

STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

 HANNING

Chapter 14. Migration from Struts Classic.....	1
Section 14.1. Translating Struts Classic knowledge.....	2
Section 14.2. Converting by piecemeal.....	8
Section 14.3. Summary.....	21

14

Migration from Struts Classic

This chapter covers

- Migrating from Struts 1 to Struts 2
- Switching to the new tag library
- Breaking up message resources
- Migrating one piece at a time

So you're convinced Struts 2 is worth the effort to learn, but you also have all this knowledge and experience with Struts 1, also called Struts Classic. And what about all those Struts 1 websites in production? This chapter compares and contrasts the similarities and differences between the two Struts versions and provides useful migration strategies for you. The good news is you don't have to relearn everything. In fact, some of the esoteric things you had to remember to do in Struts Classic have been eliminated. Also, many features we always wished for in Struts 1 have finally arrived in Struts 2. So grab a lovely beverage and let's get started. I am going to trust you are familiar with a version of Struts Classic.

14.1 Translating Struts Classic knowledge

You come to work on Monday morning to learn that your company is adopting this new version of Struts. At first you're excited to learn a modern web framework, but you've also heard it's quite different from Struts Classic. What does this mean for your career? What if you're no longer the person everyone calls the Struts guru? How do you quickly become the Struts 2 expert you know you can be? Luckily, you've already taken the first step by purchasing this fine book. In fact, chapter 1 showed you how the two Struts flavors aren't really that different. You should've walked away from chapter 1 realizing Struts Classic and Struts 2 are both MVC-patterned, but Struts 2 provides a much cleaner implementation. This chapter will transition your expert status to the new Struts before anyone realizes your confidence was shaken. Our first stop will be the Struts action.

14.1.1 Actions

You'll be happy to know the action is still the workhorse in Struts 2 that it was in Struts Classic. In fact, it's now a thoroughbred! The first change to grasp is that an action is no longer a singleton. Each request gets its own action instance that is thread-safe. This means you can have first-class instance variables of complex types! The next change is that the action has divorced the Servlet API. Listing 14.1 reveals a typical Struts 1 action. Note how the Servlet API and Struts 1 framework objects are bound into your code. Not only does this make the action hard to test, it also blurs the division of responsibility between the action and the server plumbing. A couple more things to recognize before we look at the Struts 2 action are the name of the method and the action class that our `SamplesAction` is extending. Listing 14.1 shows the requirements of the Struts Classic actions.

Listing 14.1 Struts Classic Action

```
public class SamplesAction extends Action {           ❶
    public ActionForward execute(                     ❷
        ActionMapping mapping,                        ❸
        ActionForm form,                             ❹
        HttpServletRequest request,
        HttpServletResponse response)
    {
        SamplesWebForm webForm=(SamplesWebForm) form; ❺
        // business logic here...
        return mapping.findForward("success");        ❻
    }
}
```

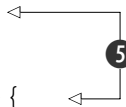
First, our `SamplesAction` was required to extend the Struts Classic `Action` class ❶. Next, we were required to receive the framework mapping and form references ❸ in the `execute()` method ❷. The framework form passed to the `execute()` method was always quirky. It had to be cast ❺ to a subclass, which could easily throw an exception and was chock full of strings representing user input that we had to wrestle into “real”

data types. You can also see that we had to receive Servlet API references ④ for the request and response. Lastly, we had to determine the ActionForward object ⑥ and return it to the framework. If you were reading closely, we were expected to do many things the framework should've been doing itself! Now let's look at the corresponding Struts 2 Action in listing 14.2.

Listing 14.2 Struts 2 Action

```
public class SamplesAction{ ①
    private SamplesBean model; ②

    public String execute(){ ③
        // business logic here ...
        return "success"; ④
    }
    public SamplesBean getModel(){
        return model;
    }
    public void setModel(SamplesBean model){
        this.model = model;
    }
}
```



First, our Struts 2 SamplesAction is a simple POJO ①. Now that action classes aren't singletons, we have an instance variable ② that preserves the user input. The method requires us to receive no arguments ③ and can be named whatever we like. The return value is a simple string that serves as a symbolic name ④ the framework digests to determine what should happen next. Lastly, the get/setModel() ⑤ behaviors are accessed by the framework to keep its internal value stack synchronized with user inputs.

Pretty slick, huh? This JavaBeans action contains no Servlet API dependencies, so testing it is a breeze. The fact that the execute() method can be called anything you like turns out to be an extremely cool feature that we'll explore in more detail later when we discuss wildcard mappings. Now let's take a closer look at how the data flows between the Web and Struts 2.

14.1.2 What happened to ActionForms?

Can we have a moment of silence in memory of the ActionForm? The ActionForm has been booted out of the framework and we're better off because of it. It never measured up and at one point in the evolution of Struts Classic, we were even allowed to pretend it didn't exist by using the infamous DynaForm. Of course the DynaForm was just an ActionForm in disguise. The ActionForm was weak at best and only existed as a bridge to shuttle user inputs between the action and web page. It did little more than hold the String request parameters from the HttpServletRequest so we didn't have to fish them out ourselves. But once we received the form, we were left to convert it into a business-savvy domain model before we could do anything useful with the user inputs.

