the select box control. This selection causes an Ajax request to be sent to our Struts 2 application. The response to this request is a JSON serialization of the artist in question. When this response arrives, our JavaScript client code instantiates a JavaScript object from that JSON and uses it to dynamically update the information in the blue window. Nothing else in the window is changed. While simple, this demonstrates a solid Ajax strategy where the page in the browser is an application unto itself. Requests to the server are for JSON-based data, not new pages.

CODING THE JSONRESULT

First things first. If we plan to have Ajax clients demanding JSON responses, we need to build a result that can do that. This leads us to our custom `JSONResult`. Making a custom result is, in some ways, just as easy as writing an action. The only requirement imposed on a result implementation is that it implement the `com.opensymphony.xwork2.Result` interface, as shown:

```
public interface Result extends Serializable {
   public void execute(ActionInvocation invocation) throws Exception;
}
```

As with most framework components, a simple interface defines a highly decoupled and nonrestrictive contract for the component implementation. In this case, we need only provide an `execute()` method for an entry point. This method receives the `ActionInvocation`, which gives it access to all the relevant information for the current request. As we've learned, this `ActionInvocation` provides access to such things as the action itself, the `ActionContext`, and the `ValueStack`. This is how results get their hands on the data.

Let's dive in and have a look at our `JSONResult`. Listing 8.1 shows the full code of our custom result.

Listing 8.1 Serializing objects from the `ValueStack` into JSON responses

```
public class JSONResult implements Result {          (1)

  public static final String DEFAULT_PARAM = "classAlias";      (2)

  String classAlias;                                  (3)

  public String getClassAlias() {
```

```
    return classAlias;
  }

  public void setClassAlias(String classAlias) {
    this.classAlias = classAlias;
  }

  public void execute(ActionInvocation invocation)
     throws Exception {

    ServletActionContext.getResponse().setContentType("text/plain");
    PrintWriter responseStream =
      ServletActionContext.getResponse().getWriter();

    ValueStack valueStack = invocation.getStack();
    Object jsonModel = valueStack.findValue("jsonModel");

    XStream xstream = new XStream(new JettisonMappedXmlDriver());

    if ( classAlias == null ){
      classAlias = "object";
    }
    xstream.alias(classAlias, jsonModel.getClass() );

    responseStream.println(xstream.toXML( jsonModel ));

  }
}
```

❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿

Admittedly, there might be a few unfamiliar things in this code. But it's short and simple, as you'll soon see. Basically, this result just takes data from the `ValueStack` and writes it to the response stream as JSON. Let's go line by line.

First, we implement the result interface ❶. Before getting to the `execute()` method ❹, which we must implement to fulfill the responsibilities of that interface, we do some stuff to parameterize our result. With a little code, we can allow our results to accept parameters from their XML declarations via the `param` tag. For this result, we want to accept a parameter than gives us a logical name under which our root data object will be serialized to the JSON. We'll explain why in a moment, but for now just observe that we implement a JavaBeans property `classAlias` ❸ and then provide a constant called `DEFAULT_PARAM` ❷ that names our `classAlias` as the default parameter. Each result can define a default parameter, which can be passed in without being named.

Now, to the heart of the matter. Once inside the `execute()` method, we get down to work. Since we're going to return an HTTP response to the client, we use the `ServletActionContext`, a servlet-specific subclass of the `ActionContext`, to set the content type ❺ on the servlet response object and get the output stream ❻ from that same object. This is standard fare for all results that write a response for the client. Next, we use the `ActionInvocation` to retrieve the `ValueStack` and programmatically pull the domain object we want to serialize from it ❼. Note that in this naive, but easy-to-follow, result implementation, we're requiring that the action place the object that it wants to send to the client in a property called `jsonModel`. We could perhaps parameterize this also, but we'll save that for later refactoring.

Point of interest

Default parameters provide another mechanism for supporting intelligent defaults. The result type we've been using with JSPs, the `RequestDispatcher`, takes the location of the JSP page as a default parameter. If you were to consult the source code for the `RequestDispatcher` result, you'd find that it defines location as its default parameter in the same fashion as the `JSONResult` defines the `classAlias` parameter as its default. The benefit of defining a default parameter is that you don't have to name that parameter when you pass it in.

In the following result definition, you can see that we just put the location of the JSP in the body of the `result` element, rather than submitting it via a `param` tag.

```
<result>/chapterEight/VisitorHomePage.jsp</result>
```

This path value is automatically set to the location property of the `RequestDispatcher` result object. It would be valid, but unnecessary, to specify the location property with an explicit `param` tag as follows:

```
<result type="dispatcher">
  <param name="location">/chapterEight/VisitorHomePage.jsp</
   param>
</result>
```

Much more verbose. Thanks to the framework's dedication to intelligent defaults, we can usually avoid such lengthy elements in our declarative architecture documents. Note that we also added the unnecessary `type` attribute here. It's unnecessary because the `RequestDispatcher` is the default result type as long as you inherit from the `struts-default` package.

Now it's time to serialize that object. We're using a couple of open source packages to do this: XStream (http://xstream.codehaus.org) and Jettision (http://jettison.codehaus.org). XStream allows us to serialize Java objects to XML, and the Jettison driver for XStream adds in the JSON part. First, we create an instance of the XStream serializer with the Jettison driver ❽. Then we use our `classAlias` to set the alias for the object we're serializing ❾. If we didn't do this, our JSON would name the object with the fully qualified class name—`manning.utils.User`. It'd be much nicer to have this just show up with a name that's more suitable for our JavaScript code, such as `user` or `artist`. We've parameterized this value so the user can choose his own alias. Finally, we serialize the `jsonModel` to the response output stream ❿.

For the curious, here's what our JSON response looks like:

```
{"artist":{"username":"Mary","password":"max","portfolioName":"Mary's
Portfolio","firstName":"Mary","lastName":"Greene",
"receiveJunkMail":"false"}}
```

Even without a short course in JSON, you should be able to see that this notation defines an associative-array–based object. An *associative array* is something like a normal array, but with values mapped to string keys instead of numerical indexes. In this case,

the JSON defines an object named `artist`. The `artist` object is an associative array containing name-value pairs equivalent to the properties of our Struts 2 Portfolio's user object. Our JavaScript client code can make quick work of instantiating an object from this bit of JSON, as you'll soon see.

