STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

Chapter 4. Adding workflow with interceptors..... 1
Section 4.1. Why intercept requests?..... 2
Section 4.2. Interceptors in action..... 5
Section 4.3. Surveying the built-in Struts 2 interceptors..... 8
Section 4.4. Declaring interceptors..... 17
Section 4.5. Building your own interceptor..... 22
Section 4.6. Summary..... 26

4 Adding workflow with interceptors

This chapter covers

- Firing interceptors
- Exploring the built-in interceptors
- Declaring interceptors
- Building your own interceptors

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

After such a bold statement, we have no choice but to back it up with a detailed explanation of interceptors and the role they play in the framework. This chapter will begin by clarifying that architectural role with a brief conceptual discussion. We'll then dissect a couple of simple interceptors from the `defaultStack` (just so we can see what's inside), provide a reference section that covers the use of all the built-in interceptors, and end by creating a custom interceptor to provide an authentication service for our secure package's actions.

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

4.1 Why intercept requests?

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

4.1.1 Cleaning up the MVC

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

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

Some of the tasks undertaken by interceptors are more easily understood as being preprocessing or postprocessing tasks. These are still technically cross-cutting; we recommend not worrying about the semantics of these terms. We present these new terms mostly to give you some ideas about the specific types of tasks handled by interceptors. A good example of a preprocessing task would be data transfer, which we're

Chapter 4. Adding workflow with interceptors. Struts 2 in Action, ISBN: 9781933988078

already familiar with. This task is achieved with the `params` interceptor. Nearly every action will need to have some data transferred from the request parameters onto its domain-specific properties. This must be done before the action fires, and can be seen as mere preparation for the actual work of the action. From this aloof perspective, we can call it a preprocessing task. This is perfect for an interceptor. Again, this increases the purity of the action component by removing code that can't be strictly seen as part of a specific action's core work.

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

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

One of the powerful functional aspects of interceptors is their ability to alter the workflow of the invocation. As we noted, figure 4.1 depicts an instance where none of the interceptors has intervened in the workflow, thus allowing the action to execute and determine which result should render the view. Sometimes, one of the interceptors will determine that the action shouldn't execute. In these cases, the interceptor can

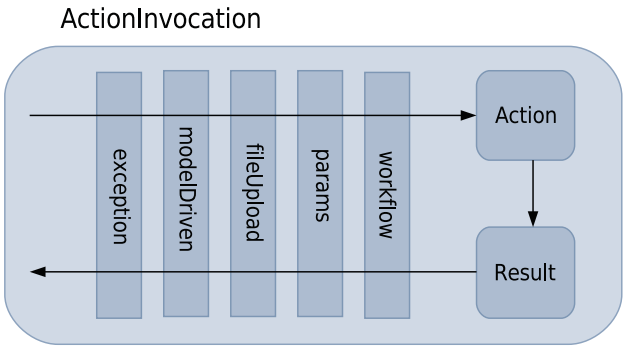

Figure 4.1 `ActionInvocation` encapsulates the execution of an action with its associated interceptors and results.

halt the workflow by itself returning a control string. Take the `workflow` interceptor, for example. As we've seen, this interceptor does two things. First, it invokes the `validate()` method on the action, if the action has implemented the `Validateable` interface. Next, it checks for the presence of error messages on the action. If errors are present, it returns a control string and, thus, stops further execution. The action will never fire. The next interceptor in the stack won't even be invoked. By returning the control string itself, the interceptor causes control to return back up the chain, giving each interceptor above the chance to do some postprocessing. Finally, the result that matches the returned control string will render the view. In the case of the `workflow` interceptor that has found error messages on the action, the control string is `"input"`, which typically maps back to the form page that submitted the invalid data.

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

4.1.2 Reaping the benefits

Layering always makes our software cleaner, which helps with readability and testing and also provides flexibility. Once we've broken these cross-cutting, preprocessing, and postprocessing tasks into manageable units, we can do cool stuff with them. The two primary benefits we gain from this flexibility are reuse and configuration.