Struts 2 has removed the ActionForm and now serves the action a business-savvy domain model freshly adapted from the user inputs. Struts 2 translates and converts

all the String request parameters to their complex data types found in the action. If you've written adapters to map web pages to domain objects, you can already see this automation will save many hours of manual work. Struts 2 also takes care of translation in the other direction to satisfy the String requirements of the HTML.

The translation between Strings and your business-savvy domain model is performed by the OGNL `DefaultTypeConverter`. It understands many data types, dates, arrays, maps, and collections. Of course, as you discovered in chapter 5, if you have requirements beyond the scope of the framework and would like to create your own type converters for your custom data types, here's the recipe:

- 1 Create your converter extending `StrutsTypeConverter`.
- 2 Register your new converter for either an individual class in `ClassName-conversion.properties` or globally in `xwork-conversion.properties`.

In case you jumped to this migration section before reading the earlier chapters, it's worth mentioning that Struts 2 populates your objects with the user input via an interceptor. Recall that interceptors do most of the heavy lifting in this new framework (see chapter 4), so this makes sense. This interceptor uses the web page input control name to find a setter method on your action class. Once the set method is located, it determines the actual data type the method is expecting and has the String converted to this anticipated type (see chapter 5). Lastly, the converted data type is passed to the setter method and the object is injected with the user input. Figure 14.1 illustrates this parameter-setting flow.

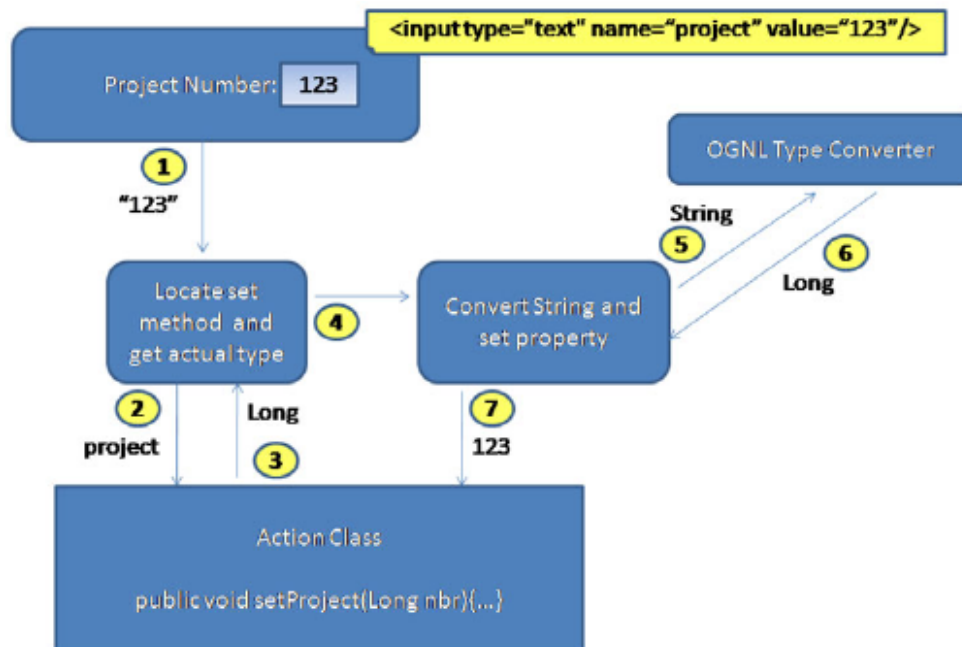


Figure 14.1 Parameter interception and conversion

Of course, if our HTML control had been named `department.project`, then our action would contain a `set/getDepartment` method that returns a `Department`. The `Department` would contain a `getProject()` method that would ultimately contain the `setProject(...)` operation. This powerful Struts 2 capability will accommodate shuttling user inputs between the web page and a rich domain model without human intervention.

Now that we've explained how Struts 2 has effectively eliminated the `ActionForm`, let's discuss web pages and their custom tags that operate as the opposite end of this data exchange.

14.1.3 Switching tag libraries

Struts has always had an affinity for tag libraries. In fact, the Struts tag libraries work closely with the Struts framework itself to handle data movement and workflow navigation. They're doing many "Struts" things behind the scenes. Struts Classic had several different tag libraries with an occasional overlap among them. Struts 2 has cleaned this up by combining its tags into one library. This simplifies figuring out which `taglib` directive to include in your web pages.

The biggest difference between the two versions of Struts is that Struts 1 tags were dependent on the old `ActionForm` whereas Struts 2 utilizes the OGNL `ValueStack`. For the sake of comparison, these two schemes are about as similar as a Ford Escort and a Chevrolet Corvette! Whereas the Struts 1 scheme merely shuttled string data types between the web page and `ActionForm`, the Struts 2 approach is amazingly different. Unless you've jumped right to this chapter in the book, you should have a clear understanding of the `ValueStack` and the Object-Graph Navigation Language that form the core of Struts 2. This `ValueStack` is an ordered list of real-time objects. This means objects are being pushed on and popped off the stack as the framework executes requests. The tags are one such agent allowed to push and pop the stack. As properties are requested, the stack is searched from the top down. This flexibility allows us to reference properties without needing to first know which object contains the property. Consider figure 14.2.