And, believe it or not, that's all there is to our custom `JSONResult`. We could make many refinements to this result, but this one works and has remained simple enough to give you a good idea of how results get their hands on the `ValueStack` and other objects they want. With a result that can return JSON to our client, we're now ready to implement some Ajax functionality in our application. The next sections show how we implement our Ajax artist browser that uses this `JSONResult`.

AN AJAX CLIENT

On the client, we use JavaScript Ajax techniques to submit an asynchronous request to our Struts 2 application. Since Ajax is beyond the scope of this book, we'll be brief here. We'd like to point out that our Ajax techniques come directly from another Manning title, *Ajax in Action*. If you'd like to learn more about Ajax, we highly recommend that acclaimed book.

The artist browser client code is defined in the ajaxUserBrowser.jsp page. That page sets up the select box and the information window, as seen in figure 8.3. The select box registers an Ajax JavaScript function, `fetchUser()`, for its `onchange` event. This function, defined in ajaxUserBrowser.js, submits the request. It gets the selected username from the select box, then submits an asynchronous request to the server. All of this can be seen in the following snippet:

```
function fetchUser(){
  console=document.getElementById('console');
  var selectBox = document.getElementById('AjaxRetrieveUser_username');
  var selectedIndex = selectBox.selectedIndex;
  var selectedValue = selectBox.options[selectedIndex].value
  sendRequest("AjaxRetrieveUser.action",
               "username=" + selectedValue , "POST");
}
```

Note that we no longer use the form to submit the request; we use programmatic access to send our request. Nonetheless, we still use a normal Struts 2 action as our target URL. You can dig deeper into the `sendRequest` function if you want to see the specifics of the Ajax stuff, but suffice it to say that it just gets an `XMLHttpRequest` object and uses that object to submit our request. It also, notably, registers a callback function that handles the JSON response when it arrives. Such callback functions are at the core of any Ajax application; they make the dynamic changes to the page when the response returns with the new data. Let's take a look at how our callback handles our JSON response. Here's the `onReadyState` function that handles our server's response:

```
function onReadyState() {
  var ready=req.readyState;
  var jsonObject=null;

  if ( ready == READY_STATE_COMPLETE ){      (1)
```

```
        jsonObject=eval( "("+ req.responseText +")" );      ❷
        toFinalConsole ( jsonObject );      ❸
      }
    }
```

The `XMLHttpRequest` object actually calls this function at several intermediate stages of completion before the response has completely returned. In this case, we're not going to do anything until the response is fully cooked. With this in mind, we test for completion ❶ before starting to process the response. If the response is complete, we retrieve the JSON response text and directly instantiate it as a JavaScript object, the `jsonObject`, by using the built-in JavaScript `eval()` function ❷. With that one line, we now have a JavaScript version of the same object that our action prepared on the Java side. That's undeniably cool. Next, we send this object off to a function that dynamically modifies our blue window to show the new artist's information ❸. Again, if you're interested in the details of our dynamic HTML techniques, you can look at the source code to see how we update the artist info on the fly, but it's off topic for us to explore in this book.

Now that we have an Ajax client in place, it's time to build the action that processes the request. As you'll see, it's no different from the other actions we've been working with throughout the book.

THE ACTION

How hard is it to write an Ajax action? There's nothing to it. In fact, it's no different than a normal action. One of the good things about the clean MVC of Struts 2 is that our action won't have to know anything about what's happening in the result. Our actions continue to function in the same old mode of receiving data from request parameters, executing some business logic, preparing the final data objects, and then handing everything over to the result. In this case, our action receives a username from the Ajax artist browser's select box, retrieves a full user object, and puts it on a JavaBeans property where the result can find it. For our Ajax example, we wire this action to a result of the `JSONResult` type so that the user object gets serialized to JSON. But we could just as easily use this same action in non-Ajax settings if we like; we'd just have to wire it to a JSP or something. The action doesn't care whether the result serializes the user object to JSON or reads properties off of it with JSP-based Struts 2 tags.

Just to prove we've got nothing up our sleeves, we look at the `RetrieveUser` action class to confirm that we're still working with a standard Struts 2 action. Listing 8.2 shows the full code from RetrieveUser.java.

Listing 8.2 `RetrieveUser` retrieves a user and sets the `jsonModel` property

```
public class RetrieveUser extends ActionSupport  {

  public String execute(){

    User user = getPortfolioService().getUser( getUsername() );      ❶
    setJsonModel(user);      ❷

    return SUCCESS;
  }
```

```
    private String username;
    private Object jsonModel;

      . . .

      JavaBeans Property Implementations Omitted for Brevity

      . . .
}
```

❸

Again, this looks just like any other action we've been working with. It doesn't matter that the request came from a JavaScript `XMLHttpRequest` object or that the response is going to be a JSON serialization. Incoming and outgoing data is still carried on JavaBeans properties ❸. The `execute()` method still conducts our business logic, consisting of a call to our service object's `getUser()` method ❶, and then exposes the resulting data object, the `user`, by setting it on a local JavaBeans property ❷. We then return the result string to indicate which result we want to fire. If that result is of type `JSONResult`, it pulls the user off of the `ValueStack`, serializes it in JSON, and sends it back to the client.

The cool thing is that this action could be reused with a normal full-page JSP result without modifying a line of code. Let's say we had a JSP page that would rerender the entire page with the updated `User` information. We could just wire it to this same action and it would read the `User` information off of the `ValueStack`, with Struts 2 tags, as it built the page. Both the JSON result and the JSP result can easily work with the `User` object on the `ValueStack`. When the action prepares the `User`, it doesn't care who does what with that data. Any result could be wired to this action. This can be helpful if you're, say, migrating an existing Struts 2 web application from classic full-page HTTP cycles to a more Ajax-based application.

DECLARING AND USING THE JSONRESULT TYPE

Now that you've seen all the parts, let's look at how we wire it all together for the Struts 2 Portfolio. We've already been through the process of declaring actions and their results, but in this case we've created a whole new result type, and that requires more work. Previously, we've only worked with results that come predeclared as a part of the `struts-default` package. Obviously, our `JSONResult` hasn't been declared in that default package. So, we need to declare our `JSONResult` as an available result type in our package. We do this with the following declaration from our chapterEight.xml document:

```
<result-types>
  <result-type name="customJSON" class="manning.chapterEight.JSONResult" />
</result-types>
```

This is simple. We use the `result-type` element, which must be contained in the `result-types` element, to map a logical name to our implementation class. With this in place, we can use this result type anywhere in our package or in any package that extends our package. We do just that in the declaration of our `AjaxRetrieveUser` action, which is the action that responds to our Ajax request. Here's the declaration of that action:

```
<action name="AjaxRetrieveUser"
    class="manning.chapterEight.RetrieveUser">
  <result type="customJSON">artist</result>
</action>
```

Since the custom JSON result type is not the default result type for this package, we must specify it with the `type` attribute. If we were going to do a lot of Ajax work, we'd probably make a separate package for those requests and declare the JSON result as our default type. Next, see that we've passed in `artist` as our default parameter, which is set to our result's `classAlias` property. Earlier, we saw how this was used as the name of the JSON serialization of our object.