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

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

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

4.1.3 Developing interceptors

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

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

4.2 Interceptors in action

Now we'll look at how interceptors actually run. We'll look at the interceptor interface and learn the mysterious process by which an interceptor is fired. Along the way, we'll meet the boss man, the `ActionInvocation`; this important class orchestrates the entire execution of an action, including the sequential firing of the associated interceptor stack. We'll also take the time to look inside the code of two of the built-in Struts 2 interceptors, just to keep it real. But first let's start with the boss man.

4.2.1 The guy in charge: ActionInvocation

A few paragraphs back, we introduced the `ActionInvocation`. While you'll almost certainly never have to work directly with this class, a high-level understanding of it is key to understanding interceptors. In fact, knowing what `ActionInvocation` does is equivalent to knowing how Struts 2 handles requests; it's very important! As we said before, the `ActionInvocation` encapsulates all the processing details associated with the execution of a particular action. When the framework receives a request, it first must decide to which action the URL maps. An instance of this action is added to a newly

created instance of `ActionInvocation`. Next, the framework consults the declarative architecture, as created by the application's XML or Java annotations, to discover which interceptors should fire, and in what sequence. References to these interceptors are added to the `ActionInvocation`. In addition to these central elements, the `ActionInvocation` also holds references to other important information like the servlet request objects and a map of the results available to the action. Now let's look at how the process of invoking an action occurs.

4.2.2 How the interceptors fire

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

Interceptor firing order

When we say the first interceptor in the stack, we're referring to the first interceptor declared in the XML as reading from the top of the page down. Let's look at the declaration of the `basicStack` from struts-default.xml to see exactly what we mean.

```
<interceptor-stack name="basicStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="params"/>
  <interceptor-ref name="conversionError"/>
</interceptor-stack>
```

In the `basicStack`, the first interceptor to fire will be the `exception` interceptor. From here, each interceptor will fire in the same sequence as you would read down the page. So the last interceptor to fire will be the `conversionError` interceptor. After the result has rendered, the interceptors will each fire again, in reverse order, to give them the opportunity to do postprocessing.

Incidentally, the `basicStack`, not to be confused with the default `defaultStack`, is just a convenient chunk of common interceptors that the `struts-default` package makes available to you to ease the process of custom stack building in case you find that the `defaultStack` isn't quite what you need.

Now for the tricky part. Continued execution of the subsequent interceptors, and ultimately the action, occurs through recursive calls to the `ActionInvocation`'s

`invoke()` method. Each time `invoke()` is called, `ActionInvocation` consults its state and executes whichever interceptor comes next. When all of the interceptors have been invoked, the `invoke()` method will cause the action itself to be executed. If this is cooking your noodle, hang in there. It'll clear up momentarily.

Why do we call it a recursive process? Let's have a look. The framework itself starts the process by making the first call to the `ActionInvocation` object's `invoke()` method. `ActionInvocation` hands control over to the first interceptor in the stack by calling that interceptor's `intercept()` method. Importantly, `intercept()` takes the `ActionInvocation` instance itself as a parameter. During its own processing, the interceptor will call `invoke()` on this object to continue the recursive process of invoking successive interceptors. Thus, in normal execution, the invocation process tunnels down through all of the interceptors until, finally, there are no more interceptors in the stack and the action fires. Again, the `ActionInvocation` itself maintains the state of this process internally so it always knows where it is in the stack.

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

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

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

```
public String intercept(ActionInvocation invocation) throws Exception {
   long startTime = System.currentTimeMillis();      (1)
   String result = invocation.invoke();      (2)
   long executionTime = System.currentTimeMillis() - startTime;
     ... log the time ...                                          (3)
   return result;
}
```

The `TimerInterceptor` times the execution of an action. The code is simple. The `intercept()` method, defined by the `Interceptor` interface, is the entry point into an interceptor's execution. Note that the intercept method receives the `ActionInvocation` instance. When the intercept method is called, the interceptor's preprocessing phase consists of recording the start time (1). Next, the interceptor must decide whether it'll pass control on to the rest of the interceptors and the action. Since this interceptor has no reason to halt the execution process, it always calls `invoke()`,passing control to whatever comes next in the chain (2).

