

STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

 HANNING

Chapter 3. Working with Struts 2 actions.....	1
Section 3.1. Introducing Struts 2 actions.....	2
Section 3.2. Packaging your actions.....	4
Section 3.3. Implementing actions.....	10
Section 3.4. Transferring data onto objects.....	20
Section 3.5. File uploading: a case study.....	25
Section 3.6. Summary.....	30



Working with Struts 2 actions

This chapter covers

- Bundling actions into packages
- Implementing actions
- Introducing object-backed properties and `ModelDriven` actions
- Uploading files

With the overviews and introductions behind us, it's time to study the core components of Struts 2. First up, actions. As we've learned, actions do the core work for each request. They contain the business logic, hold the data, and then select the result that should render the result page. This is an action-oriented framework; actions are at its heart. In the end, you'll spend much of your time as a Struts 2 developer working with actions.

This chapter will give you everything you need to start building your application's actions. Using the XML-based mechanism for declarative architecture, we'll explore all of the options available to us when we declare our actions, see some convenience classes that aid development of actions, and cover the most common ways of carrying

data in the action. The previous chapter used JavaBeans properties. We'll also see how interceptors work together with actions to provide much of the framework's functionality. In fact, we'll end with a useful case study of a file upload action. We will also start to develop our full-featured sample application, the Struts 2 Portfolio, to help demonstrate the concepts and techniques of this chapter.

As it turns out, we can't show you actions without introducing a fair amount of other material. We'll try to keep the focus on actions as much as possible. But you'll probably pick up enough of the other stuff to get your own simple Struts 2 application up and running. There's a lot to learn in this chapter. If you're ready, let's go.

3.1 *Introducing Struts 2 actions*

To set the stage, we'll start with a sketch of the role that actions play in the framework. We'll explain the purpose and various roles of the action component. We'll contrast the Struts 2 action with the similarly named component in Struts 1. And we'll study the obligations that an object serving in the role of an action has toward the framework in general. Struts 2 is an egalitarian enterprise. Any class can be an action as long as it satisfies its obligations to the framework. Let's find out what these important components do for the framework.

3.1.1 *What does an action do?*

Actions do three things. First, as you probably understand by now, an action's most important role, from the perspective of the framework's architecture, is encapsulating the actual work to be done for a given request. The second major role is to serve as a data carrier in the framework's automatic transfer of data from the request to the view. Finally, the action must assist the framework in determining which result should render the view that'll be returned in the request response. Let's see how the action component fulfills each of these various roles.

By the way, we're going to demonstrate our points in the coming paragraphs with examples from the HelloWorld application from chapter 2. But don't worry; we'll start building the real-world Struts 2 Portfolio in a few pages.

ACTIONS ENCAPSULATE THE UNIT OF WORK

Earlier in this book, we saw that the action fulfills the role of the MVC model for the framework. One of the central responsibilities of this role is the containment of business logic; actions use the `execute()` method for this purpose. The code inside this method should concern itself only with the logic of the work associated with the request. The following code snippet, from the previous chapter's HelloWorld application, shows the work done by the HelloWorldAction.

```
public String execute() {  
    setCustomGreeting( GREETING + getName() );  
    return "SUCCESS";  
}
```

The action's work is to build a customized greeting for the user. As we can see, this action's `execute()` method does little else than build this greeting. In this case, the

business logic amounts to little more than a concatenation. If it were much more complex, we'd probably have bumped that logic out to a business component and injected that component into the action. The use of dependency injection, which helps keep code such as actions clean and decoupled, is supported by the framework. We'll learn some techniques that utilize the framework's Spring integration for injecting these components later in the book. For now, just keep in mind that our actions hold the business logic, or at least the entry point to the business logic, and they should keep that logic as pure and brief as possible.

ACTIONS PROVIDE LOCUS FOR DATA TRANSFER

Being the model component of the framework also means that the action is expected to carry the data around. While you might think this would make actions more complicated, it actually makes them cleaner. Since the data is held local to the action, it's always conveniently available during the execution of the business logic. There might be a bunch of JavaBeans properties adding lines of code to the action, but when the `execute()` method references the data in those properties, the proximity of the data makes that code all the more succinct.

Listing 3.1, also from `HelloWorldAction`, shows the code that allows that action to carry request data.

Listing 3.1 Transferring request data to the action's JavaBeans properties

```
private String name;

public String getName() {
    return name;
}

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

private String customGreeting;

public String getCustomGreeting()
{
    return customGreeting;
}

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

The action merely implements JavaBeans properties for each piece of data that it wishes to carry. We saw this in action with the `HelloWorld` application. Request parameters from the form are moved to properties that have matching names. As we saw, the framework does this automatically. In this case, the `name` parameter from the `name` collection form will be set on the `name` property. In addition to receiving the incoming data from the request, these JavaBeans properties on the action will also expose the data to the result. The `HelloWorld` action's logic sets the custom greeting on the `customGreeting` property, which makes it available to the result as well.

In addition to these simple JavaBeans properties, there are a couple of other techniques for using the action as a data transfer object. We'll examine these alternatives later in this chapter, and will also examine the mechanisms by which the actual data transfer occurs. For now, we just want to recognize that the action serves as a centralized data transfer object that can be used to make the application data available in all tiers of the framework.

The use of actions as data transfer objects should probably ring some alarms in the minds of alert Struts 1 developers. In Struts 1, there's only one instance of a given action class. If this were still true, we couldn't use the action object itself as a data carrier for the request. In a multithreaded environment, such as a web application, it'd be problematic to store data in instance fields as we've seen. Struts 2 solves this problem by creating a new instance of an action for each request that maps to it. This fundamental difference allows Struts 2 objects to exist as dedicated data transfer objects for each request.

ACTIONS RETURN CONTROL STRING FOR RESULT ROUTING

The final duty of an action component is to return a control string that selects the result that should be rendered. Previous frameworks passed routing objects into the entry method of the action. Returning a control string eliminates the need for these objects, resulting in a cleaner signature and an action that is less coupled to specific routing code. The value of the return string must match the name of the desired result as configured in the declarative architecture. For instance, the `HelloWorldAction` returns the string "SUCCESS". As you can see from our XML declaration, SUCCESS is the name of the one of the result components.