That covers all the pieces of our Ajax artist browser. With this mapping in place, our Ajax request, submitted by our JavaScript client-side application, will come into the framework and hit the `AjaxRetrieveUser` action. Looking back to figures 8.1 and 8.2, you can see that the framework handles this Ajax submission no differently from normal requests. The `AjaxRetrieveUser` action prepares the `User` data, puts it on the `ValueStack`, and hands processing of the response over to the `customJSON` result. That result serializes our user object into JSON and sends it back to the client. On the client, a callback method receives the JSON, makes a JavaScript object out of it, and passes that object to a method that dynamically updates the page to show the new user information. That's one way to do Ajax with Struts 2.

Ajax Tips

As you might suspect from a newly minted web application framework, there's a bit of attention being paid to Ajax. Using a custom result to return an Ajax-suitable response is one good way to build a Struts 2 Ajax application. JSON is definitely not the only option for implementing Ajax. If you wanted to use something else, such as XML, you could easily roll your own result type for returning just about anything.

We should point out that there's already a plug-in to the framework that provides a more robust version of a JSON result type. You can find this and other plug-ins in the Struts 2 plug-in repository at http://cwiki.apache.org/S2PLUGINS/home.html. We recommend you test-drive that plug-in if you want to do something along the lines we've demonstrated in this chapter to integrate Ajax into your Struts 2 application.

Additionally, there's an Ajax theme for the Struts 2 tag API. At the time of writing, this tag library is too beta for us to document. However, we highly recommend visiting the Struts 2 website to check on the status of this exciting project.

By now, you can probably see the flexibility the result component adds to the Struts 2 framework. The separation is so clean that the same set of actions can be wired to many different kinds of result types. Additionally, the result type itself is flexible enough to support any kind of result that you could dream up. It has all of the resources of the framework, such as the all-important `ValueStack`, at its disposal.

Now that you have a feel for what results can do, it's time to peruse the various types that come with the framework. The offering is rich; we'll start by looking at the most commonly used types.

8.2 Commonly used result types

In this section, we cover the usage of the most common result types. The framework comes with quite a few built-in result types, and we invite you to peruse the whole set of them on the Struts 2 website: http://struts.apache.org/2.x/docs/result-types.html. In this section, we show you how to use the ones we think will cover 90 percent of your development needs. The results shown in table 8.1 cover the most common use cases of a classic web application. We cover each of these thoroughly in this section.

Table 8.1 The most commonly used built-in result types

Result type	Use case	Parameters
`dispatcher`	JSPs, other web application resources such as servlets	`location` (default), `parse`
`redirect`	Tell browser to redirect to another URL	`location` (default), `parse`
`redirectAction`	Tell browser to redirect to another Struts action	`actionName` (default), `namespace`, arbitrary parameters that will become querystring params

Note that all these built-in result types are defined in struts-default.xml. They're only built in if you extend the `struts-default` package. We assume that you'll do this. A result-type declaration maps a logical name to a class implementation. The following snippet shows the declaration of the `FreemarkerResult` from struts-default.xml:

```
<result-type name="freemarker"
  class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
```

In all packages that extend the `struts-default` package, we can simply refer to the `FreemarkerResult` by the more convenient name `freemarker`. Throughout our following coverage of the various built-in result types, we'll indicate both the formal class name and the logical name to which that class is mapped: for example, the `FreemarkerResult`, a.k.a. `freemarker`. You'll see what we mean. Incidentally, we cover the `FreemarkerResult` in section 8.3.

8.2.1 The RequestDispatcher, a.k.a. dispatcher

The short story is that you use this result when you want to render a JSP page as the result of your action. For the normal Struts 2 workflow, this understanding of the dispatcher result type will serve you well enough. By normal Struts 2 workflow, we mean that a request comes in to an action, that action processes the request and hands off to a result that writes the response back to the client. In this routine use case, using the `RequestDispatcher` result is easy.

The fact that we've been using this result type in all of our sample code, without mentioning it, speaks to the ease with which you can use it. So far, we haven't even had to specify this type when declaring our results. This is because when you extend the `struts-default` package, you inherit the `DispatcherResult` as the default type. One result type per package is allowed to claim itself as the default type. Here's the snippet from struts-default.xml that specifies the `dispatcher` as the default type of the `struts-default` package:

```
<result-type name="dispatcher"
  class=" . . . ServletDispatcherResult" default="true"/>
```

First, note that we've omitted the full package name of this class due to space concerns. With this declaration in place, we can write simple result elements like the following if we are using JSPs:

```
<action name="PortfolioHomePage" class=". . . chapEight.PortfolioHomePage">
  <result>/chapterEight/PortfolioHomePage.jsp</result>
</action>
```

Using JSPs in the common use case is a no-brainer; for the most part, we can keep our gray matter out of this. However, there's a fair chunk of logic occurring, or potentially occurring, within this innocent-looking result. At the least, you might be wondering why it's called the `DispatcherResult`. And, eventually, you'll probably need to know how to use this result in a nonstandard Struts 2 workflow. With that in mind, we present the following account of what a dispatcher result does. Warning: it helps to understand the Servlet API. If you need a primer, we still recommend reading the Servlet Specification; it's the shortest path to enlightenment.

THE SERVLET HEART OF THE DISPATCHER RESULT

At the core of the `DispatcherResult` result type is a `javax.servlet.RequestDispatcher`. This object, from the Servlet API, provides the functionality that allows one servlet to hand processing over to another resource of the web application. Usually this is a servlet, and this servlet is commonly a JSP page. The handoff must be to another servlet in the same web application as the first servlet. The `RequestDispatcher` exposes two methods for handing execution over to the other servlet: `include()` and `forward()`. These two methods determine how much control over the response will be given to the secondary resource.

A temporary handoff is done via a call to the `include()` method. This means that the first servlet has already started writing its own response to the client but wants to include the output from the second servlet in that response. A permanent handoff is done via a call to the `forward()` method. In this case, the first servlet must not have sent any response to the client before the call is made. In a forward, the servlet is delegating the complete response rendering to the second servlet.