After calling `invoke()`, the interceptor waits for the return of this method. `invoke()` returns a result string that indicates what result was rendered. While this string tells the

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

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

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

- During the *preprocessing* phase, the interceptor can be used to prepare, filter, alter, or otherwise manipulate any of the important data available to it. This data includes all of the key objects and data, including the action itself, that pertain to the current request.
- Call `invoke()` or divert workflow. If an interceptor determines that the request processing should not continue, it can return a control string rather than call the `invoke()` method on the `ActionInvocation`. In this manner, it can stop execution and determine itself which result will render.
- Even after the `invoke()` method returns a control string, any of the returning interceptors can arbitrarily decide to alter any of the objects or data available to them as part of their *postprocessing*. Note, however, that at this point the result has already been rendered.

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

4.3 Surveying the built-in Struts 2 interceptors

Struts 2 comes with a powerful set of built-in interceptors that provide most of the functionality you'll ever want from a web framework. In the introductory portion of this book, we said that a good framework should automate most of the routine tasks of

the web application domain. The built-in interceptors provide this automation. We've already seen several of these and have used them in our Struts 2 Portfolio sample application. The ones we've used have all been from the `defaultStack` that we've inherited by extending the `struts-default` package. While this `defaultStack` is useful, the framework comes with more interceptors and preconfigured stacks than just that one. In this section, we'll introduce you to the most commonly used built-in interceptors. In the next section, we'll show you how to declare which of these interceptors should fire for your actions, and even how to arrange their order.

Now let's explore the offerings. If an interceptor is in the `defaultStack`, it'll be clearly noted as such.

4.3.1 Utility interceptors

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

TIMER

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

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

LOGGER

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

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

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

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

4.3.2 Data transfer interceptors

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

PARAMS (DEFAULTSTACK)

This familiar interceptor provides one of the most integral functions of the framework. It transfers the request parameters to properties exposed by the `ValueStack`. We've

also discussed how the framework uses OGNL expressions, embedded in the name attributes of your form's fields, to map this data transfer to those properties. In chapter 3, we explored techniques for using this to move data to properties exposed directly on our actions as well as on domain model objects with `ModelDriven` actions. The `params` interceptor doesn't know where the data is ultimately going; it just moves it to the first matching property it can find on the `ValueStack`. So how do the right objects get onto the `ValueStack` in time to receive the data transfer? As we learned in the previous chapter, the action is always put on the `ValueStack` at the start of a request-processing cycle. The model, as exposed by the `ModelDriven` interface, is moved onto the `ValueStack` by the `modelDriven` interceptor, discussed later in this chapter.

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

STATIC-PARAMS (DEFAULTSTACK)

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

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

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

AUTOWIRING

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

SERVLET-CONFIG (DEFAULTSTACK)

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

- `ServletContextAware`—Sets the `ServletContext`
- `ServletRequestAware`—Sets the `HttpServletRequest`
- `ServletResponseAware`—Sets the `HttpServletResponse`

- `ParameterAware`—Sets a map of the request parameters
- `RequestAware`—Sets a map of the request attributes
- `SessionAware`—Sets a map of the session attributes
- `ApplicationAware`—Sets a map of application scope properties
- `PrincipalAware`—Sets the `Principal` object (security)

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

FILEUPLOAD (DEFAULTSTACK)

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

4.3.3 Workflow interceptors

The interceptors we've covered so far mostly realize some concrete task, such as measuring execution time or transferring some data. Workflow interceptors provide something else entirely. They provide the opportunity to conditionally alter the workflow of the request processing. By *workflow* we mean the path of the processing as it works its way down through the interceptors, through the action and result, and then back out the interceptors. In normal workflow, the processing will go all the way down to the action and result before climbing back out. Workflow interceptors are interceptors that inspect the state of the processing and conditionally intervene and alter this normal path, sometimes only slightly, and sometimes quite drastically.

WORKFLOW (DEFAULTSTACK)

One of the interceptors is actually named `workflow`. Consider this one to be the gold standard for what a workflow-oriented interceptor can do. We've already used and discussed this interceptor. As we've learned, it works with our actions to provide data validation and subsequent workflow alteration if a validation error occurs. Since we've already used this interceptor, we'll leverage our familiarity to learn more about how interceptors work by looking at the code that alters execution workflow. Listing 4.1 shows the code from this important interceptor.