As you can see from this illustration, the OGNL searches for your expression starting at the top of the stack and proceeds downward until either it's located or the list is exhausted. Learning this powerful navigation language will be a valuable tool in your toolkit. Visit the

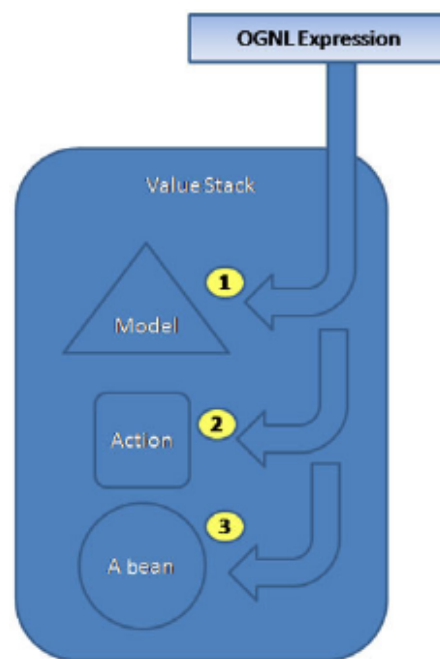


Figure 14.2 ValueStack and expression navigation

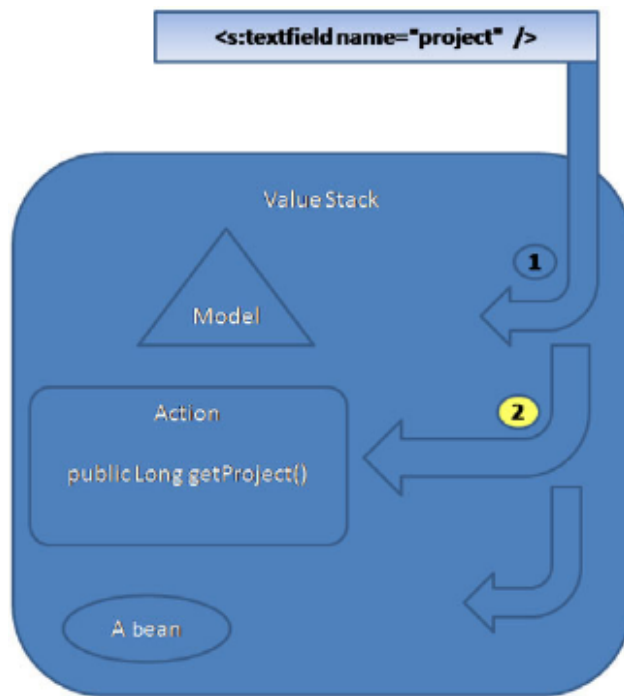


Figure 14.3 Custom tag interplay with ValueStack

website <http://www.ognl.org> for more information. Now, let's take a look at an example of our web page interacting with dynamic objects on this stack. Figure 14.3 reveals how a Struts 2 custom tag would find the project property.

In this case, the first object where the project property could be accessed was in the Action object.

Migrating your Struts 1 web pages to Struts 2 will require replacing the Struts 1 tag directives with the single `<%@ taglib prefix="s" uri="/struts-tags"%>`. Details of this conversion are included later in this section. The Struts 2 tags automatically include support for internationalization and field-level messages without much fuss. This allows you to remove much of the HTML you had to write to get this support with the Struts Classic tags.

Before leaving this section, we need to comment on JSTL. Struts Classic contained several tag libraries, each of which contained a related grouping of tags. Once JSTL came onto the scene, you were encouraged to select the JSTL version where there was a corresponding tag in a Struts Classic library. This was because of the JSTL support for the new Expression Language that enabled objects to be interwoven with the tag itself to produce the desired markup.

The JSTL `${someObject.someMethod}` expression language is rich and can be used to reach deep into an object graph. Plus there's also support for working with Maps and Lists. Struts 2 has full access to JSTL in the same way Struts Classic does. Where we see the real difference is that Struts 2 has its own tag libraries that interact with the OGNL and the ValueStack where Struts Classic knows nothing about these new capabilities.

Next we look at how to organize all those internationalized resource files.

14.1.4 Breaking up message resources

Java has always provided native support to allow us to easily access locale-sensitive text and formatting. The hardest part is having all the text translated into other dialects, which has nothing to do with Struts. Struts Classic leveraged this native support through the use of the `java.util.PropertyResourceBundle` and the `java.util.ListResourceBundle`, and so does Struts 2. Where Struts 2 differs is in the flexibility you have in selecting a language bundle. This is covered in greater detail in chapter 11. So that we don't confuse what's meant by the bundle selection, let me first identify what we mean about this selection. Suppose our website has been designed to accommodate English, German, Spanish, and French. This would require us to have the following files on our classpath:

- `MyWebBundle.properties`
- `MyWebBundle_en.properties`
- `MyWebBundle_de.properties`
- `MyWebBundle_es.properties`
- `MyWebBundle_fr.properties`

The file selected would depend on the language the requester passed in on the HTTP header. For sake of illustration, let's suppose it was a French requester. Judging from what we know so far, `MyWebBundle_fr.properties` would be selected. In Struts 1, all your keys were expected to be in this file. Since it's never a good idea to put all the text for your entire website in a single file, Struts 1 allowed you to have French text in multiple French files. However, it was a messy technique that was clearly added as an afterthought. It required a custom Java class and you had to prefix your message keys with a file designator so the custom class could determine which file it should select your text from. It was clumsy and error-prone.