These two dispatch methods are quite different from each other, but they do share some things in common. Most importantly, they're both distinguished from an HTTP redirection. Whereas a redirection sends the browser a redirect response telling it to make another request to a different URL, the dispatcher methods don't send the

browser any response at all. The transfer of control, from one resource to another, is completely within the server. In fact, it's even in the same thread of execution. Another important detail is that the `include()` and `forward()` methods both pass the original request and response objects to the new servlet. This important detail allows the second servlet to essentially process the same request.

NORMAL WORKFLOW: DISPATCHING AS A FORWARD()

Now let's get back to Struts 2. Let's consider the normal Struts 2 workflow from a Servlet perspective. By normal workflow, we mean the simple case where an action receives the request, processes the business logic, prepares the data, then asks a result to render the response. This case has no indirection, no slight of hand, just an action and a result. We start by examining the case with which we're already familiar, the JSP result.

First of all, note that the framework is itself actually just a servlet. Well, a servlet filter, but that's a fine point not worth quibbling over for the moment. When the framework first processes a request, we can consider it as the work of the *primary* servlet. Thus, when your initial action is executing, it's executing as the primary servlet. When it's finished, the action selects a JSP result to render the view. Since JSP pages are actually servlets, this transfer of execution from the action to the JSP page is a transfer of control from the first servlet to a second. This normal case is done with the `RequestDispatcher`'s `forward()` method.

This implies that the Struts 2 action, processing under the control of the first servlet, obviously can't have written anything to the response yet. If it had, we'd be limited to an `include()` call. By forwarding to the result JSP servlet, the action gives full control over writing the response to that JSP. This is the low-level expression of the framework's MVC separation of concerns, in case you're interested in that sort of thing. Note that the JSP has access to all of the data from the action via the `ThreadLocal ActionContext`, and its most important citizen, the `ValueStack`; this access to the `ActionContext` depends on the fact that the dispatching of requests, with `forward()` or `include()`, always occurs on the same thread of execution.

FORWARDING TO ANOTHER SERVLET

But you don't have to limit yourself to JSPs. You can point a `RequestDispatcher` result at any resource of a web application. By web application, we're talking about the Servlet API concept. In addition to a JSP, it could be another servlet or even a static resource of some kind that can be served by your web application. In the chapter 8 version of the Struts 2 Portfolio, we've built a demonstration of handing off to another servlet in the web application. On the chapter 8 home page, enter your favorite color in the field shown in figure 8.4.

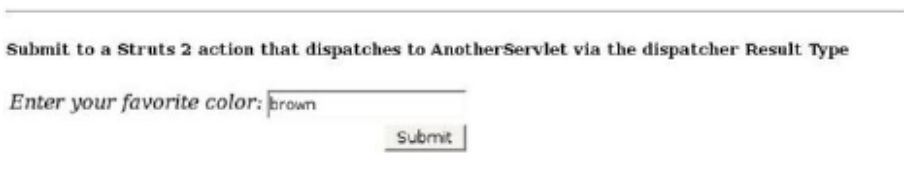

Figure 8.4 A simple form that submits to a Struts 2 action that uses another servlet to render the result.

When you submit your form, the `ForwardToAnotherServlet` Struts 2 action does some simple processing and then turns over the rendering of the result, a.k.a. the servlet response, to another servlet in our web application. Listing 8.3 shows our `AnotherServlet` class, the servlet that renders the response for our action.

Listing 8.3 `AnotherServlet` renders a response, pulling data from various locations

```
public class AnotherServlet extends HttpServlet {                    (1)

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

    response.setContentType( "text/html");                            (2)
    response.getOutputStream().println( "<html>");
    response.getOutputStream().println( "<head>");
    response.getOutputStream().println( "</head>");
    response.getOutputStream().println( "<body>");
    response.getOutputStream().println("<p>Hello from
      anotherServlet's doPost()</p>");
    response.getOutputStream().println("<p>Attribute set in the
      struts 2 action = " +
      request.getAttribute("attributeSetInS2Action" ) + "</p>");      (3)
    response.getOutputStream().println("<p>Favorite color from
      request parameters = " + request.getParameter("favoriteColor")
      + "</p>");                                                      (4)
    String propertyFromAction = (String)

   ActionContext.getContext().getValueStack().findValue("testProperty");
    response.getOutputStream().println("<p>Value retrieved from action
      property on ValueStack = " + propertyFromAction + "</p>");      (5)
    response.getOutputStream().println( "</body>");
    response.getOutputStream().println( "</html>");
  }
}
```

`AnotherServlet` extends `HttpServlet` (1) to become a servlet. Then it overrides the `doPost()` method. As you can see, the `AnotherServlet` servlet writes a fully structured HTML page (2) to the response's output stream. It even sets the content-type header for the HTTP response. This is what we mean when we say that a forward gives full control of the response rendering over to the other servlet. Next, the servlet begins writing to the response stream.

`AnotherServlet` pulls data from several sources while rendering the response just to demonstrate the various methods for passing data. First, we pull a named `request` attribute (3) into the rendering. This attribute will have been set in our Struts 2 action. This demonstrates passing data via the Servlet API objects, the request map in this case. Next, we pull the request parameter (4), the favorite color that we entered in the form that was submitted. This also comes to us via the Servlet API's request object.

But what about the Struts 2 data? Next, just to prove that the `ActionContext`'s `ThreadLocal` nature does indeed make all of the Struts 2 stuff available to our dispatched servlet, we pull some values, programmatically, from the `ValueStack` itself (5). As it turns out, this value from the `ValueStack` was a property we set on our action

during its execution. This method of retrieving values from the `ValueStack`, from the secondary servlet, approximates the details of how our Struts 2 JSP tags have been accessing our `ValueStack` values. In this case, we had to go with programmatic access, but that's what the tag implementations do themselves.

Now let's take a look at the `ForwardToAnotherServlet` action to see how all of this data was put in place prior to the dispatch. Listing 8.4 shows the complete code from FrowardToAnotherServlet.java.