Listing 4.1 Altering workflow from within an interceptor

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

  Action action = invocation.getAction();          (1)

  if (action instanceof Validateable) {
    Validateable validateable = (Validateable) action;
      validateable.validate();                     (2)
  }
  if (action instanceof ValidationAware) {
    ValidationAware validationAwareAction =
    ValidationAware) action;                       (3)

    if (validationAwareAction.hasErrors()) {
      return Action.INPUT;                         (4)
    }
  }
  return invocation.invoke();
}
```

If you recall, the actions of our Struts 2 Portfolio use a form of validation implemented in a couple of interfaces that cooperate with the `workflow` interceptor. The action implements these interfaces to expose methods upon which the interceptor will work. First, the interceptor must obtain an instance of the action from the `ActionInvocation` ❶ so that it can check to see whether the action has implemented these interfaces. If the action has implemented the `Validateable` interface, the interceptor will invoke its `validate()` method ❷ to execute the action's validation logic.

Next, if the action implements the `ValidationAware` interface, the interceptor will check to see whether any errors were created by the validation logic by calling the `hasErrors()` method ❸. If some are present, the `workflow` interceptor takes the rather drastic step of completely halting execution of the action. It does this, as you can see, by returning its own `INPUT` control string ❹. Further execution stops immediately. The `INPUT` result is rendered, and postprocessing occurs as control climbs back out of the interceptor stack. Note that our Struts 2 Portfolio actions all inherit implementations of these interfaces by extending the `ActionSupport` convenience class.

The `workflow` interceptor also introduces another important interceptor concept: using `params` to tweak the execution of the interceptor. After we finish covering the built-in interceptors, we'll cover the syntax of declaring your interceptors and interceptor stacks in section 4.4.3. At that time, we'll learn all about setting and overriding parameters. For now, we'll just note the parameters that an interceptor can take. The `workflow` interceptor can take several parameters:

- `alwaysInvokeValidate` (true or false; defaults to true, which means that `validate()` will be invoked)
- `inputResultName` (name of the result to choose if validation fails; defaults to `Action.INPUT`)
- `excludeMethods` (names of methods for which the `workflow` interceptor shouldn't execute, thereby omitting validation checking for a specific entry point method on an action)

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

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

This list of exclude methods is meant to support actions that expose methods other than `execute()` for various processing tasks related to the same data object. We haven't shown how to do this yet, but we will in chapter 15. For instance, imagine you want to use the same action to prepopulate as well as process a form. A common scenario is to combine the create, read, update, and delete (CRUD) functions pertaining to a single data object into a single action. For now, note that the main benefit of such a strategy is the consolidation of code that pertains to the same domain object. One difficulty with this strategy is that the process that prepopulates the form can't be validated because there is no data yet. To handle this problem, we can put the prepopulation code into an `input` method and list this method as one that should be excluded from the `workflow` interceptor's validation measurements.

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

VALIDATION (DEFAULTSTACK)

We've already shown one basic form of validation offered by Struts 2. To recap that technique, the `Validateable` interface, as we've seen, provides a programmatic validation mechanism; you put the validation code into your action's `validate()` method and it'll be executed by the `workflow` interceptor. The `validation` interceptor, on the other hand, is part of the Struts 2 validation framework and provides a declarative means to validate your data. Rather than writing validation code, the validation framework allows you to use both XML files and Java annotations to describe the validation rules for your data. Since the validation framework is such a rich topic, chapter 10 is dedicated to it.

For now, we should note that the `validation` interceptor, like the `Validateable` interface, works in tandem with the `workflow` interceptor. Recall that the `workflow` interceptor calls the `validate()` method of the `Validateable` interface to execute validation code before it checks for validation errors. In the case of the validation framework, the `validation` interceptor itself executes the validation logic. The `validation` interceptor is the entry point into the validation framework's processing. When the validation framework does its work, it'll store validation errors using the same `ValidationAware` methods that your handwritten `validate()` code does. When the `workflow` interceptor checks for error messages, it doesn't know whether they were created by the validation framework or the validation code invoked through the `Validateable` interface. In fact, it doesn't matter. The only thing that really matters is that the `validation`

interceptor fires before the `workflow` interceptor, and this sequencing is handled by the `defaultStack`. You could even use both methods of validation if you liked. Either way, if errors are found, the `workflow` interceptor will divert workflow back to the input page.

PREPARE (DEFAULTSTACK)

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

The `prepare` interceptor is flexible as well. For instance, you can define special prepare methods for the different execution methods on a single action. As we said earlier, sometimes you'll want to define more than one execution entry point on your action. (See the CRUD example in chapter 15 for details.) In addition to the `execute()` method, you might define an `input()` method and an `update()` method. In this case, you might want to define specific preparation logic for each of these methods. If you've implemented the `Preparable` interface, you can also define preparation methods, named according to the conventions in table 4.1, for each of your action's execution methods.

Table 4.1 Parameters to the `prepare` interceptor

Action Method Name	Prepare Method 1	Prepare Method 2
`input()`	`prepareInput()`	`prepareDoInput()`
`update()`	`prepareUpdate()`	`prepareDoUpdate()`

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