Struts 2 makes it easy to set up language files for your entire application, sections of the site, or even down to the action and property level. At the application level, it works the same as Struts Classic. Simply provide the default bundle in the `struts.properties` as follows:

```
struts.custom.i18n.resources=resources.package
```

Where things get more interesting in Struts 2 is how you can deal with special cases. Let's suppose you have special needs for an action named `MemberAction`. If you create a bundle called `MemberAction.properties` and place it alongside this action class, Struts 2 will retrieve values from this locale bundle. There are many variations between choosing the scope of the entire application or an individual Action. This following is the search order:

- 1 A `ResourceBundle` is selected with the same name and package as the class of the object on the stack, including the interfaces and superclasses of that class. The search hierarchy is as follows:
 - Look for the message in a `ResourceBundle` for the class.

- If not found, look for the message in a `ResourceBundle` for each implemented interface.
 - If not found, traverse up the class's hierarchy to the parent class, and repeat from step 1.
- 2 If the message text isn't found in the class hierarchy search and the object implements `ModelDriven`, call `getModel()` and do a class hierarchy search for the class of the model object. There was no concept in Struts Classic for scoping messages to a particular bean.
 - 3 If the message text still is not found, search the class hierarchy for default package texts. For the package of the original class or object, you look for a `ResourceBundle` named `package.properties` in that package. For instance, if the class is `com.strutsschool.enrollment.MemberAction`, look for a `ResourceBundle` named `com.strutsschool.enrollment.package.properties`. You continue along this line for each superclass in turn.
 - 4 If Struts 2 hasn't found the text at this point, it checks whether the message key refers to a property of an object on the `ValueStack`. If a search for members on the `ValueStack` returns a nonnull object and the text key you're looking for is `member.course.description`, use the member's class to look for the text key `course.description`, searching up its class hierarchy, and so on, as in previous steps.
 - 5 The last resort is to search for the text in the default `ResourceBundles` that have been registered in `struts.properties`.

As you look at this flexibility, it's clear that Struts 2 learned a valuable lesson from Struts 1. All these capabilities might at first seem ridiculous, but once you find yourself faced with that exceptional use case, you'll be happy to know the framework is prepared to provide an elegant solution to your problem.

A wise man once said you can't eat an elephant in one bite. I didn't understand what he meant until I was faced with migrating a large Struts Classic web application to Struts 2. In the next section, we begin discussing how we can do this in small increments.

14.2 *Converting by piecemeal*

We're going to extend a convention that's been followed throughout the book. We'd like to refer to Struts Classic as S1 and Struts 2 as S2. This makes it easier to discuss the two frameworks as we compare and contrast them. While S2 is a quantum leap forward, converting your S1 application is fairly mechanical. We'll soon begin to see tools that assist with the conversion, but for now, you can also find comfort in the fact that we don't need to convert everything at once. Much like eating an elephant in one bite, this just isn't practical. There are several possibilities when it comes to the S1 and S2 unity:

- Leave the S1 application unchanged.
- Convert the entire S1 application to S2.

- Merge S1 and S2 technologies and convert using a piecemeal approach:
 - Calls between the two are easy
 - The S1 plug-in allows you to use S1 actions in an S2 application

If you have a stable S1 website that doesn't require maintenance, you might as well leave it alone. If you have a relatively small S1 site that's evolving, you might consider a wholesale migration to S2. Lastly, since most S1 applications are large, we concentrate on the merge and piecemeal approach, with a sidebar on the Struts 1 plug-in. If you're the proud parent of an S1 website and would like to adapt it to S2, let's begin our conversion.

14.2.1 Eating an elephant a piece at a time

While it's true that S2 is easier to configure and provides many more features, the two frameworks can coexist, as they each

- Declaratively map a URL to a Java class
- Declaratively map a response to a web resource
- Contain custom tags to link requests to their respective framework

The first thing to consider is the declarative URL mapping that determines where the request should be routed. S1 is typically mapped to digest a URL matching `/*.do` and S2 `/*.action`. While these may be configured using different extensions, they need to be unique if you plan to combine the two frameworks into a single web application. Let's take the first bite of the elephant. The first step is to copy the S2 JARs to the `WEB-INF/lib` folder of our S1 application and add the S2 elements to the `web.xml` file. Listing 14.3 covers these elements in detail.

Listing 14.3 web.xml combining both S1 and S2 into the same web application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- Struts 2 -->
  <filter>
    <filter-name>struts2</filter-name>           ❶
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>

  </filter>
  <!-- extensions are included in the struts.properties -->
  <!-- struts.action.extension=action -->
  <filter-mapping>
    <filter-name>struts2</filter-name>           ❷
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

```

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener ❸
  </listener-class>
</listener>
<listener>
  <listener-class>
    org.apache.struts2.tiles.StrutsTilesListener ❹
  </listener-class>
</listener>

<!-- Struts 1 -->
<servlet>
  <servlet-name>action</servlet-name> ❺
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>

  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/classes/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name> ❻
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<!-- Either version -->
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>

```

The S2 framework leverages a filter to intercept requests ❶ and the URL mapping is set to /* ❷, where the actual extensions are specified in the `struts.properties` file. This property is shipped to handle requests with the `.action` extension, but can be changed. S1 mappings ❺ ❻ are shown for unity. The listener ❸ fires up the Spring framework, which S2 uses to instantiate its objects. This listener expects the file `applicationContext.xml` to exist in the `WEB-INF` folder even if it's empty. This file is shown in listing 14.4. The listener ❹ fires up the Tiles 2 framework, which S2 may interface with for common look and feel.