Listing 8.4 Putting data on a property and in the request map

```
public class ForwardToAnotherServlet extends ActionSupport implements
  RequestAware{

  public String execute(){
    getRequest().put( "attributeSetInS2Action", "Hello from
    a request attribute set in the S2 Action");                    (1)
      return SUCCESS;
    }

    private Map request;

    public void setRequest(Map request) {
      this.request = request;
    }
    public Map getRequest (){
      return request;
    }

    public String getTestProperty(){                              (2)
      return "myValueFromActionProperty";
    }
}
```

Not too much new here. We just want to follow the data flow to demonstrate how two servlets, linked by the `RequestDispatcher`, share in the processing of a single request. First, we see that our action uses the `RequestAware` interface to have the `request` map injected. It then uses the `request` map to set the `attributeSetInS2Action` attribute (1), which `AnotherServlet` will retrieve, as we saw in listing 8.3. We also see that this action exposes the `testProperty` JavaBeans property (2), which `AnotherServlet` pulls off the `ValueStack` with programmatic access.

As for wiring our action to our result rendering servlet, it works just like a JSP:

```
<action name="ForwardToAnotherServlet"
    class="manning.chapterEight.ForwardToAnotherServlet">
  <result>/anotherServlet</result>
</action>
```

We put the path to the servlet, relative to our web application, just as we've been putting the paths to our JSPs. Note that this all assumes we've declared `AnotherServlet` with a servlet mapping in our web application's web.xml file. Take a look at that file, in the source code, if you want to see how that's done.

We should sum up the point of this example. The `AnotherServlet` example renders an HTML page result. If you have the choice, you should never do this. This is

what JSPs are for. However, this example will prove useful as a demonstration of accessing data from another servlet if you ever find yourself called to integrate Struts 2 with an external servlet. This is something that happens fairly often. As this example demonstrates, it's easy to pass data from the framework to external web application resources, such as `AnotherServlet`.

Now we've seen two normal workflow examples of an initial action passing control over to a servlet, or JSP, for rendering its response. In these normal cases, we're still in the realm of the `forward()` method of the `RequestDispatcher`. Let's now look at some cases where the `include()` method is used.

DISPATCHING AS AN INCLUDE()

It's quite important to know when the `dispatcher` will use a `forward()` and when it'll use an `include()`. The main reason is because included material must make sense as a fragment, since it's dumped midstream into the primary response. Our previous servlet example, which wrote its own `<html>`, `<head>`, and `<body>` tags, would most likely not be a suitable target for inclusion, as the initial servlet might already have written these elements. So when will we see an include?

The most common case is the use of the Struts 2 `action` tag. This tag, which we've already seen in chapter 6, invokes the execution of another action from the execution of a JSP page. All actions invoked via the `action` tag will be includes. This means the result of the action, if it renders, must be a valid fragment of the first result. If you want more details on this case, please refer back to our `action` tag examples in chapter 6.

SETTING UP A REQUESTDISPATCHER RESULT

Configuring a `dispatcher` result is simple. First, the `dispatcher` result must be defined as a result type. As with all of the built-in result types, this is already done in the `struts-default` package, which your packages will typically extend. In the case of the `dispatcher` result type, it was configured as the default result type. The following snippet shows the configuration of the `dispatcher` result type from the struts-default.xml file:

```
<result-type name="dispatcher"
   class="org.apache.struts2.dispatcher.ServletDispatcherResult"
   default="true"/>
```

The `result-type` element maps a logical name to a Java class that provides an implementation of the `Result` interface. These declarations are made on the package level and can be inherited. This is just as we've seen with interceptors. Since our Struts 2 Portfolio packages extend the `struts-default` package, we can use this result type in our applications. We just need to reference it by its name, as follows:

```
<action name="SelectPortfolio"
  class="manning.chapterSeven.SelectPortfolio">
    <result type="dispatcher" >
      /chapterSeven/SelectPortfolio.jsp
    </result>
</action>
```

This `action` declaration, from our chapterSeven.xml file, doesn't actually specify the `type` attribute of the result element like this. We've added it only for this listing. We can omit it in the actual application because the declaration of the `dispatcher` result type sets its default attribute to `true`.

Like many of the result types, it's possible to parameterize the `dispatcher` result type in a given instance. The `dispatcher` has two parameters, but they're seldom used, as they both provide strong default values. The first parameter is the `location` parameter. This parameter gives the location of the servlet resource to which we should dispatch the request. The second parameter is the `parse` parameter. This parameter determines whether the location string will be parsed for OGNL expressions. Here's the previously seen action declaration rewritten to explicitly use these parameters:

```
<action name="SelectPortfolio"
    class="manning.chapterSeven.SelectPortfolio">
  <result type="dispatcher" >
    <param name="location">/chapterSeven/SelectPortfolio.jsp</param>
    <param name="parse">true</param>
  </result>
</action>
```

You probably won't use these parameters too often because the default settings are so sensible. We show them here to demonstrate how to parameterize your results. In this case, the `location` parameter is what's known as the default parameter of the `result` element. Since `location` is the default parameter, you can omit the `param` tag, as we usually do, and pass the path to your JSP in as the text content of your `result` element. As for the `parse` parameter, it's true by default. If you ever feel the need to squelch parsing of OGNL in your location string, you can use this parameter.

Why would you want to put OGNL in your location string? The short answer is to make it dynamic. For instance, you could dynamically build querystring parameters. In the case of the dispatcher result, dynamically inserting data doesn't make a lot of sense. All the data that you might insert into the querystring will be available to your result page anyway, via OGNL, tags, and so on. We demonstrate how to take advantage of OGNL parsing in the location string in the next section, which covers redirect results.

Rather than returning a page to the client yourself, you'll sometimes want to redirect the client to another web resource. The next two results, from table 8.1, provide ways of redirecting to other resources. We start with the standard redirect result.

8.2.2 The ServletRedirectResult, a.k.a. redirect

Though the `RequestDispatcher` result can hand over processing to another resource, and is commonly the right tool for the job, you'll sometimes want to use a redirect to hand control to that other resource. What's the difference? The defining characteristic of the `RequestDispatcher` is that the handoff is completely on the server side. Everything happens within the Servlet API and on the same thread. This means that all the data from the first resource is available to the second resource via both the Servlet API and the Struts 2 `ActionContext`. We saw this clearly in our examples from

the `RequestDispatcher` section. This is critical if your second servlet expects to have full access to all the data.

A redirect, on the other hand, considers the current request finished and issues an HTTP redirect response that tells the browser to point to a new location. This act officially closes the current request from both the Struts 2 `ActionContext` perspective and the Servlet request perspective. In other words, all server-side request- and action-related state is gone. While it's still possible to persist some of the current request's data over to the second resource, the techniques for doing so are inefficient. We'll show how to do this, but if you really need to carry much data over to the second resource, you should probably just use the `RequestDispatcher`.