```
<action name="HelloWorld" class="manning.chapterOne.HelloWorld">
  <result name="SUCCESS">/chapterTwo/HelloWorld.jsp</result>
  <result name="ERROR">/chapterTwo/Error.jsp</result>
</action>
```

The `HelloWorld` application has a simple logic for determining which result it will choose. In fact, it'll always choose the "SUCCESS" result. Most real-world actions will have a more complex determination process, and the result choices will almost always include some sort of error result to handle problems that might occur during the action's interaction with the model. Regardless of the complexity, actions must ultimately return a string that maps to one of the result components available for rendering the view for that action.

You should now realize what an action does, but before we design one, we need to create the packages to contain them. In the next section, we'll see how to organize our actions into packages and take our first glimpse at the Struts 2 Portfolio application, the main sample application for this book.

3.2 Packaging your actions

Whether you declare your action components with XML or Java annotations, when the framework creates your application's architecture, it'll organize your actions and

other components into logical containers called *packages*. Struts 2 packages are similar to Java packages. They provide a mechanism for grouping your actions based on commonality of function or domain. Many important operational attributes, such as the URL namespace to which actions will be mapped, are defined at the package level. And, importantly, packages provide a mechanism for inheritance, which among other things allows you to inherit the components already defined by the framework. In this section, we'll check out the details of Struts 2 packages by examining the Struts 2 Portfolio application's packaging.

First, we'll give a quick summary of our sample application's functionality and purpose in order to set the stage for factoring our actions into separate packages.

3.2.1 The Struts 2 Portfolio application

Throughout this book we'll develop and examine a sample application called the Struts 2 Portfolio. Artists can use the application to create an online portfolio of their work. The portfolio is a simple gallery of images. Artists must first register with the system to have a portfolio. It's free, but we'll collect some harmless personal information. Once the artist has an account, she can log in to the secure portion of the application to conduct such sensitive business as creating new portfolios, as well as adding and deleting images from those portfolios. The other side of the portfolio is the public side. A visitor to the public site can view the images in any of the portfolios. This public face of the portfolio won't be protected by security.

While this application is simple, it has enough complexity to demonstrate the core Struts 2 concepts, including packaging strategies. A quick analysis of our requirements tells us that we have two distinct regions in our web application. We have some functions that anyone can use, such as registering for accounts or viewing portfolios, and we have some secure functions, primarily account administration. Ultimately, these functionalities will be implemented with actions, and we can bet that the secure actions will have different requirements than nonsecure actions. Let's see how we can use Struts 2 packages to group our actions into secure and nonsecure packages.

3.2.2 Organizing your packages

It's up to you to decide on an organizational theme for your application's package space. We'll organize the Struts 2 Portfolio's packages based upon commonality of functionality, a common strategy. The important thing is to see how the packages can be declared and configured to achieve a given organizational structure. As noted earlier, you can declare your application's architectural components with XML files or with Java annotations located in your action class files. We also noted that we'll use XML files for our sample code. With that in mind, let's take a look at the chapterThree.xml file that declares the components for our first cut of the Struts 2 Portfolio application. You can find this file in /WEB-INF/classes/manning/chapterThree. This XML file contains declarations of two packages, one for public actions and one for secure actions. Listing 3.2 shows the declaration of the secure package.

```
<package name="chapterThreeSecure" namespace="/chapterThree/secure"
```

Listing 3.2 Declaration of a package

```

    extends="struts-default">
<action name="AdminPortfolio" >
    <result>/chapterThree/AdminPortfolio.jsp</result>
</action>

<action name="AddImage" >
    <result>/chapterThree/ImageAdded.jsp</result>
</action>

<action name="RemoveImage" >
    <result>/chapterThree/ImageRemoved.jsp</result>
</action>

</package>

```

The package declared in listing 3.2 contains all of the secure actions for the application. These actions require user authentication. A glance at the names of these actions should be sufficient to give a good idea of their functional purpose. Obviously, the actions that add and delete images from a portfolio should be secured behind authentication. We want to make sure that the user who's removing images from the portfolio actually owns that portfolio. Grouping these together allows us to share declarations of components that might be useful to our authentication mechanism. Additionally, we've chosen to give a special URL namespace to these secure actions. We want our users to notice from the URL that they have entered a secure region of the website.

Now, let's look at the package declaration itself. You can only set four attributes on a package: name, namespace, extends, and abstract. Table 3.1 summarizes these attributes.

Table 3.1 The attributes of the Struts 2 package

Attribute	Description
name (required)	Name of the package
namespace	Namespace for all actions in package
extends	Parent package to inherit from
abstract	If true, this package will only be used to define inheritable components, not actions

While it may be challenging to choose the strategy by which you divide your actions into packages, declaring them is simple. The only required attribute is the name. The name attribute is merely a logical name by which you can reference the package. In listing 3.2, we've named our package `chapterThreeSecure`. This, like all good names, indicates the purpose of this package: it contains the secure actions of the chapter 3 version of the Struts 2 Portfolio sample application.

Next, we set the namespace attribute to `/chapterThree/secure`. As we've seen, the namespace attribute is used to generate the URL namespace to which the actions of these packages are mapped. In the case of the `AddImage` action from listing 3.2, the URL will be built as follows: <http://localhost:8080/manningHelloWorld/chapterThree/secure/AddImage.action>

When a request to this URL arrives, the framework consults the `/chapterThree/secure` namespace for an action named `AddImage`. Note that you can give the same namespace to more than one package. If you do this, the actions from the two packages map to the same namespace. This isn't necessarily a problem. You might choose to separate your actions into separate packages for some functional reason that doesn't warrant a distinct namespace. In our case, we decide that we want the user to see a URL namespace change when he enters the secure region of the application.

NOTE *The Struts 2 Portfolio sample application*—We should make a few comments about the structure of the sample application. All of the sample code that we'll develop in this book comes in a single WAR file, `Struts2InAction.war`; it's one big web application in other words. (Recall that we also provided a skeletal repackaging of the `HelloWorld` example as a standalone web application, but that's just a bonus to help you see what a minimal Struts 2 web application looks like.) Inside the application are many "sub-applications," if you will. For instance, each chapter has its own version of the Struts 2 Portfolio. We've used the Struts 2 packaging mechanism to isolate these versions from one another. They all have their own namespaces and Struts 2 XML files. Not only does this allow us to offer versions of the Struts 2 Portfolio that focus on the specific goals of each chapter, thus decreasing the learning curve, it also serves to further demonstrate the usefulness of packages.

If you don't set the namespace attribute, your actions will go into the default namespace. The default namespace sits beneath all of the other namespaces waiting to resolve requests that don't match any explicit namespace. Consider the following: <http://localhost:8080/manningHelloWorld/chapterSeventy/secure/AddImage.action>

If this request arrives at our sample web application, the framework will attempt to locate the `/chapterSeventy/secure` namespace. As this namespace doesn't exist, the `AddImage` action won't be found in it. As a last resort, the framework will search the default namespace for the `AddImage` action. If it's found there, the URL resolves and the request is serviced. Note that the default namespace is actually the empty string `"`. You can also define a root namespace such as `/`. The root namespace is treated as all other explicit namespaces and must be matched. It's important to distinguish between the empty default namespace, which can catch all request patterns as long as the action name matches, and the root namespace, which is an actual namespace that must be matched.

The next attribute that we set in listing 3.2 is `extends`. This important attribute names another package whose components should be inherited by the current package. This is similar to the `extends` keyword in Java. If you think of the named package

as being the superclass of your current package, you'll understand that the current package will inherit all the members of the superclass package. Furthermore, the current package can override members of the superclass package. Package inheritance plays a particularly important role in the use of the intelligent defaults we've been touting. Most of the intelligent defaults are defined in a built-in package called `struts-default`. You can inherit and use the components defined in that package by making your packages extend `struts-default`. But what exactly do you inherit?

3.2.3 Using the components of the `struts-default` package

Using the intelligent default components of the `struts-default` package is easy. You only need to extend that package when creating your own packages. Our chapter-ThreeSecure package does just this. By definition, an intelligent default shouldn't require that a developer do anything manually. Indeed, once you extend this package, many of the components automatically come into play. One good example is the default interceptor stack, which we've already used in the HelloWorld application. How did we use it?

DEFINITION The `struts-default` package, defined in the system's `struts-default.xml` file, declares a huge set of commonly needed Struts 2 components ranging from complete interceptor stacks to all the common result types.

Here's the secret. Most of the interceptors that you'll ever need are found in the `struts-default` package that's declared in the `struts-default.xml` file. You can think of this important file as the framework's own declarative architecture artifact. The `struts-default` package that it defines contains common architectural components that all developers can reuse simply by having their own packages extend it. If you want to see the whole file, it can be found at the root level of the distribution's main JAR file, `struts2-core.jar`. Listing 3.3 shows the elements from this file that declare the default interceptor stack that will be used by most applications.

Listing 3.3 The `struts-default` package declares many commonly used components

```
<package name="struts-default">
. . .
<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-model-driven"/>
  <interceptor-ref name="model-driven"/>
  <interceptor-ref name="fileUpload"/>
```

1 Package element

2 Declares defaultStack interceptor stack

```

<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>

. . .