```
alwaysInvokePrepare - Default to true.
```

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

MODELDRIVEN (DEFAULTSTACK)

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

4.3.4 Miscellaneous interceptors

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

EXCEPTION (DEFAULTSTACK)

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

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

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

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

First, we define a global result. We need to do this because this error page isn't specific to one action, and global results are available to all actions in the package. The `exception-mapping` element tells the `exception` interceptor which result to render for a given exception. When the `exception` interceptor executes during its postprocessing phase, it'll catch any exception that has been thrown and map it to a result. Before yielding control to the result, the `exception` interceptor will create an

`ExceptionHolder` object and place it on top of the `ValueStack`. The `ExceptionHolder` is a wrapper around an exception that exposes the stack trace and the exception as JavaBeans properties that you can access from a tag in your error page. The following snippet shows our error JSP page:

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

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

As you can see, our `property` tag, explained with the other Struts 2 tags in chapter 6, references the `exceptionStack` property that has been placed on the `ValueStack`. We've created an `ErrorProne` action, which automatically throws an exception, so we can see all this in action. For a guaranteed application failure, hit the error-prone link from the home page of the chapter 4 version of the application. Note that you don't need to have one page catch all exceptions; you can have as many exception mappings as you like, mapping specific exception types to a variety of specific results.

TOKEN AND TOKEN-SESSION

The `token` and `token-session` interceptors can be used as part of a system to prevent duplicate form submissions. Duplicate form posts can occur when users click the Back button to go back to a previously submitted form and then click the Submit button again, or when they click Submit more than once while waiting for a response. The token interceptors work by passing a token in with the request that is checked by the interceptor. If the unique token comes to the interceptor a second time, the request is considered a duplicate. These two interceptors both do the same thing, differing only in how richly they handle the duplicate request. You can either show an error page or save the original result to be rerendered for the user. We'll implement this functionality for the Struts 2 Portfolio application in chapter 15.

SCOPED-MODELDRIVEN (DEFAULTSTACK)

This nice interceptor supports wizard-like persistence across requests for your action's model object. This one adds to the functionality of the `modelDriven` interceptor by allowing you to store your model object in, for instance, session scope. This is great for implementing wizards that need to work with a data object across a series of requests.

EXECANDWAIT

When a request takes a long time to execute, it's nice to give the user some feedback. Impatient users, after all, are the ones who make all those duplicate requests. While the token interceptors discussed earlier can technically solve this problem, we should still do something for the user. The `execAndWait` interceptor helps prevent your users from getting antsy. We'll implement this functionality for the Struts 2 Portfolio application in chapter 15.

In addition to providing these useful interceptors, the `struts-default` package also provides several built-in stacks made of these interceptors. We've already seen the `strutsDefault` stack, but we should look at what else is available.

4.3.5 Built-in stacks