Listing 14.4 `applicationContext.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
"-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd ">
<beans>

</beans>

```

Our S1 application is now capable of handling both S1 and S2 requests. The `*.do` requests will be handled by S1 and the `*.actions` by S2. We can now begin to add new S2 features and also migrate existing S1 artifacts to S2. Likewise, we may have action mappings in S1 that direct to S2 resources and vice versa. Infusing our existing S1 application with S2 capabilities is what we refer to as *merging technologies*. This allows us to systematically migrate S1 to S2 while immediately being allowed to share many resources between the two “sides” of the web application.

Next, we look at how the action mappings have changed.

14.2.2 The action mappings

One of the real advances in website design was the advent of symbolic mapping. This allowed us to modify website behavior without getting tangled up in the HTML. This declarative mapping has been the hallmark of S1 for years and has improved markedly in S2. Any seasoned S1 developer can tell you about the hours she’s spent modifying `struts-config.xml`. This file is essentially a registry where the symbolic features of your website are matched with their concrete counterparts. Let’s take a look at a typical S1 action mapping in listing 14.5.

Listing 14.5 Struts Classic action mapping

```
<struts-config>
  <form-beans>
    <form-bean name="reportForm" type="ReportForm"/> ❶
  </form-beans>

  <action-mappings>
    <action
      path="/reportDateSelection" ❷
      input="/reportDateSelection.page" ❸
      name="reportForm" ❹
      type="ReportDateSelection" ❺
      scope="request"> ❻
      <forward name="success" path="reportByDate.page"/>
      <forward name="largeDateRange" path="reportWarning.page"/> ❼
    </action>
  </action-mappings>
</struts-config>
```

It was easy to retire as a wealthy S1 developer if you were paid by the keystroke! As you can see, this single web feature was verbose and often led to a bloated `struts-config.xml` file. In order to capture the user inputs for this action, we first had to create a form bean ❶ that was separate from the action itself. This form bean was assigned a name and could be associated with many different actions. The type `ReportDateSelection` ❺ is the Struts action class associated with this mapping. The path attribute ❷ is where we specified the symbolic name of the web feature. In this case, we’re prompting the user for a date range to be used in selecting data for a report. The input attribute ❸ is where we specify what page should be displayed if validation fails. The name attribute ❹ is what

links this action to the form bean ❶, and the scope ❷ is where we specify where S1 should store the form bean. Lastly, the forward tags ❸ are where we specify the list of eligible targets from which the action may select.

A web request for `someContext/reportDateSelection.do` would invoke this S1 action mapping. S2 streamlines this mapping substantially by leveraging smart defaults and storing user inputs inside the action itself. Listing 14.6 shows what this action mapping looks like in Struts 2.

Listing 14.6 Struts 2 action mapping

```
<struts>
<package name="invoicing" namespace="/invoicing"> ❶

<action name="reportDateSelection" class="ReportDateSelection"> ❷
  <result name="input">reportDateSelection.page</result>
  <result name="largeDateRange">reportWarning.page</result> ❸
  <result>reportByDate.page</result>
</action>

</package>
</struts>
```

The first thing to notice is the new package tag ❶ used in our mappings. This works in a way similar to packages in Java, where similar actions are grouped together in a common namespace. The web request `someContext/invoicing/reportDateSelection.action` would invoke this S2 action. Note how this allows us to use this same action name in another package namespace as in `someContext/payables/reportDateSelection.action`. The action mapping itself ❷ has been greatly simplified. First off, there's no form bean to mess with, as S2 encapsulates the user inputs inside the action itself. Lastly, the eligible targets or "next steps" resulting from this action are cleanly laid out in the form of result tags ❸. Note that the last result doesn't specify a name. This is because success is the default. Also, unlike the S1 `ActionForward`, results in S2 can actually help prepare the response.

We've now discussed how to infuse S2 capabilities into our S1 application and we've looked at S1 and S2 mappings side by side. Now we'll look at the steps to migrate S1 artifacts to S2.

14.2.3 Where the action meets the form

In S1, the user inputs were stored in a form bean that the framework passed into the action class. S2 combines the utility of the form within the action itself, thereby eliminating the form bean. So the first step in the migration is to modify the S1 action to accommodate the user inputs. This could be as simple as removing the S1 code from the form bean and including this POJO as a field inside the action class. Listings 14.7 and 14.8 show the before-and-after form beans.

Listing 14.7 Struts 1 action form

```
public class ReportForm extends ActionForm {
    private boolean average;
    private boolean totals;
    private String fromDate;
    private String toDate;
    getters/setters...
}
```

Annotations: ① Framework class (points to `extends ActionForm`), ② Simple types (points to the private fields).

This S1 form bean extends a framework class ① and is comprised largely of primitive data types ②. Listing 14.8 shows the migrated S2 version of this bean, which is a simple POJO.

Listing 14.8 Struts 2 POJO