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

```

← **3** Sets the defaultStack as default stack

Inside the package element ❶, an interceptor stack is declared ❷. It's named, appropriately, `defaultStack`. Near the end of the listing, you can see that this stack is declared as the default interceptor stack for this package ❸, as well as any other packages that extend the `struts-default` package. Throughout the course of the book, we'll see and discuss all of the interceptors in the default stack. (We'll even learn how to write our own interceptors in chapter 4.) For now, we'll just point to one that you should be able to appreciate. Note the interceptor called `params`. If you've been wondering about the mysteriously automatic transfer of data from the request to the action, here's the answer. This important `params` interceptor has been the one moving data from the request parameters to our action's JavaBeans properties. There's no magic going on. The work is getting done with good old-fashioned lines of code. For the curious, these specific lines of code provide good insight into the inner workings of Struts 2; check out the source for `com.opensymphony.xwork2.interceptor.ParametersInterceptor` if you just can't help yourself.

As you can see from the `params` interceptor, much of the core functionality of the framework has been implemented as interceptors. While you're not *required* to extend the `struts-default` package when you create your own packages, omitting this bit of inheritance amounts to rejecting the core of the framework. Consider the package inheritance shown in Figure 3.1.

In this package, we see that the all-important `defaultStack` will be available to our `chapterThreeSecure` and `chapterThreePublic` packages. Figure 3.1 also shows `someOtherPackage`, which doesn't extend `struts-default`. This package starts from ground zero. Without the important interceptors defined in the `defaultStack` of the `struts-default` package, most of the framework's features are missing in action. Not the least of these would be the automatic data transfer we've discussed. Without these features, the framework is bare to say the least. You could always take the time

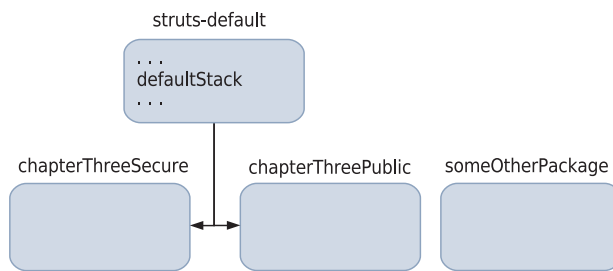


Figure 3.1 Much of the framework's functionality is obtained by extending the `struts-default` package.

to redeclare all of the components that the framework itself declares in the `struts-default.xml` file, but that would be tedious and pointless. Unless you have compelling reasons, you should always extend the `struts-default` package. And, as you'll see when we discuss interceptors in depth in chapter 4, the `struts-default` package declares its components in such a way as to make them flexible as well as reusable. All told, think twice before forgoing the extension of `struts-default`.

Now we have a pretty good idea of the mechanics of organizing our application into packages, as well as how to extend packages such as the built-in `struts-default` package. We've even seen one of the packages from the Struts 2 Portfolio sample application, the `chapterThreeSecure` package. In the end, packages will become something you don't think about much. In fact, once you have your application package structure in place, you won't even work with them much. We're now ready to build our Struts 2 Portfolio actions.

3.3 *Implementing actions*

Now it's time to start developing some actions. In this section, we'll cover all the basics of writing actions for your Struts 2 applications. As examples, we'll show you some of our actions from the chapter 3 version of the Struts 2 Portfolio application. While we'll provide thorough coverage of those actions, there's always more to investigate in the sample application than we have time to explore in the book. Don't hesitate to crack the source code; it's well commented!

Implementing Struts 2 actions is easy. Earlier in this chapter, we saw that the contract between the framework and the classes that back the actions provides a great deal of flexibility. Basically, any class can be an action if it wants. It simply must provide an entry method for the framework to invoke when the action is executed. Let's take a look at how the framework makes it even easier to implement your actions. As a case study, we'll continue our look at the Struts 2 Portfolio by digging into a couple of its own action classes.

NOTE Struts 2 actions don't have to implement the `Action` interface. Any object can informally honor the contract with the framework by simply implementing an `execute()` method that returns a control string.

3.3.1 *The optional Action interface*

Though the framework doesn't impose much in the way of formal requirements on your actions, it does provide an optional interface that you can implement. Implementing

the Action interface costs little and comes with some convenient benefits. Let's see why most developers implement the Action interface when developing their actions, even though they don't have to.

WARNING Struts 2 gives developers both a fast development path built on intelligent defaults and an extremely high degree of flexibility to elegantly solve the most arcane use cases. When learning the framework, it can help to focus on the straightforward solutions supported by the intelligent defaults. Once comfortable with the normal way of handling things, the framework's flexibility will be natural and powerful. Without a good understanding of the straight and narrow, the framework's flexibility can admittedly leave one feeling concerned about which path to follow.

Most actions will implement the `com.opensymphony.xwork2.Action` interface. It defines just one method:

```
String execute() throws Exception
```

Since the framework doesn't make any type requirements, you could just put the method in your class without having your class implement this interface. This is fine. But the Action interface also provides some useful String constants that can be used as return values for selecting the appropriate result. The constants defined by the Action interface are

```
public static final String ERROR    "error"
public static final String INPUT    "input"
public static final String LOGIN    "login"
public static final String NONE     "none"
public static final String SUCCESS  "success"
```

These constants can conveniently be used as the control string values returned by your `execute()` method. The true benefit is that these constants are also used internally by the framework. This means that using these predefined control strings allows you to tap into even more intelligent default behavior.

As an example, consider pass-through actions. Remember the pass-through action we used in the HelloWorld application? We said that it was a best practice to route even simple requests through actions. In the HelloWorld application, we used one of these empty actions to hit our JSP page that presents the form that collects the user's name. Here's the declaration of that action:

```
<action name="Name">
  <result>/chapterOne/NameCollector.jsp</result>
</action>
```

The Name action doesn't specify a class to provide the action implementation because there's nothing to do. We just want to go to the JSP page. Conveniently, the Struts 2 intelligent defaults provide a default action implementation that we inherit if we don't specify one. This default action has an empty `execute()` method that does nothing but automatically return the Action interface's SUCCESS constant as its control string. The framework must use this string to choose a result. Luckily, or maybe not so luckily,

the default name attribute for the result element is also the `SUCCESS` constant. Since our sole result forgoes defining its own name, it inherits this default and is automatically selected by our action. This is the general pattern by which many of the intelligent defaults operate.

But wait; we don't need to implement the `Action` interface ourselves, because the framework provides an implementation we can borrow. Next we'll look at a convenience class that implements this and other helpful interfaces that help you further leverage the out-of-the-box features of the framework.

3.3.2 *The `ActionSupport` class*

In this section, we're going to introduce the `ActionSupport` class, a convenience class that provides default implementations of the `Action` interface and several other useful interfaces, giving us such things as data validation and localization of error messages. This convenience class is a perfect example of the Struts 2 straight and narrow we spoke of a bit earlier. The framework doesn't force you to use this, but it's a good idea to use it when learning the framework. In fact, it's pretty much always a good idea to use it unless you have reason not to.

Following in the tradition of "support" classes, `ActionSupport` provides default implementations of several important interfaces. If your actions extend this class, they automatically gain the use of these implementations. This alone makes this class worth learning. However, the implementations provided by this class also provide a great case study in how to make an action cooperate with interceptors to achieve powerfully reusable solutions to common tasks. In this case, validation and text localization services are provided via a combination of interceptors and interfaces. The interceptors control the execution of the services while the actions implement interfaces with methods that are invoked by the interceptors. This important pattern will become clearer as we work through the details of `ActionSupport` by examining its use in our Struts 2 Portfolio application, which we'll do a couple of pages from now.

BASIC VALIDATION

While Struts 2 provides a rich and highly configurable validation framework, which we'll fully examine in chapter 10, `ActionSupport` provides a quick form of basic validation that will serve well in many cases. Moreover, it's a great case study of how a cross-cutting task such as validation can be factored out of the action's execution logic through the use of interceptors and interfaces. The typical pattern is that the interceptor, while controlling the execution of a given task, may coordinate with the action by invoking methods that it exposes. Usually, these methods are part of a specific interface implemented by that action. In our case, `ActionSupport` implements two interfaces that coordinate with one of the interceptors from the default stack, the `DefaultWorkflowInterceptor`, to provide basic validation. If your package extends the `struts-default` package, thereby inheriting the default interceptor stack, and your action extends `ActionSupport`, thereby inheriting implementation of the two necessary interfaces, then you already have everything you need for clean validation of your data.

Just to make it clear where all of this built-in functionality comes from, we'll show you where these default interceptors are defined and how to ensure that you're inheriting them. Listing 3.4 shows the declaration of the workflow interceptor as it's found in the `struts-default.xml` file.

Listing 3.4 Declaration of DefaultWorkflowInterceptor from `struts-default.xml`