The Struts 2 framework comes with many built-in stacks that provide convenient arrangements of the built-in interceptors. We've been using one of these, the `defaultStack`, for all of our Struts 2 Portfolio packages. We inherit all of this, and other built-in stacks, just by having our packages extend the `struts-default` package defined in struts-default.xml. To make things simple, we recommend that you also use the `defaultStack` unless you have a clear imperative to do otherwise. Most of the other built-in stacks you could use are just pared-down versions of the `defaultStack`. This paring down is not done to make more efficient versions of the stacks by eliminating unnecessary interceptors. Rather, the smaller stacks are meant to be modular building blocks for building larger ones. If you find yourself building your own stacks, try to use these modular pieces to help simplify your task. Still, you might ask, why do we need the `scoped-modelDriven` interceptor in the stack if we aren't using it? Isn't this a performance hit? We think it isn't. So far, it seems that unused interceptors don't affect performance that much. Additionally, messing with the interceptors can be the fastest way to introduce debugging complexity. Ultimately, we always recommend using the built-in path of least resistance as long as possible—which means the `defaultStack`. While Struts 2 is flexible, it's also meant to perform well and be useful right out of the box.

4.4 Declaring interceptors

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

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

4.4.1 Declaring individual interceptors and interceptor stacks

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

Listing 4.2 Interceptor declarations from the `struts-default` package

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

  . . .
  <interceptors>
    <interceptor name="execAndWait" class="ExecuteAndWaitInterceptor"/>
    <interceptor name="exception" class="ExceptionMappingInterceptor"/>
    <interceptor name="fileUpload" class="FileUploadInterceptor"/>
    <interceptor name="i18n" class="I18nInterceptor"/>
    <interceptor name="logger" class="LoggingInterceptor"/>
    <interceptor name="modelDriven" class="ModelDrivenInterceptor"/>
    <interceptor name="scoped-modelDriven" class= . . ./>
    <interceptor name="params" class="ParametersInterceptor"/>
    <interceptor name="prepare" class="PrepareInterceptor"/>
    <interceptor name="static-params" class=. . ./>
    <interceptor name="servlet-config" class="ServletConfigInterceptor"/>
    <interceptor name="sessionAutowiring"
                 class="SessionContextAutowiringInterceptor"/>
    <interceptor name="timer" class="TimerInterceptor"/>
    <interceptor name="token" class="TokenInterceptor"/>
    <interceptor name="token-session" class= . . . />
    <interceptor name="validation" class= . . . />
    <interceptor name="workflow" class="DefaultWorkflowInterceptor"/>
       . . .
    <interceptor-stack name="defaultStack">
      <interceptor-ref name="exception"/>
      <interceptor-ref name="alias"/>
      <interceptor-ref name="servlet-config"/>
      <interceptor-ref name="prepare"/>
      <interceptor-ref name="i18n"/>
      <interceptor-ref name="chain"/>
      <interceptor-ref name="debugging"/>
      <interceptor-ref name="profiling"/>
      <interceptor-ref name="scoped-modelDriven"/>
      <interceptor-ref name="modelDriven"/>
      <interceptor-ref name="fileUpload"/>
      <interceptor-ref name="checkbox"/>
      <interceptor-ref name="static-params"/>
      <interceptor-ref name="params">
        <param name="excludeParams">dojo\..*</param>
      </interceptor-ref>
      <interceptor-ref name="conversionError"/>
      <interceptor-ref name="validation">
         <param name="excludeMethods">input,back,cancel,browse</param>
      </interceptor-ref>
      <interceptor-ref name="workflow">
        <param name="excludeMethods">input,back,cancel,browse</param>
      </interceptor-ref>
    </interceptor-stack>

  </interceptors>

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

</package>
```