```
public class Report {
    private boolean average;
    private boolean totals;
    private Date fromDate;
    private Date toDate;
    getters/setters...
}
```

Annotations: POJO (points to the class name), Actual data types (points to the private fields).

As you can see, this is a plain Java class with useful data types to contain user inputs. In listing 14.1, we looked at a typical S1 action class and saw that it expected a framework form bean. In listing 14.9, we peer into the S2 action class to see how the user data is made available.

Listing 14.9 Struts 2 action with model bean

```
public class ReportDateSelection extends ActionSupport{
    private Report model;
    public String execute(){
        // business logic here ...
        return SUCCESS;
    }
    public Report getModel(){
        return model;
    }
    public void setModel(Report model){
        this.model = model;
    }
}
```

Annotations: ① Extends S2 class (points to `extends ActionSupport`), ② Web data encapsulated in model bean (points to `private Report model;`), ③ Accessors for ValueStack (points to `getModel()` and `setModel()`).

First, our S2 `ReportDateSelection` extends the S2 `ActionSupport` base class ①. While it isn't necessary, it contains several handy behaviors and tokens that we would end up writing ourselves. In this case, we're only leveraging the `SUCCESS` token. Next, since action classes are no longer singletons, we can have an instance variable ② that

is the model POJO that preserves the user input. Lastly, the `get/setModel()` ③ behaviors are accessed by the framework to keep its internal `ValueStack` synchronized with user inputs.

Let's turn our attention to web pages and how we go about migrating them to S2. This is probably the most time-consuming part, as this is the dimension of a website that the user sees. Much care and concern goes into this layer of development to allow the system to be user-friendly. In the next section, we turn our S1 pages into their S2 counterparts.

14.2.4 Turn the page

In the previous section, we eliminated the S1 form bean and moved the user inputs into the action itself. Now that we've explained how Struts 2 has effectively eliminated the `ActionForm`, let's discuss the web pages and their custom tags that operate as the opposite end of this user data exchange. The first thing that changes is the tag library declaration. Rather than have several libraries, S2 has combined the tags into a single library. The next big difference is that S2 tags get and set objects on the `ValueStack` whereas S1 used a form bean. Lastly, the S2 tags do more than simply shuttle web page content; they assist in the presentation of the web page. This is a real time saver and results in web pages that are much more readable. The S1 custom tags helped separate the Java from the presentation code in your web pages. S2 custom tags not only eliminate Java from your web pages but also drastically reduce the HTML you're expected to write. Before we get into the nitty-gritty, take a look at the web page in figure 14.4. Admittedly, it isn't complex from the end-user's point of view, but designing a screen that's dynamically generated from static and dynamic portions can be a considerable challenge.

Listing 14.10 shows the source code behind this S1 web page. Note that much of this code has to do with positioning elements on the page and pulling language constants from internationalized resource bundles. Also, we placed the user messages next to each control that might fail validation.

Figure 14.4 Report selection web page

Listing 14.10 Struts 1 web page

```
<%@taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<H3> <bean:message key="promptTitle" /> </H3>
<BR>
<html:errors />
<html:form action="reportParameters">
<TABLE border="0" cellpadding="0">
<TBODY>
<tr>
```



```

<TD align="right">
<b><bean:message key="average" /></b>
</TD>
<TD>
<html:checkbox property="average"></html:checkbox>
</TD>
</tr>
<tr>
<TD align="right">
<b><bean:message key="totals" />
</b>
</TD>
<TD>
<html:checkbox property="totals"></html:checkbox>
</TD>
</tr>
<tr>
<TD align="right">
<b><bean:message key="fromDate" /></b>
</TD>
<TD>
<html:text property="fromDate"></html:text>
</TD>
<TD>
<html:errors property="fromDate" />
</TD>
</tr>
<tr>
<TD align="right">
<b><bean:message key="toDate" /></b>
</TD>
<TD>
<html:text property="toDate"></html:text>
</TD>
<TD>
<html:errors property="toDate" />
</TD>
</tr>
</TBODY>
</TABLE>
<html:submit property="submit">
<bean:message key="ok" />
</html:submit>
</html:form>

```

That's a bunch of code for such a compact web pagelet, and we haven't even discussed cascading style sheets. Listing 14.11 shows the source code for this same web page designed using S2 custom tags.

Listing 14.11 Struts 2 web page

```

<%@ taglib prefix="s" uri="/struts-tags"%>
<H3><s:label key="promptTitle" /></H3>
<s:form>

```

```

<s:checkbox key="average" labelposition="left"/>
<s:checkbox key="totals" labelposition="left"/>
<s:textfield key="fromDate" />
<s:textfield key="toDate"/>