```

. . .
<interceptor name="workflow" 1
class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>
. . .
<interceptor-stack name="defaultStack"> 2
. . .
<interceptor-ref name="params"/> 3
. . .
<interceptor-ref name="workflow">
  <param name="excludeMethods">input,back,cancel,browse</param> 4
</interceptor-ref>
. . .
<interceptor-stack name="defaultStack">
. . .

```

In listing 3.4, we first see the declaration element for the workflow interceptor 1, specifying a name and an implementation class. Note that this is called the workflow interceptor because it will divert the workflow of the request back to the input page if a validation error is found. Next, we see the declaration of the default interceptor stack 2. We haven't included all of the interceptors in this listing. Instead, we'll focus on the interceptors that participate in the validation process. Note that the params interceptor 3 comes before the workflow interceptor 4. The params interceptor will move the request data onto our action object. Then, the workflow interceptor will help us validate that data before accepting it into our model. The workflow interceptor must fire after the params interceptor has had a chance to move the data on to the action object. As with most interceptors, sequence is important.

Now, let's see how this validation actually works. We'll use one of our actions from this chapter's version of the Struts 2 Portfolio, the Register action, to demonstrate. As with the params interceptor, the workflow interceptor seeks to remove the logic of a cross-cutting task, validation in this case, from the action's execution logic. When the workflow interceptor fires, it'll first look for a `validate()` method on the action to invoke. You'll place your validation logic in `validate()`. This method is exposed via the `com.opensymphony.xwork2.Validateable` interface. Technically speaking, `ActionSupport` implements the `validate()` method, but we have to override its empty implementation with our own specific validation logic.

As we've said, we're going to demonstrate the concepts and strategies of this book by developing the Struts 2 Portfolio application. In this section, we'll examine the Register action from the chapter 3 version of that application. Let's look at the entire source of that action class. Listing 3.5 shows the entire source of the Register action, found in the source directory of the sample application at `manning/chapterThree/Register.java`.

Listing 3.5 The Register action provides validation logic in the `validate()` method.

```
public class Register extends ActionSupport { ❶

    public String execute() {
        User user = new User();
        user.setPassword( getPassword() );
        user.setPortfolioName( getPortfolioName() );
        user.setUsername( getUsername() ); ❷

        getPortfolioService().createAccount( user );
        return SUCCESS;
    }

    private String username;
    private String password;
    private String portfolioName;

    public String getPortfolioName() {
        return portfolioName;
    }
    public void setPortfolioName(String portfolioName) {
        this.portfolioName = portfolioName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }

    public void validate() {
        PortfolioService ps = getPortfolioService(); ❸

        if ( getPassword().length() == 0 ) {
            addFieldError( "password", "Password is required." );
        }
        if ( getUsername().length() == 0 ) {
            addFieldError( "username", "Username is required." );
        }
    }
}
```



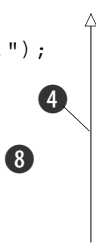

```

        if ( getPortfolioName().length() == 0 ){
            addFieldError( "portfolioName", "Portfolio name is required.");
        }

        if ( ps.userExists(getUsername() ) ){
            addFieldError("username", "This user already exists.");
        }
    }

    public PortfolioService getPortfolioService( ) {
        return new PortfolioService();
    }

```



While we show the entire source here, remember that we're somewhat focused on the validation mechanism offered by `ActionSupport`. With that in mind, we'll skim the nonvalidation parts and then focus on validation. We'll come back to treat the rest in detail over the next few pages. First, note that our action does indeed extend `ActionSupport` ❶. Also, note that we provide an `execute()` ❷ method that contains the business logic, registering a user in this case. After that, we see a set of `JavaBeans` properties ❸. These are common features of actions; they serve to receive the data from the framework's automatic transfer and then carry that data throughout the framework's processing.

But now we're focused on examining the basic validation mechanism provided by `ActionSupport`. As you can see, our action provides a `validate()` method ❹ that contains all of our logic for checking the validity of the data received by our `JavaBeans` properties. This leaves this action's `execute()` method focused on business logic. The validation logic that we've provided is simple. We test each of the three fields to make sure they're not empty by testing the length of each `String` property ❺. If a piece of data doesn't validate, we create and store an error ❻ via methods provided by the `ActionSupport` superclass, such as `addFieldError()`.

We also test that the user doesn't already exist in the system ❼. This test requires a dip into our business logic and data tiers. At this point in the book, the `Struts 2 Portfolio` application uses a simple encapsulation of business logic and data persistence. The `PortfolioService` object is capable of conducting our simple business needs at this stage. In case you're interested, it contains all the business rules in its simple methods, and persists data only in memory. Even our current management techniques are crude; our action just instantiates a `PortfolioService` object ❽ when it needs one. Later in the book, we learn how to integrate with more sophisticated technologies for managing such important resources. For now, this keeps our study of the action component more clear.

What happens if validation fails? If any of the fields are empty, or if the username is already in the system, we call a method that adds an error message. After all the validation logic has executed, control returns to the workflow interceptor. Note that there is no return value on the `validate()` method. The secret, as we'll see, is in the error messages that our validation generates.

Even though control has returned to the workflow interceptor, it's not finished. What does the workflow interceptor do now? Now it's time to earn the "workflow" name. After calling the `validate()` method to allow the action's validation logic to execute, the workflow interceptor will check to see whether any error messages were generated by the validation logic. If it finds errors, then the workflow interceptor will alter the workflow of the request. It'll immediately abort the request processing and return the user back to the input form, where the appropriate error messages will be displayed on the form. Try it out! Fire up your application and open the chapter 3 version of the Struts 2 Portfolio at <http://localhost:8080/Struts2InAction/chapterThree/PortfolioHomePage.action>

Choose to create an account and fill out the form, but omit some data. For instance, we've omitted the password. When we submit the form, the validation fails and diverts the workflow back to the input form again, as seen in figure 3.2.

Figure 3.2 The default workflow interceptor returns us to the input form with validation error messages displayed on the appropriate fields.

Two obvious questions remain. Where were those error messages stored, and how did the workflow interceptor check to see whether any had been created? The `com.opensymphony.xwork2.ValidationAware` interface defines methods for storing and retrieving error messages. A class that implements this important interface must maintain a collection of error messages for each field that can be validated, as well as a collection of general error messages that pertain only to the action as a whole. Luckily for us, all of these methods and the collections that back them are already provided by the `ActionSupport` class. To use them, we invoke the following methods:

```
addFieldError ( String fieldName, String errorMessage )
addActionError ( String errorMessage )
```

To add a field error, we must pass the name of the field, as we do in listing 3.5, along with the message that we want displayed to the user. Adding an action-scoped error message is even easier, as you don't need to specify anything other than the message.

The `ValidationAware` interface also specifies methods for testing whether any errors exist. The workflow interceptor will use these to determine whether it should redirect the workflow back to the input page. If it finds that errors exist, it'll look for a result with the name `input`. The following snippet from `chapterThree.xml` shows that the `Register` action has declared such a result:

```
<action name="Register" class="manning.chapterThree.Register">
  <result>/chapterThree/RegistrationSuccess.jsp</result>
```

```
<result name="input">/chapterThree/Registration.jsp</result>
</action>
```

In this case, the workflow interceptor, if it finds errors, will automatically forward to the result that points to the `Registration.jsp` page because its name is `"input"`. And, of course, this JSP page is our input form.

Now we've seen how interceptors clean up the action's execution logic. But some of you might not be convinced. If you're tempted to complain, "But the validation method is still on the action object!" you're right. But this doesn't taint the most important separation of concerns; the validation logic is distinctly separate from the action's own execution logic. This is what keeps our action focused on its pure unit of work—registering a new user. Check out the `execute()` method's succinct phrasing of the business logic of this task:

```
public String execute() {
    User user = new User();
    user.setPassword( getPassword() );
    user.setPortfolioName( getPortfolioName() );
    user.setUsername( getUsername() );

    getPortfolioService().createAccount( user );
    return SUCCESS;
}
```

We just make the user object and create the account. No problem. If there were some exception that might be generated from our business object's account-creation process, we might have a bit of added complexity in our choice of which result we should display. For our purposes, we're just assuming success.

But it's not just about clean-looking code. A more subtle point is that the control flow of the validation process is also separated from the action. This isn't just a case of factoring the validation logic out of the `execute()` method and into a more readable helper method. The validation workflow is itself layered away from the action's workflow because the validation logic is invoked by the workflow interceptor. In other words, the workflow interceptor is really the one controlling the execution of the validation logic. The interceptors all fire before the action itself gets a chance to execute. This separation of control flow is what allows the workflow interceptor to abort the whole request processing and redirect back to the input page without ever entering the action's `execute()` method. This is exactly the kind of separation that interceptors are meant to provide.

Before moving on, some of you are probably wondering how the error message made its way onto the registration form when we sent the user back to try again. All of this is handled for you with the Struts 2 UI component tags. We won't cover these now, as we're staying focused on the action, but you'll learn all about them in chapter 7.

Now it's beginning to feel like we've really learned something. We can write actions that automatically receive and validate data. That's cool, but let's not get distracted from the real topic at hand. The real lesson to take out of this section is about how

actions work together with interceptors to get common chores done without polluting the action's core logic. If you can wrap your mind around this action/interceptor teamwork, then you'll find the rest of the book merely an elaboration on that theme. Of course, there's a lot of cool stuff waiting for you in the remaining chapters, but this is the essence of the framework's approach to solving problems.

Before we move on, we need to look at the other problem that `ActionSupport` solves for you—localized message text.

USING RESOURCE BUNDLES FOR MESSAGE TEXT

In our `Register` action's validation logic, we set our error messages using `String` literals. If you look back at listing 3.5, you can see that we pass the `String` literal `Username is required` to the `addFieldError()` method. Using `String` literals like this creates a well-known maintenance nightmare. Furthermore, changing languages for different user locales is virtually impossible without some layer of separation between the source code and the messages themselves. The well-established best practice is to bundle these messages together into external and maintainable resource bundles, commonly implemented with simple properties files. `ActionSupport` provides built-in functionality for easily managing just that.

`ActionSupport` implements two interfaces that work together to provide this localized message text functionality. The first interface, `com.opensymphony.xwork2.TextProvider`, provides access to the messages themselves. This interface exposes a flexible set of methods by which you can retrieve a message text from a resource bundle. `ActionSupport` implements these methods to retrieve their messages from a properties file resource. No matter which method you use to retrieve your message, you'll refer to the message by a key. The `TextProvider` methods will return the message associated with that key from the properties file associated with your action class.

Getting started with a properties file resource bundle is easy. First, we need to create the properties file and give it a name that mirrors the action class for which it provides the messages. The following code snippet shows the contents from our `Register` action's associated properties file, `Register.properties`:

```
user.exists=This user already exists.
username.required=Username is required.
password.required=Password is required.
portfolioName.required=Portfolio Name is required.
```

In case you're unfamiliar with properties files, they're just simple text files. Each line contains a key and its value. In order to have the `ActionSupport` implementation of the `TextProvider` interface find this properties file, we just need to add it to the Java package that contains our `Register` class. In this case, you can find this file in the package structure at `manning.chapterThree`.

Once the properties file is in place, we can use one of the `TextProvider` `getText()` methods to retrieve our messages. Listing 3.6 shows our new version of the `Register` action's validate logic.

Listing 3.6 Using ActionSupport to get the validation error messages

```

public void validate(){

    PortfolioService ps = getPortfolioService();