NOTE If you need to persist data from the initial request to the resource that's the target of your redirect, you have two choices. The first choice is to persist the data in querystring parameters that are dynamically populated with values from the `ValueStack`. You can embed OGNL in the `location` parameter to do this. We'll demonstrate this technique when we discuss the `ActionRedirect` result, a special redirect result to be used when redirecting to other Struts 2 actions.

The second option is to persist data to a session-scoped map. This has a couple of drawbacks. First, it only works when the secondary resource belongs to the same web application. Second, best practices warn against unnecessary use of the session scope as a data storage.

The most common reason for using a redirect arises from the need to change the URL shown in the browser. When a redirect is issued, the browser handles the response by making a new request to the URL given in the redirect. The URL to which the redirect is issued replaces the previous URL in the browser. The fact that the new URL replaces the old URL in the browser makes the redirect particularly useful when you don't want the user to resubmit the previous URL by clicking the Reload button. Or, for that matter, any time you just want to change the URL in the browser.

SETTING UP A REDIRECT RESULT

As with all Struts 2 components, the `redirect` result must be declared and mapped to a logical name. And as with all built-in components, this occurs in the struts-default.xml document. The following snippet shows the declaration of this result type from that file:

```
<result-type name="redirect"
class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
```

Nothing unusual here. Note that this result isn't set to default as the `RequestDispatcher` declaration was. This just means that we need to explicitly specify our `type` attribute when we use this result type. The following snippet shows how we might configure a redirect result that would tell the browser to go to another URL:

```
<action name="SendUserToSearchEngineAction" class="myActionClass">
    <result type='redirect'>http://www.google.com</result>
</action>
```

To redirect to another resource, we just need to specify a full URL, or a relative one if the redirect is to another resource in our web application, and name the redirect type explicitly. Here we redirect to the omnipotent Google search engine. Note again that the name given to the `type` attribute matches the `name` attribute of the `result-type` element from the `struts-default` package.

The redirect result supports two parameters: `location` and `parse`. These are the same parameters supported by the `RequestDispatcher`. As you can guess, the `location` parameter is the default parameter, so we don't have to specify it by name when we give the URL for the location in the body of the `result` element. And `parse` is `true` by default, so unless you have reason to turn off the framework's parsing of the location value for OGNL expressions, you won't be handling that parameter much either. Since the use of embedded OGNL to pass along querystring params to the secondary resource is particularly helpful with redirect results, we'll explore this technique now. But keep in mind that you can use embedded OGNL with any of the result types that support it.

EMBEDDING OGNL TO CREATE DYNAMIC LOCATIONS

We should now look at an example of how to embed an OGNL expression in the `location` parameter value in order to pass some data forward to the second action. Note that this method of embedding OGNL to create dynamic parameter values can be used in any of the previous result types that support the `parse` parameter. The following example shows how we could pull a value from the `ValueStack`, at runtime, and pass that value as a querystring parameter to the same URL we used in the previous example:

```
<action name="SendUserToSearchEngineAction" class="myActionClass">
  <result type='redirect' >
    http://www.google.com/?myParam=${defaultUsername}
  </result>
</action>
```

First of all, take careful notice of the fine point of the dollar sign. We've been using the percent sign throughout the book for our OGNL escape sequence, but in the context of our struts.xml, and other XML documents used for the declarative architecture, we must use a `$` instead. Other than this inconsistency, the OGNL access still works the same. The OGNL looks up the property on the `ValueStack`. Provided that the imaginary `myActionClass` did something to make the `defaultUsername` property appear on the `ValueStack`, such as exposing it as a JavaBeans property, this result would render into a redirect response that'd point the browser to the following URL (assuming that the `defaultUsername` property, on the `ValueStack`, held the value of `Mary`): http://www.google.com/?myParam=Mary

Of course, this URL will get you nowhere. We show a real example of using this kind of embedded OGNL when we discuss the `redirectAction` result in the next section. For now, just note that you can use OGNL to build dynamic parameter values by pulling values from the `ValueStack`. You can do this to all the parameters that a result takes, as long as the result type itself supports the parsing of OGNL.

8.2.3 The ServletActionRedirectResult, a.k.a. redirectAction

The `redirectAction` result does the same thing as the plain redirect result, with one important difference. This version of `redirect` can understand the logical names of the Struts 2 actions as defined in your declarative architecture. This means that you don't have to embed real URLs in your result declarations. Instead you can feed the `redirectAction` names and namespaces from your action and package declarations. This makes your declarations more robust in the face of changes to URL patterns. As an example of such a URL pattern change, let's say you wanted to change the action extension from .action to .go. If you'd used the plain redirect result extensively to target Struts 2 actions, then you'd have a lot of hard-coded URLs to adjust.

As an example, let's look at the `Login` action mapping from our earlier versions of the Struts 2 Portfolio application. This mapping, as seen in the following snippet, uses the plain `redirect` result:

```
<action name="Login" class="manning.chapterSeven.Login">
  <result type="redirect">
    /chapterSeven/secure/AdminPortfolio.action
  </result>
  <result name="input">/chapterSeven/Login.jsp</result>
</action>
```

As you can see, we have a real URL embedded in our declarative architecture. This would have to be manually corrected in the face of our hypothetical action extension change. The following snippet, from the chapter 8 version of the application, shows how we can do the same thing with the `redirectAction` instead:

```
<action name="Login" class="manning.chapterEight.Login">
  <result type="redirectAction">
    <param name="actionName">AdminPortfolio</param>
    <param name="namespace">/chapterEight/secure</param>
  </result>
  <result name="input">/chapterEight/Login.jsp</result>
</action>
```

Functionally, this is no different than the previous version. They both create a redirect response that points to the following URL: http://localhost:8080/manningSampleApp/chapterEight/secure/AdminPortfolio.action

The difference is that the `redirectAction` result would stand up to a variety of changes in the URL pattern handling, such as the action extension we mentioned earlier.