1 **Interceptors element**

2 **All interceptor elements**

3 **Declaring a stack**

4 **Interceptor references**

5 **Parameters**

6 **Default reference**

The `interceptors` element ❶ contains all the `interceptor` and `interceptor-stack` declarations of the package. Interceptor stacks are just a convenient way of referencing a sequenced chunk of interceptors by name. Each `interceptor` element ❷ declares an interceptor that can be used in the package. This just maps an interceptor implementation class to a logical name, such as mapping `com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor` to the name `workflow`. (In listing 4.2, we've snipped the package names to make the listing more readable.) These declarations don't actually create an interceptor or associate that interceptor with any actions; they just map a name to a class.

Now we can define some stacks of interceptors ❸. Since most actions will use the same groups of interceptors, arranged in the same sequence, it's common practice to define these in small, building-block stacks. The `struts-default` package declares several stacks, most importantly the `defaultStack`.

The contents of the `interceptor-stack` element are a sequence of `interceptor-ref` elements. ❹ These references must all point to one of the logical names created by the interceptor elements. Creating your own stacks, as we'll see when we build a custom interceptor later in this chapter, is just as easy. The `interceptor-ref` elements can also pass in parameters to configure the instance of the interceptor that is created by the reference ❺.

Finally, a package can declare a default set of interceptors. ❻ This set will be associated with all actions in the package that don't explicitly declare their own interceptors. The `default-interceptor-ref` element simply points to a logical name, in this case the `defaultStack`. This important line is what allows our actions to inherit a default set of interceptors when we extend the `struts-default` package.

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

XML DOCUMENT STRUCTURE

Before moving on to show how you specify the interceptors that'll fire for your specific actions, we should make a point about the sequence of elements within the XML documents we use for declarative architecture. These XML documents must conform to certain rules of ordering. For instance, each `package` element contains precisely one `interceptors` element, and that element must come in a specific position in the document. The complete DTD, struts-2.0.dtd, can be found on the Struts 2 website. For now, note the following snippet from the DTD, which pertains to the structure of listing 4.2:

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

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

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

4.4.2 Mapping interceptors to actions

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

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

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

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

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

```
    <result>Success.jsp</result>
</action>
```

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

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

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

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

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

4.4.3 Setting and overriding parameters

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

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

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

```
<action name="YourAction" class="org.actions.youractions.YourAction">

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

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

Next up, rolling your own authentication interceptor!

4.5 Building your own interceptor

We've said several times that you probably won't need to build your own interceptor. On the other hand, we hope that we've sold the power of interceptors well enough to get you itching to start rolling your own. Apart from the care needed when sequencing the stack and learning to account for this sequencing in your debugging, interceptors can be simple to write. We round out the chapter by creating an authentication interceptor that we can use to provide application-based security for our Struts 2 Portfolio application. This form of authentication is probably far too simple for most real applications, but it's a well-known use case and serves as a perfect example for interceptors.

We'll start by looking at the technical details of implementing an interceptor.

4.5.1 Implementing the Interceptor interface

When you write an interceptor, you'll implement the `com.opensymphony.xwork2.interceptor.Interceptor` interface:

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

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

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

4.5.2 Building the AuthenticationInterceptor

The authentication interceptor will be simple. If you recall the three phases of interceptor processing—preprocessing, calling `ActionInvocation.invoke()`, and postprocessing—you can anticipate how our `AuthenticationInterceptor` will function. When a request comes to one of our secure actions, we'll want to check whether the

request is coming from an authenticated user. This check is made during preprocessing. If the user has been authenticated, the interceptor will call `invoke()`, thus allowing the action invocation to proceed. If the user hasn't been authenticated, the interceptor will return a control string itself, thus barring further execution. The control string will route the user to the login page.

You can see this in action by visiting the chapter 4 version of the Struts 2 Portfolio application. On the home page, there's a link to add an image without having logged in. The add image action is a secure action. Try clicking the link without having logged in. You'll be automatically taken to the login page. Now, log in and try the same link again. The application comes with a default user, username = "Arty" and password = "password". This time you're allowed to access the secure add image action. This is done by a custom interceptor that we've placed in front of all of our secure actions. Let's see how it works.

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

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

Listing 4.3 The `Login` action authenticates the user and stores the user in session scope

```
public class Login extends ActionSupport implements SessionAware {      (1)

  public String execute(){

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

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

    return SUCCESS;
  }

  . . .

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

The first thing of interest is that our `Login` action uses the `SessionAware` interface ❶ to have the session-scoped map conveniently injected into a setter ❺. This is one of the services provided by the `ServletConfigInterceptor` provided in the `defaultStack`. (See the section on that interceptor earlier in this chapter to find out all the other objects you can have injected through similar interfaces.) As for the business logic of the login itself, first we use our service object to authenticate the username and password combination ❷. Our authentication method will return a valid `User` object if everything checks out, or null if it doesn't. If the user is null, we send her back to the `INPUT` result, which is the login form ❸ that she came from. If the user is not null, we'll store the user object in the session map ❹, officially marking her as an authenticated user.