    if ( getPassword().length() == 0 ){
        addFieldError( "password", getText("password.required") );    ❶
    }
    if ( getUsername().length() == 0 ){
        addFieldError( "username", getText("username.required") );
    }
    if ( getPortfolioName().length() == 0 ){
        addFieldError( "portfolioName", getText( "portfolioName.required" ) );
    }
    if ( ps.userExists(getUsername()) ){
        addFieldError("username", getText( "user.exists" ));
    }
}

```

As you can see, instead of String literals, we now retrieve our message text from ActionSupport's implementation of TextProvider. We now use the `getText()` ❶ method to retrieve our messages from properties files based upon a key. This layer of separation makes our message text much more manageable. Changing messages means only editing the properties file; the source code's semantic keys never need to be changed.

ActionSupport also provides a basic internationalization solution for the localizing message text. The `com.opensymphony.xwork2.LocaleProvider` interface exposes a single method, `getLocale()`. ActionSupport implements this interface to retrieve the user's locale based upon the locale setting sent in by the browser. You could implement your own version of this interface to search somewhere else for the locale, such as in the database. But if the browser setting is good enough for your requirements, you don't have to do too much to achieve a basic level of internationalization.

You still retrieve your message texts as we did earlier. Even when we weren't taking advantage of it, ActionSupport's TextProvider implementation has been checking the locale every time it retrieves a message text for us. It does this by calling the `getLocale()` method of the `LocaleProvider` interface. With the locale in hand, the TextProvider, a.k.a. ActionSupport, tries to locate a properties file for that locale. Of course, you have to provide the properties file for the locale in question, or it will just serve up the standard English. But it's simple to provide properties files for all locales that you wish to support. In Struts 2 Portfolio, we're providing a Spanish properties file. The hard part is finding a translator. In order to see this in action, set your browser's language support to Spanish and submit the registration form again, omitting one of the fields to see the error message that it provides.

As with validation, the internationalization provided by ActionSupport is relatively primitive. If it suffices for your application, great. If you need more, we'll see how to get cutting-edge internationalization from the Struts 2 framework in chapter 11. For now, we've got a pretty decent start on building actions.

Next, we'll look at some alternative—advanced, some would say—methods of implementing our data transfer with complex objects instead of simple JavaBeans properties.

3.4 *Transferring data onto objects*

Up until now, our actions have all received data from the request on simple JavaBeans properties. While they are powerful and elegant, we can do even better. Rather than receiving each piece of data individually, and then creating an object on which to place these pieces of data, we can expose the complex object itself to the data transfer mechanisms of the platform. Not only does this save time by eliminating the need to create and populate the object that aggregates the individual pieces of data, it can also save work by allowing us to directly expose an already-existing domain object to the data transfer. While a couple of caveats must be kept in mind, the use of these complex objects as direct data transfer objects presents a powerful option to the developer.

NOTE *Struts 1 to Struts 2 Perspective*—In case you're feeling homesick, we should note the departure of the familiar Struts 1 `ActionForm`. `ActionForms` played an important role in data validation and type conversion for the Struts 1 framework, but the cost was high. For each domain object, you typically had to create a mirroring form bean. To add insult to injury, you were then tasked with an additional manual data transfer when you finally moved the valid data from the form bean onto your domain object. For many, one of the biggest thrills of Struts 2 will be letting the framework transfer, validate, and bind data directly onto application domain objects, where it can stay!

If we want to use complex objects rather than simple JavaBeans properties to receive our data, we have a couple of options for implementing such deep transfers. Our first option is also JavaBeans-based. We can expose a complex object itself as a JavaBeans property and have the data moved onto the object directly. Another alternative is to use something called a `ModelDriven` action. This option involves a simple interface and another one of the default interceptors. Like the object-backed JavaBeans property, the `ModelDriven` action also allows us to use a complex Java object to receive our data. The differences between these two methods are slight, and there are no functional consequences to choosing one over the other. But you might prefer one over the other depending on your project requirements. We'll learn each technique and demonstrate with examples from the Struts 2 Portfolio.

3.4.1 *Object-backed JavaBeans properties*

We've already seen how the `params` interceptor, included in the `defaultStack`, automatically transfers data from the request to our action objects. To enable this transfer, the developer needs only to provide JavaBeans properties on her actions, using the same names as the form fields being submitted. This is easy, but despite this ease, we frequently find ourselves occupied with another tedious task. This tedious task con-

sists of collecting these individually transferred data items and transferring them to an application domain object that we must instantiate ourselves. Listing 3.7 shows our previous version of the Register action's `execute()` method.

Listing 3.7 Collecting data and building the domain object by hand

```
public String execute() {
    User user = new User();
    user.setPassword( getPassword() );
    user.setPortfolioName( getPortfolioName() );
    user.setUsername( getUsername() );

    getPortfolioService().createAccount( user );
    return SUCCESS;
}
```

①

While we were impressed with the succinct quality of this method only a few pages ago, we can now see that five of the seven lines do nothing more than assemble the individual pieces of data ① that the framework has transferred onto our simple JavaBeans properties. We're still psyched that the data has been automatically transferred and bound to our Java data types, but why not ask for more?

Why not ask the framework to go ahead and transfer the data directly to our `User` object? Why not ask the framework to instantiate the user object for us? Since Struts 2 provides powerful data transfer and type conversion facilities, the true power of which we'll discover later in the book, we can ask for these things and get them. In this case, it's simple. Let's rewrite our Register action so that it replaces the individual JavaBeans properties with a single property backed by the `User` object itself. Listing 3.8 shows the new version of our new action as implemented in the `manning.chapterThree.ObjectBacked.ObjectBackedRegister` class.

Listing 3.8 Using an object-backed property to receive data transfers

```
public String execute() {
    getPortfolioService().createAccount( user );
    return SUCCESS;
}

private User user;

public User getUser() {
    return user;
}

public void setUser(User user) {
    this.user = user;
}

public void validate() {
    . . .
}
```

①

②

```

        if ( getUser().getPassword().length() == 0 ) {           ❸
            addFieldError( "user.password", getText("password.required") );
        }
        . . .
    }

```

Listing 3.8 has now reduced our business logic to a one-liner. We hand an already instantiated and populated `User` object to our service object's account creation method ❶. That's it. This logic is much cleaner because we let the framework handle instantiating our `User` object and populating its attributes with data from the request. Previously, we'd done this ourselves. In order to let the framework handle this tedious work, we simply replaced the individual JavaBeans properties with a single property backed by the `User` object itself ❷. We don't even have to create the `User` object that backs the property because the framework's data transfer will handle this for us when it starts trying to move the data over. Note that our validation code now must use a deeper notation ❸ to reach the data items, because they must go through the user property to get to the individual fields themselves.

Similarly, we also have to make a couple of changes in the way we reference our data from our results, JSPs in this case. First of all, we have to change the field names in the form that submits to our new action. The bottom line is that we now have another layer in our JavaBeans properties notation. The following code snippet shows the minor change to the textfield name in our form, found in our `Registration_OB.jsp` page.

```
<s:textfield name="user.username" label="Username"/>
```

As you can see, the reference now includes the user to reflect the depth of the property in the action. Previously, when we exposed each piece of user data as an individual JavaBeans property, our reference didn't require the user portion of this reference. The names of the other fields in this form are similarly transformed.

We make a similar alteration at the other end of the request. When we render our resulting success page, we must use the deeper property notation to access our data. The following code snippet from `RegistrationSuccess_OB.jsp` shows the new notation:

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="user.portfolioName" /> Portfolio</h3>
```

As you can see, directly using an application domain object as a JavaBeans property allows us to let the framework do even more of our work for us. The minor consequences are that we have to go a dot deeper when we reference our data from the JSP pages. Now, we'll take a look at another method of exposing rich objects to the framework's data transfer facilities, one that gives us the same cleaner `execute()` method as the object-backed JavaBeans property, but doesn't introduce the extra dot in our view tier data access.

3.4.2 *ModelDriven actions*

`ModelDriven` actions depart from the use of JavaBeans properties for exposing domain data. Instead, they expose an application domain object via the `getModel()` method,

which is declared by the `com.opensymphony.xwork2.ModelDriven` interface. While this method introduces a new interface, as well as another interceptor, it's simple in practice. The interceptor is already in the default stack; the data transfer is still automatic and even easier to work with than previous techniques. Let's see how it works.

Implementing the interface requires almost nothing. We have to declare that our action implements the interface, but there's only one method exposed by `ModelDriven`, the `getModel()` method. By *model*, we mean the model in the MVC sense. In this case, it's the data that comes in from the request and is altered by the execution of the business logic. That data is then made available to the view, JSP pages in the case of our Struts 2 Portfolio application. Listing 3.9 shows the new action code from the `manning.chapterThree.modelDriven.ModelDrivenRegister` class.

Listing 3.9 Automatically transferring request data to application domain objects

```
public class ModelDrivenRegister extends ActionSupport
    implements ModelDriven { ❶

    public String execute() {
        getPortfolioService().createAccount( user );
        return SUCCESS;
    }

    private User user = new User(); ❷
    public Object getModel() {
        return user;
    }

    public void validate() {
        . . .
        if ( user.getPassword().length() == 0 ) {
            addFieldError( "password", getText("password.required") ); ❸
        }
        . . .
    }
    . . .
}
```

First, we see that our new action implements the `ModelDriven` interface ❶. The only method required by this interface is `getModel()`, which returns our model object, the familiar `User` object. Note that with the `ModelDriven` method, we have to initialize the `User` object ourselves ❷. We'll see why in chapter 5 when we explore the details of the data transfer mechanisms, but for now just keep an eye on this slight but important detail.