There is one other goodie supported by the `redirectAction` result. We can easily configure request parameters to be passed on to the target action. With the plain redirect, we had to write the querystring parameter out by hand. Now we can use the `param` tag. In this case, we give the parameter whatever name and value we like. These arbitrary name-value pairs are appended as querystring parameters to the generated URL. Take the following real example from our chapter 8 version of the Struts 2 Portfolio's `Login` action:

```
<action name="Login" class="manning.chapterEight.Login">
  <result type="redirectAction">
    <param name="actionName">AdminPortfolio</param>
    <param name="namespace">/chapterEight/secure</param>
    <param name="param1">hardCodedValue</param>          (1)
    <param name="param2">${testProperty}</param>        (2)
  </result>
  <result name="input">/chapterEight/Login.jsp</result>
</action>
```

This result configuration is no different than the previous except for the addition of two request parameters. First, we hard-code a value for `param1` (1). Then, we use our previously learned OGNL embedding skills to dynamically pass a second parameter (2). Here's the new URL to which our result will redirect the browser, complete with our querystring parameters: http://localhost:8080/manningSampleApp/chapterEight/secure/AdminPortfolio.action?param1=hardCodedValue&param2=777

As you can see, our `Login` action must have done something to make the `testProperty` property exist on the `ValueStack` and hold the value of `777`. As it turns out, our `AdminPortfolio` action won't do anything with these values except write them to the result page to prove they went through. But this should be adequate to demonstrate how one goes about passing dynamic values into result parameter values with embedded OGNL. Again, any result that supports a `parse` parameter supports these kinds of techniques.

With that, we're finished with the most commonly used result types. Some developers will rarely need anything more. But while JSPs still retain their preeminence, many teams are choosing Velocity or FreeMarker to render their response pages. In the next section, we look at the results that support using Velocity or FreeMarker templates instead of JSPs.

8.3 JSP alternatives

In this section, we a look at two result types that support alternative view-layer technologies: FreeMarker and Velocity. While using these is as straightforward as using the `dispatcher` result for JSPs, we do want to give clear demonstrations of using each. For each of these alternative technologies, we'll rewrite one of the pages that we've already done with JSPs and the `dispatcher` result.

In the past, some energy has been spent debating whether templating engines provide better performance than JSPs. At this point, any performance difference doesn't seem quite large enough to warrant making a view-layer technology decision on that basis alone. For mission-critical performance, you might want to investigate the performance issues in the context of the most recent versions of the common web application servers. The most obvious performance issues center around the fact that both Velocity and FreeMarker results write directly to the original response stream of the original request. In other words, the `RequestDispatcher` and all of the Servlet overhead implied therein aren't involved. This and other performance issues are well documented on the

Web, and we suggest you consult the most recent e-opinions if you need to hit maximum levels of performance.

8.3.1 VelocityResult, a.k.a. velocity

Velocity templates are a lightweight and well-proven technology for mixing dynamic data into the rendering of view-layer pages. Using the Struts 2 tag API in Velocity templates, even for the Velocity novice, poses a gentle learning curve. Besides the quick bit about learning the syntactical differences of how the tags are written, the most sophisticated portion is still the OGNL used by the tags to reference the data in the `ActionContext` and the `ValueStack`. But, as we've indicated, the OGNL works the same as in all the JSP-based examples we've given throughout the book.

In passing, we note that the `velocity` result type also exposes the Struts 2 data to the native Velocity expression language capabilities. This means that you could forgo using the Struts 2 tag API and still access values on the `ValueStack` or in the `ActionContext`. We won't cover the details of this access for a couple of reasons. First of all, we really believe that the Struts 2 tag API represents the best way to render your pages. Second, the manner in which the data objects are exposed to the Velocity EL is not tightly synchronized with the exposure to the OGNL used in the Struts 2 tags. This can be confusing when trying to learn the OGNL. If you find that you want to use the native expression language capabilities of Velocity to access the data exposed by the Velocity result, please consult the Struts 2 website at http://struts.apache.org/2.x/docs/velocity.html.

Now let's see how to get set up and start using Velocity templates as your view-layer technology. To demonstrate this, we'll reimplement the Struts 2 Portfolio's `ViewPortfolio` action.

USING VELOCITY RESULTS

The first thing you need to do is make sure that you have the Velocity JAR files in your application. At the time of writing, the Struts 2 distribution doesn't come with the Velocity JAR files. It does come with a built-in `velocity` result type, defined in the following snippet from struts-default.xml:

```
<result-type name="velocity"
  class="org.apache.struts2.dispatcher.VelocityResult"/>
```

This is just like all declarations we've seen. With this in place, we can use the logical name to specify Velocity as the type for our results. Note that many applications that choose to use Velocity templates as their view-layer technology will want to go ahead and declare the `velocity` result type as the default result for their packages. We aren't doing this for the Struts 2 Portfolio application, but if you want to do so you can. Just add a redeclaration of the `velocity` result to your package's `result-types` element with the `default` attribute set to `true`, as in the following snippet:

```
<result-types>
  <result-type name="velocity"
    class="org.apache.struts2.dispatcher.VelocityResult" default="true"/>
</result-types>
```

Now let's take a look at our Velocity version of the `ViewPortfolio` action. If you go to the visitor's page of the chapter 8 version of the application, you'll find three versions of the `ViewPortfolio` page, one each for JSPs, Velocity, and FreeMarker templates. If you test them out, you'll see that they all work exactly the same. Even the XML declarations themselves look very similar. Here's how we declare our `velocity` result version of the `ViewPortfolio` action:

```
<action name="ViewPortfolioVM" class="...ViewPortfolio" >
  <result type="velocity">/chapterEight/ViewPortfolio.vm</result>
</action>
```

The only difference here is that the type of the result is specified as the `velocity` result type defined in the `struts-default` package. The default parameter is still the `location` parameter. Listing 8.5 shows the Velocity template that renders the result.

Listing 8.5 Using Velocity templates to reimplement the `ViewPortfolio` page

```
<html>
  <head>
    <title>Viewing Portfolio</title>
  </head>
  <body>

    <h5>
      This is the #sproperty ("value=portfolioName") portfolio of the
      artist currently known as #sproperty ("value=username")
    </h5>
    <a href='#surl ("action=PortfolioHomePage")'>Home</a>
  </body>
</html>
```

We've already covered the syntax differences of the Struts 2 tags as applied across the various view-layer technologies. As you can see, we're using the Struts 2 property tags and, while the syntax is a bit different, the OGNL in them is the same. It still pulls data from the `ValueStack` just as when using the `property` tag from a JSP. You might also note some fundamental differences between JSP and Velocity, such as the lack of directives at the top of the page. As we said, Velocity is simpler than JSP in some ways. If you want to learn more about Velocity, check out their excellent documentation on the Web at http://velocity.apache.org.

That's about all there is to using the `velocity` result type. As we indicated at the start of this section, the `velocity` result also exposes all of the Struts 2 data to the native Velocity expression language. In some cases, such as in the reuse of existing Velocity templates, that might prove useful. Note also that the `velocity` result type supports the same embedded OGNL, and `parse` parameter, that we saw used with the earlier `dispatcher` and `redirect` result types.