<s:submit key="ok" action="reportParameters" />
</s:form>

```

What do you think about this? The same web page in S2 requires a fraction of the developer coding because the standard markup is generated by the tags. Actually, the markup is merely associated with the tags. S2 utilizes FreeMarker to generate the markup for the tags. In this case, the markup was HTML, but FreeMarker can generate any type of markup you like. Imagine being able to generate markup for HTML, WML, XSLT, and more, all from the same source. S2 has packaged these FreeMarker templates into themes to generate markup according to your preferred look and feel. Note that each tag contains a key property, which serves multiple roles. If you refer to listing 14.8, you'll find these keys are also the properties in our model object. What isn't as obvious is the fact that these keys are also located in our language ResourceBundles, which is how the S2 tags retrieved the screen labels. These keys also become the DOM id and name properties for the page elements that are essential for validation and AJAX support.

14.2.5 *No speak English*

Chances are you have a rich set of localized messages that have been serving your S1 application. Since S1 and S2 both leverage the underlying Java i18n facilities, those same resources can also serve the S2 framework. In S1, we specified our resource files in the struts-config.xml with the following tag:

```
<message-resources parameter="applicationResources" />
```

This existing ResourceBundle can be used in S2 by placing the following entry in your struts.properties configuration file:

```
struts.custom.i18n.resources= applicationResources
```

This entry indicates that we have a ResourceBundle with the root name applicationResources and an alternate file named applicationResources_es.properties, where es is a standard locale code for Spanish. Both the ResourceBundle and the struts.properties file should be placed in the classes directory of a web application, so that they're accessible on the classpath. If you want to expand your migration to take full advantage of the flexibility S2 has to offer in this area, refer to section 14.1.4 for a review of the ways you can further break these message resources apart. Listing 14.12 reveals sample content for our reporting web page.

Listing 14.12 Localized language files

```

### applicationResources.properties
promptTitle=Report Selection
average=Compute Averages

```

```

totals=Compute Totals
type=Report Type
fromDate=From Date
toDate=To Date
ok=OK

### applicationResources_es.properties
promptTitle= Informe de selección
average= Promedios del cálculo
totals= Totales del cálculo
type= Tipo de informe
fromDate= A partir de fecha
toDate= Hasta la fecha
ok=OK

```

S1 action classes retrieved localized messages with

```
getResources(request).getMessage("promptTitle")
```

S2 is more succinct:

```
getText("promptTitle")
```

This is another contrast between S1 passing HTTP objects during method calls versus S2 dependency injection. The `getText` behavior in S2 can interrogate the request on its own. Lastly, you may recall that, in S1, if you asked for a message whose key couldn't be found in the resource file, you received a null in return. S2 will return the message key in this case, which indicates a missing message.

Now that we've discussed the changes in language support, let's turn our attention to validation. The language support we just looked at will play a big role in user messages as we construct meaningful dialog with the user interacting with our website.

14.2.6 The data police

How do your action classes handle invalid data entered by the user? We hope you never allow invalid data to make it into your action classes, at least not the kind that could be easily prevented. We now compare the S1 and S2 validation frameworks to see how they perform their traffic cop duties. These respective frameworks are layered between the web page and action class, and the validation rules for determining the data validity are configured independent of either one. In S1, the traffic cop was the Commons Validator, which leaned on the S1 ActionForm derivatives ValidatorForm and ValidatorActionForm. There was a great deal of stitching involved to get Commons Validator to cooperate with S1, and when one or the other API would evolve, you were back with a needle and thread to get it working again. In fact, as I was writing this chapter, my application started throwing heaps of stack trace data at the console that required a couple of hours to discover a mismatch in XML files. The funny thing is, while I was reading through the minutiae I had this vague reminiscence of researching the same problem a few years ago.

The validator engine in S2 is a core component of the framework that evolved from Open Symphony XWork. After working with it awhile, you'll discover how easily

it performs all the tests the Commons Validator handled, even the complicated validations you once had to write Java code to authenticate. This section is intended to help you migrate your validations from S1 to S2. For a full tour guide of the S2 validation system, see chapter 10.

We compare the differences by validating the data keyed into the web page shown in figure 14.4. I'm assuming you already have Common Validator configured and plugged into S1, so we won't discuss that here. With S1, we added our rules to a validations.xml file as shown in listing 14.13.

Listing 14.13 Struts 1 validations.xml file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
"-//Apache Software Foundation//DTD Commons Validator Rules
Configuration 1.3.0//EN"
"http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd ">

<form-validation>
  <formset>
    <form name="S1ReportForm">
      <field property="fromDate" depends="required,date">
        <arg0 key="fromDate"/>
      </field>
      <field property="toDate" depends="required,date">
        <arg0 key="toDate"/>
      </field>
    </form>
  </formset>
</form-validation>
```

The `<formset>` ❶ tag defines a grouping of `<form>` ❷ tags according to a country and language code. The `<form>` ❸ section appeared many times in a typical S1 application, and each grouping addressed a particular form bean/web page scenario. As you can imagine, this file became large and was difficult to administer in a multiple-developers environment. The form bean property validity depends ❹ on the rules it's registered to pass. If a rule fails validity, the property name for this key ❺ is substituted into the respective error message that's displayed back to the web browser.

S2 takes a simpler approach to solving the validity problem. Rather than lump all the website validations in a single file, S2 allows you to manage the rules at the object level. This eases the burden of source file contention and allows you to focus on simpler units of work. There are many conventions you can use to achieve more or less granularity, and a common approach is the naming convention `{Action}-validation.xml`. However, if annotations are your deal, you can decorate your action classes instead of using the XML approach. In addition to base validation, S2 offers client-side validation with JavaScript and a new Ajax-based option. Let's take a look at the S2 validation for our report action shown in listing 14.9. The `{Action}-validation.xml` naming convention for this action yields `ReportDateSelection-validation.xml`, revealed in listing 14.14.