We should note one pitfall to avoid. By the time the `execute()` method of your `ModelDriven` action has been invoked, the framework has obtained a reference to your model object, which it'll use throughout the request. Since the framework acquires its reference from your getter, it won't be aware if you change the model field

internally in your action. This can cause some data inconsistency problems. If, during your execution code, you change the object to which your model field reference points, your action's model will then be out of sync with the one still held by the framework. The following code snippet demonstrates the problem:

```
public String execute() {
    user = new User();
    user.setSomething();
    getPortfolioService().createAccount( user );
    return SUCCESS;
}

private User user = new User();
public Object getModel() {
    return user;
}
```

In this action's `execute()` method, the developer has, for some reason, set the user reference to a new object. But the framework still has a reference to the original object as initialized in the instance field declaration for user. When the framework invokes the result, your JSP page data access will be resolved against the old object. Whatever this erroneous code has set, it'll be unavailable. You can, of course, manipulate that original model object to your heart's content. Just don't make a new one, or point the existing reference to another one!

As in the previous object-backed JavaBeans property method, using a domain object to receive all of the data allows us the luxury of a clean `execute()` method. Again we incur a slight penalty related to the depth of our references. As you can see in the validation code of listing 3.9, we now refer to the password by referencing the model object en route to the password field ❸.

However, we don't incur any depth of reference penalty in our view layer. All references in the JSP pages return to the simplicity of the original Register action that used the simple, individual JavaBeans properties for data transfer. The following code snippets, from `Registration_MD.jsp`

```
<s:textfield name="username" label="Username"/>
```

and `RegistrationSuccess.jsp`

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

show the renewed simplicity of view-layer references to data carried in the Model-Driven action. This is considered one of the primary reasons for choosing the ModelDriven method over the object-backed JavaBeans property method of exposing domain objects to the data transfer.

Using domain objects for data transfer is great, but a word of caution is necessary. We'll explore a potential danger next.

3.4.3 Last words on using domain objects for data transfer

First, we want to point out a potential danger in using domain objects for data transfer. The problem comes when the data gets automatically transferred onto the object. As we've seen, if the request has parameters that match the attributes on your domain object, the data will be moved onto those attributes. Now, consider the case where your domain object has some sensitive data attributes that you don't really want to expose to this automatic data transfer, perhaps an ID. A malicious user could add an appropriately named `quering` parameter to the request such that the value of that parameter would automatically be written to your exposed object's attribute. Of course, you can remove these attributes from the object, but then you start to lose the value of reusing existing objects rather than writing new ones. Unfortunately, there's no good solution to this issue yet. Usually, you won't have anything to worry about, but it's something to keep in mind when you're developing your actions.

Ultimately, it'll be up to you to choose a method of receiving the data from the framework. Each method has its purpose, and we believe that the requirements of your projects will typically determine which approach is most appropriate. Throughout the rest of this book, we'll see many examples of best practices and integration with other technologies that'll spell out some of the cases when one or another method serves best. Sometimes, it's appropriate to use a little of each. Did we forget to mention that you can do all of them at the same time if you like? Again, the platform is flexible. Now it's time to look at a case study.

3.5 File uploading: a case study

At this point, you have the tools you need to write your application's action components and wire them into a rudimentary Struts 2 application. We suspect you've even deduced enough to get started implementing a view layer with JSP results. Later in the book, you'll see how much more the framework has to offer your view layer when we get to results, tags, UI components, and Ajax integration in part 3. For now, we want to round out our treatment of the action component by showing a useful case study that, while showing you how to do something practical, also serves to reiterate how actions and interceptors work together to solve the common problems of the web application domain.

Most of you will have to implement file upload at some point. Our sample application, the Struts 2 Portfolio, will need to upload some image files; otherwise the portfolio would be drab. One reason we're showing you how to upload files now, rather than later in the book, is that we believe it helps demonstrate the framework's persistent pattern of using interceptors to layer the logic of common tasks out of the action itself. So let's learn how our actions can work with an interceptor from the default stack to implement ultraclean and totally reusable file uploading.

3.5.1 Getting built-in support via the `struts-default` package

As with most tasks that you find yourself doing routinely, Struts 2 provides built-in help for file uploading. In this case, the default interceptor stack includes the `FileUploadInterceptor`. As you might recall, `struts-default.xml` is the system file that defines all of

the built-in components. Listing 3.10 shows the elements from that file that declare the `fileUpload` interceptor and make it a part of the default interceptor stack.

Listing 3.10 Declaring the `FileUploadInterceptor` and adding it to the stack

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

  <interceptors>

    . . .

    <interceptor name="fileUpload"
      class="org.apache.struts2.interceptor.FileUploadInterceptor"/>

    . . .

  </interceptors>

  . . .

  <interceptor-stack name="defaultStack">

    . . .

    <interceptor-ref name="model-driven"/>
    <interceptor-ref name="fileUpload"/>
    <interceptor-ref name="params"/>

    . . .

  </interceptor-stack>

</package>
```

As you can see, the `struts-default` package contains a declaration of the `fileUpload` interceptor, backed by the `org.apache.struts2.interceptor.FileUploadInterceptor` implementation class. This interceptor is then added to the `defaultStack` so that all packages extending the `struts-default` package will automatically have this interceptor acting on their actions. We make our Struts 2 Portfolio packages extend this package to take advantage of these built-in components.

3.5.2 What does the `fileUpload` interceptor do?

The `fileUpload` interceptor creates a special version of the automatic data transfer mechanisms we saw earlier. With the previous data transfers, we were dealing with the transfer of form field data from the request to matching JavaBeans properties on our action objects. The `params` interceptor, also part of the `defaultStack`, was responsible for moving all of the request parameters onto the action object wherever the action provided a JavaBeans property that matched the request parameter's name. In listing 3.10, you can see that the `defaultstack` places the `fileUpload` interceptor just before the `params` interceptor. When the `fileUpload` interceptor executes, it processes a multi-part request and transforms the file itself, along with some metadata, into request parameters. It does this using a wrapper around the servlet request. Table 3.2 shows the request parameters that are added by this `fileUpload` interceptor.

Table 3.2 Request parameters exposed by the `FileUpload` interceptor

Parameter name	Parameter type and value
[file name from form]	File—the uploaded file itself
[file name from form]ContentType	String—the content type of the file
[file name from form]FileName	String—the name of the uploaded file, as stored on the server

After the `fileUpload` interceptor has exposed the parts of the multipart request as request parameters, it's time for the next interceptor in the stack to do its work. Conveniently, the next interceptor is the `params` interceptor. When the `params` interceptor fires, it moves all of the request parameters, including those listed in table 3.2, onto the action object. Thus, all a developer needs to do to conveniently receive the file upload is add JavaBeans properties to her action object that match the names in table 3.2.

FYI We should note the elegant use of interceptors as demonstrated by the `fileUpload` interceptor. As we've said, the Struts 2 framework tries desperately to keep its action components as clean as possible. A large part of this effort consists of the use of interceptors to layer cross-cutting tasks away from the core processing tasks of the action itself. The `fileUpload` interceptor demonstrates this by encapsulating the processing of multipart requests and injecting the processed upload data into the action object's JavaBeans setter methods.

We've also referred to the role of interceptors in terms of preprocessing and postprocessing. In the case of the `fileUpload` interceptor, the preprocessing is the transformation of the multipart request into request parameters that the `params` interceptor will automatically move to our action. The postprocessing comes when the interceptor fires again after our action to dispose of the temporary version of the uploaded file.

How will all of this look in code? The Struts 2 Portfolio uses file uploading, so let's have a look.

3.5.3 Looking at the Struts 2 Portfolio example code

The Struts 2 Portfolio uses this file upload mechanism to upload new images to the portfolio. The first part of such a task is presenting a form through which users can upload files. You can visit the image upload page by first creating an account in the chapter 3 version of the Struts 2 Portfolio sample application. Our end-user workflow is incomplete right now, but we'll fix that in coming chapters. Once you've created an account, choose to work with your portfolio and choose to add a new picture. You'll see a page presenting you with a simple form to upload an image. The following code snippet, from `chapterThree/ImageUploadForm.jsp`, shows the markup that creates the form you see:

```

<h4>Complete and submit the form to create your own portfolio.</h4>
<s:form action="ImageUpload" method="post" enctype="multipart/form-data">
    <s:file name="pic" label="Picture"/>
    <s:submit/>
</s:form>

```

When we create this form, we have to take note of a couple of points. First, note that we're using Struts 2 tags to build the form. We'll cover the Struts 2 tag library in chapters 6 and 7. For now, just accept that this tag generates the HTML markup of a form that allows the user to upload a file. Next, note that we set the encoding type of the form to `multipart/form-data`. This important attribute signals to the framework that the request needs to be handled as an upload. Without this setting, it won't work. Finally, note that the file will be submitted by the form under the name attribute we provide to the file tag. This detail is important because you'll use this name to build the JavaBeans properties that will receive the upload data.

With our JSP ready to present the form, let's see the action that will receive and process the upload. First, make sure that the package to which your action belongs is extending the `struts-default` package so that it inherits the default interceptor stack, and the `fileUpload` interceptor. Listing 3.11, a snippet from our `manning/chapterThree/chapterThree.xml` file, shows that we've done this.

Listing 3.11 Extending the `struts-default` package to inherit file upload processing

```

<package name="chapterThreeSecure" namespace="/chapterThree/secure"
    extends="struts-default">
    . . .
    <action name="AddImage" >
        <result>/chapterThree/ImageUploadForm.jsp</result>
    </action>
    <action name="ImageUpload" class="manning.chapterThree.ImageUpload">
        <result>/chapterThree/ImageAdded.jsp</result>
        <result name="input">/chapterThree/ImageUploadForm.jsp</result>
    </action>
    . . .
</package>

```

This is our package of secure actions for the Struts 2 Portfolio. We haven't added security yet, but we know that these actions will require security of some kind, so we've put them into a separate package. We'll add the security with a custom interceptor in chapter 4.

With the `defaultStack` and its file upload interceptor on our side, we just need to add properties to our action object that match the parameter names, as seen in table 3.2.

We've already seen that our file will be submitted under the name `pic`. Using the naming conventions in table 3.2, we can derive the JavaBeans property names that we need to implement. Listing 3.12 shows the JavaBeans properties implemented by the

manning.chapterThree.ImageUpload class. (As always, check out the Struts 2 Portfolio source code if you want to see more of the sample code.)

Listing 3.12 The JavaBeans properties that'll receive the uploaded file and metadata

```
File pic;
String picContentType;
String picFileName;

public File getPic() {
    return pic;
}

public void setPic(File pic) {
    this.pic = pic;
}

public String getPicContentType() {
    return picContentType;
}

void setPicContentType(String picContentType) {
    this.picContentType = picContentType;
}

public void setPicFileName(String picFileName) {
    this.picFileName = picFileName;
}

public String getPicFileName() {
    return picFileName;
}
```

You're not obligated to implement all of these. If you choose not to implement some of them, you just won't receive the data. No harm, no foul. At any rate, using the `fileUpload` interceptor is about as easy as writing these JavaBeans properties. Thanks to the separation of the upload logic, the action's work itself is simple. As shown in the following code snippet from the `ImageUpload` action, the action can focus on the task at hand.

```
public String execute(){

    getPortfolioService().addImage( getPic() );
    return SUCCESS;

}
```

There's nothing here but the call to our business logic. The image file is conveniently just a getter away, just as all of the auto-transferred data has been. Thanks to the teamwork of the `fileUpload` interceptor and the `params` interceptor, uploading files is almost as easy as handling primitives. Incidentally, you can set the path to the directory where the action will save the image file with a parameter to the image upload action element in `chapterThree.xml`.

Now let's look at a couple of tweaks you can make to the `fileUpload` interceptor to handle such things as multiple file uploads.

MULTIPLE FILES AND OTHER SETTINGS

Uploading multiple files with the same parameter names is also supported. All you have to do is change your action's JavaBeans properties to arrays; that is, `File` becomes `File[]`, and the two strings become string arrays. The three arrays are always the same length, and their order is the same, meaning that index 0 for all three arrays represents the same file and file metadata. There are also many other configurable parameters regarding the `fileUpload` interceptor, ranging from the maximum file size to the implementation of the multipart request parser that'll be used to handle the request. In general, the extreme flexibility of the Struts 2 framework makes it impossible to provide complete coverage of all the details in a book such as this. This book strives to filter out as much of the extraneous detail as possible in order to make the concepts of the framework more visible. For such details and minutia, the Struts 2 website serves as a good reference.

3.6 **Summary**

In this chapter, we learned a lot about building Struts 2 actions. We began our tour of this important component by examining the role of actions within the framework. Actions have to do three things. First and foremost, they encapsulate the framework's interaction with the model. This means, ultimately, that the calls to the business logic and data tier will be found in the `execute()` method of the action class. The second job of the action is to serve as the data transfer object for the request processing. We suspect we've made this point particularly clear by now. Finally, the action also takes responsibility for returning a control string that'll be used by the framework to select the appropriate result component for rendering the view back to the user.

We also saw how to package our action components into Struts 2 packages. These packages help provide a logical organization to your application's framework components, actions in particular. Using the package structure, we can do several important things. We can map URL namespaces to groups of actions. We can also take advantage of the inheritance mechanisms of packages to define reusable groups of framework components. We've already used this feature by having our Struts 2 Portfolio packages extend the built-in `struts-default` package to take advantage of its default interceptor stack, among other things. Let this serve as a model for creating your own package hierarchies.

We then showcased a couple of key players provided by the framework to ease your work. First we saw the `Action` interface, which provides some important definitions of constants that the framework uses for commonly used control strings. After that, we took a long look at the functionality provided by the `ActionSupport` class. This helpful class implements several important interfaces and cooperates with key interceptors from the `defaultStack` to provide built-in implementations of such valuable domain tasks as validation and a basic form of internationalization. We demonstrated all of this with our Struts 2 Portfolio sample application.

One of the more important considerations when implementing your own Struts 2 actions will be the method of data transfer that you use. Several options are available. We covered two methods that both implement JavaBeans properties on the action object itself. The first of these matches simple properties to individual parameters on the incoming request. The next JavaBeans properties method provides properties that are backed by complex domain objects. Finally, we saw that you can use an entirely different method to expose your complex domain objects by implementing `ModelDriven` actions. The choice of which method to use will largely depend upon the requirements of your project and the action at hand. Flexibility is a recurring theme of the Struts 2 framework.

We rounded off the chapter with a case study. We looked at using one of the framework's built-in interceptors to add a file upload action to our sample application. While we saw that uploading files with Struts 2 can be easy, we also tried to point out some important lessons that this example demonstrates about the framework itself. In particular, we've shown what we mean when we say the framework tries to provide a clean implementation of MVC. In particular, the file upload example shows how proper cooperation between interceptors and actions can provide a web application with reusable and flexible encapsulations of cross-cutting tasks, as well as super-clean actions.

You know what an action is by now. Next up is a detailed look at interceptors, and we'll enhance the Struts 2 Portfolio by putting interceptors to good use.