With the `Login` action in place, we can look at how the `AuthenticationInterceptor` protects secure actions from unauthenticated access. Basically, the interceptor will check to see whether the user object has been placed in the session map. Let's check it out. Listing 4.4 shows the full code.

Listing 4.4 Inspecting the heart of the `AuthenticationInterceptor`

```
public class AuthenticationInterceptor implements Interceptor {

  public void destroy() {
  }

  public void init() {
  }

  public String intercept( ActionInvocation actionInvocation )
                              throws Exception{

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

    if (user == null) {
      return Action.LOGIN;
    }
    else {

      Action action = ( Action ) actionInvocation.getAction();

      if (action instanceof UserAware) {
         ((UserAware)action).setUser(user);
      }

      return actionInvocation.invoke();
    }
  }

}
```

Implements interceptor

Empty implementations

❶

❷

❸

❹

❺ Continue action invocation

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

If the user object is null ❸, then the user hasn't been authenticated through the login action. At this point, we return a result string, without allowing the action to continue. This result string, `Action.LOGIN`, points to our login page. If you consult the chapterFour.xml file, you'll see that the `chapterFourSecure` package defines the login result as a global result, available to all actions in the secure package. In chapter 8, we'll learn about configuring global results.

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

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

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

Now we'll apply this interceptor to our secure actions. Since we put all of our secure actions into a single package, we can build a custom stack that includes our `AuthenticationInterceptor`, and then declare that as the default interceptor reference for the secure package. This is the benefit of packaging actions according to

shared functionality. Listing 4.5 shows the elements from chapterFour.xml that configure the `chapterFourSecure` package.

Listing 4.5 Declaring our interceptor and building a new default stack

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

  <interceptors>                                          ❶

    <interceptor name="authenticationInterceptor"         ❷
      class="manning.utils.AuthenticationInterceptor"/>

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

  </interceptors>

  <default-interceptor-ref name="secureStack"/>           ❹

   . . .
</package>
```

With all of our secure actions bundled in this package, we just need to make a stack that includes our `AuthenticationInterceptor` and then declare it as the default. You can see how easy this is. First, we must have an `interceptors` element ❶ to contain our `interceptor` and `interceptor-stack` declarations. We have to map our Java class to a logical name with an `interceptor` element ❷. We've chosen `authenticationInterceptor` as our name. Next, we build a new stack that takes the `defaultStack` and adds our new interceptor to the top of it ❸. We put it on top because we might as well stop an unauthenticated request as soon as possible. Finally, we declare our new `secureStack` as the default stack for the package ❹. Note that the `default-interceptor-ref` element isn't contained in the `interceptors` element; it doesn't declare any interceptors, it just declares the default value for the package. Every action in this package will now have authentication with automatic routing back to the login page, as well as injection of the user object for any action that implements the `UserAware` interface. It feels like we've accomplished something, no? The best part is that our interceptor is completely separate from our action code and completely reusable.

4.6 Summary

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

By now, you should have come to grips with the role of the interceptor in the framework. To reiterate, the interceptor component provides a nice place to separate the logic of various cross-cutting concerns into layered, reusable pieces. Tasks such as

logging, exception handling, and dependency injection can all be encapsulated in interceptors. With the functionality of these common tasks thus modularized, we can easily use the declarative architecture to customize stacks of interceptors to meet the needs of our specific actions or packages of actions.

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

Interceptors themselves are invoked via their `intercept()` method. The execution of an interceptor can be broken into three phases: preprocessing, passing control on to the rest of the action invocation by calling `invoke()`, and postprocessing. Interceptors can also divert workflow by returning a control string instead of calling `invoke()`. They also have access to all key data via the `ActionInvocation` instance they receive. Ultimately, interceptors can do just about anything.

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

Now that we've covered actions and interceptors, we should be ready to move on to the view layer and start exploring the rich options that the framework offers for rendering result pages. Before we do that, we have one more stop on our tour of the core components of framework. Most likely, it's a stop you've been wondering about. Next up, chapter 5 will work on dispelling that mysterious OGNL cloud that surrounds the data transfer mechanisms of the framework.