Now, let's see how to use FreeMarker results.

8.3.2 FreemarkerResult, a.k.a. freemarker

Another built-in result type makes it possible to easily transition to FreeMarker as your choice of page-rendering technology. When compared to JSPs, FreeMarker templates

feature the same potential performance benefits as Velocity templates. Again, this mainly arises from the fact that the FreeMarker templates are written directly to the original request's response stream. Many developers wax poetic over the rich features of FreeMarker. And, as we've learned, Struts 2 uses it internally to render the UI components. It's clearly a hot technology for the view layer.

As with Velocity, we don't have the space to give a tutorial on FreeMarker templates, but you don't really need to know too much about FreeMarker to get started. In particular, using the Struts 2 tags from FreeMarker should be a snap. If you need to know more about FreeMarker in general, check out the excellent documentation on the FreeMarker website: http://www.freemarker.org. Again, like the `velocity` result, this result exposes much of the Struts 2 data to the native FreeMarker expression language. And again, we won't cover those details here because we think it can needlessly confuse our coverage of the Struts 2 tag API. However, all the details can be found on the Struts 2 site: http://struts.apache.org/2.x/docs/freemarker.html.

We now reimplement the same `ViewPortfolio` action, this time using a FreeMarker result to render the view.

USING FREEMARKER RESULTS

You won't have to add any JAR files to use FreeMarker. Since the framework uses FreeMarker itself, those resources are already included in the distribution. And the result itself is already declared in struts-default.xml with the following line:

```
<result-type name="freemarker"
    class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
```

As long as you extend the `struts-default` package, this result type is available for your use. All you have to do is specify the logical name `freemarker` in your result's `type` attribute. Again, if you plan to make FreeMarker your primary view-layer technology, you might as well make it the default result type for your packages by adding a line to your package's `result-type` element, as in the following snippet:

```
<result-types>
  <result-type name="freemarker"
    class="org.apache.struts2.views.freemarker.FreemarkerResult"
    default="true"/>
</result-types>
```

Now, let's take a look at how we set up the `ViewPortfolio` action to use the `freemarker` result. The following snippet, from our chapterEight.xml, shows the declaration of the FreeMarker version of the `ViewPortfolio` action:

```
<action name="ViewPortfolioFM"
    class="manning.chapterEight.ViewPortfolio" >
  <result type="freemarker">/chapterEight/ViewPortfolio.ftl</result>
</action>
```

Again, we point the result's `type` attribute to FreeMarker, and the `location` parameter at the FreeMarker template itself. As with the Velocity result, you can use OGNL embedding to pass dynamic values into your parameter values. Or you can turn this

parsing off with the parse parameter. Listing 8.6 shows the ViewPortfolio template in full.

Listing 8.6 Using FreeMarker templates to reimplement the ViewPortfolio action

```
<html>
  <head>
    <title>Viewing Portfolio</title>
  </head>
  <body>
    <h5>
      This is the <@s.property value="portfolioName" /> portfolio of
      the artist currently known as <@s.property value="username" />
    </h5>
    <a href="<@s.url action='PortfolioHomePage'/>">Home</a>
  </body>
</html>
```

As you can see, it's quite intuitive. The syntax for the Struts 2 tag API is a bit different, as documented earlier in this book, but it still uses the same OGNL to access data on the same ValueStack. As promised, the Struts 2 tag API makes switching between view-layer technologies a snap.

That completes our tour of the most commonly used built-in result types. We have one more topic to hit before wrapping up our coverage of results. Up until now, we've just shown how to declare a result locally to a single action. Sometimes, however, you might want to reuse a single result across many actions. In the next section, we show how to declare results that can be used with all the actions in a package.

8.4 Global results

As an alternative to configuring results locally to specific actions, you can also configure results globally. This means that a result can be used from any action in the entire package. When an action returns a result control string, such as "error", the framework first consults the set of results as defined locally to the action. If no error result is found, it then consults the set of globally defined results for an error result. In this fashion, your actions can utilize any globally defined result just by returning the appropriate string. Note that a locally defined result will override a globally defined result during this lookup process.

This is particularly useful for such results as errors. It's common to display all error states via a standard error page. This page can be reused throughout the application. We've already discussed this in chapter 4, in the context of the Exception interceptor. For that example, we declared a global error result. Declarations of global results go inside your package's global-results element, as seen in the following snippet from our chapter 4 example:

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

The `result` declaration is no different. The only difference is the location of the declaration; it's inside the `global-results` element rather than in an `action` element. With this declaration in your package, any of your package's actions can return a result string of `error` and the rendering of the result page will be handled by this JSP result. Please refer back to chapter 4 if you want to check out the implementation.

8.5 Summary

In this chapter, we saw the last of the framework's core MVC components, the result. Results provide an encapsulation of whatever work should occur after the action has fired. As we've seen, this is most frequently the generation of an HTTP response that's sent back to the client browser as an answer to its original request. For many developers, the JSP-based generation of an HTML page will serve most of the needs of their applications. We've seen that the framework also provides support for using a couple of other technologies to render these pages: Velocity and FreeMarker.

One of the most important things that we covered in this chapter was implementing a custom result that can return a JSON response suitable for Ajax requests. This custom result serves a couple of purposes. First, it helps demonstrate how to integrate Ajax applications into the Struts 2 framework. Ajax techniques are still somewhat of a moving target, and integrating with them will be a focal point for many development teams in the near to immediate future. The flexibility of the Struts 2 framework was intended for just such cases.

We hope that the JSON custom result also demonstrates the internal details of results in general, so that you can confidently come up with custom result types as solutions to the unforeseen integration problems you'll assuredly be faced with. Custom results can provide powerful solutions. They can access all the important data from the request, including the `ValueStack`, `ActionContext`, and even the action itself. Furthermore, the `result` component has been designed to keep the action completely oblivious to the details of the result. This can provide powerful reuse of both results and actions. As we noted when implementing the Ajax artist browser for the Struts 2 portfolio, our action that looks up an artist by username could just as easily be used in a non-Ajax context.

With the completion of this chapter, we've completed not only our tour of the view layer of Struts 2 but our tour of the core components themselves. The next part of the book addresses finer points of web application polish, such as integration with Spring for IOC resource management, data persistence with Hibernate, the framework's XML-based validation framework, and support for internationalization.