Listing 14.14 Struts 2 ReportDateSelection-validation.xml file

```

<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>
  <field name="model.fromDate"> ❶
    <field-validator type="required"> ❷
      <message key="required" /> ❸
    </field-validator>

    <field-validator type="date"> ❷
      <message key="date" /> ❸
    </field-validator>
  </field>
</validators>

```

The `<field>` ❶ tag defines a grouping of `<field-validator>` ❷ tags, each of which validates the field against specific rules. If a rule fails validity, the message key ❸ is constructed as the error message displayed in the web browser. The message in S2 can be constructed with variable data pulled directly from the `ValueStack` associated with this request. This allows you to build messages that make good sense to the user without having to jump through all the `{0}`, `{1}`, `{2}` hurdles to substitute dynamic bits into a message. If it's easy, chances are it'll get done. Please review the chapter on validation to get the big bang. This isn't just the Commons Validator warmed over!

This evening as I'm shopping the Web for better rates on HDTV service, I'm again reminded of how many Struts Classic applications are hosted on this planet. Most of the big players are using Struts Classic. As I gaze into those embedded `.do` web extensions, I start to think of ways I could offer you better returns on your S1 investments as you move to Struts 2. Before we wrap up, I want to discuss a plug-in that allows you to snap Struts Classic actions, forms, and validations into a Struts 2 application. I'm well aware that certain S1 artifacts are too "something" to refactor right away, so this might be a tool to allow you to continue receiving dividends on those works of art.

14.2.7 Can we just get along?

If you're a student of design patterns, chances are the open-closed principle is among your 10 commandments. Like any well-designed software, you should be able to extend the functionality without modifying existing code. Struts 2 leverages plug-ins for this very purpose. If you use Firefox or Eclipse, you already know how this works. When you need to use a feature that wasn't included in the "baseline," you install a plug-in that provides the capability you seek. Chapter 12 discussed plug-ins in detail, so here we simply discuss a plug-in that allows Struts Classic components to appear as if they were Struts 2 components. This plug-in is called `struts2-struts1-plugin`, appropriately enough, and to take advantage of it, you simply drop it in the `WEB-INF/lib` folder of your application and start using the new feature. This plug-in utilizes available S2 interceptors to adapt the request life cycle of an S1 action. Studying this will not only help you leverage your S1 actions, but may also reveal smarter design decisions.

All right, we know the action class was a singleton in S1 and that it interacted with a form bean in a scope you specified. S2 actions are thread-safe and have no concept of a form bean. So let's see how the S1 plug-in dresses up the old action/form pair to appear as an S2 action. If you aren't comfortable with interceptors, you might want to refer to the index for a refresher because you're about to see the interceptor magic performing a few tricks.

This plug-in creates a new S2 package called `struts1-default` that extends `struts-default`. The new `struts1-default` package includes new interceptors that are sandwiched into a default interceptor stack assembled to mimic the S1 request cycle logic. You simply create an S2 package that extends the `struts1-default` package and you're ready to roll. Listing 14.15 illustrates how we configure an S1 action to be executed in your S2 application.

Listing 14.15 Packaging Struts 1 actions in Struts 2 package

```
<?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="hybridActions"
    namespace="/old2new"
    extends="struts1-default">

<action name="enroll" class="org.apache.struts2.s1.Struts1Action" >
    <param name="className">com.strutsschool.s1.actions.EnrollAction</param>

    <interceptor-ref name="scopedModelDriven" >
        <param name="className">s1.webapp.forms.EnrollForm</param>
        <param name="name">enrollForm</param>
        <param name="scope">request</param>

    </interceptor-ref>

    <interceptor-ref name="struts1Stack"></interceptor-ref>

    <result name="success">enrollPage</result>
</action>

</package>
</struts>
```

The `<package>` ① is a typical configuration with a namespace ② and parent package to extend ③. The action mapping is where things start to get interesting. The class you specify here is the S1 plug-in class ④, and it expects a parameter ⑤ which is the qualified name of the existing S1 action class. The next peculiar section is the configuration of the `scopedModelDriven` ⑥ interceptor. This interceptor is an integral part of S2 and has a companion interface with the same name. The S1 class ④ implements this interface so the parameters for `className` ⑦, `name` ⑧, and `scope` ⑨ are where we

specify our S1 action form. The last interceptor ⑩ is actually defined in the S1 plug-in as the stack of interceptors that make the magic happen.

This plug-in also allows you other configurations according to what your S1 mapping looked like. For instance, you might not have used a form bean, or perhaps you were using the S1 Commons Validator. These use cases and more can be configured with the new plug-in.

14.3 Summary

This chapter has covered many practices that you can begin using to migrate S1 apps to S2. It shared a pragmatic approach to migrating your existing applications a piece at a time and wrapped up with a discussion about how the struts2-struts1-plugin allows you to bring your S1 artifacts into the S2 application without tampering with them at all. As you might imagine, this isn't exhaustive coverage of every migration technique possible, as that would be overwhelming. Regardless of the path you choose, this chapter will get you on your path to migration.