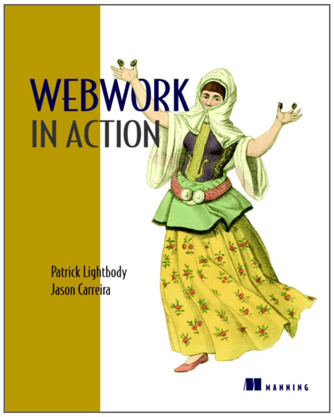


2 *HelloWorld, the WebWork way*

This chapter covers

- How to set up a WebWork project
- How to create your first WebWork action
- Input and output of data from your action
- An introduction to the WebWork JSP tag library

In this chapter, we'll walk through a brief example that demonstrates the basics of WebWork. By the end of this chapter, you should have enough understanding of WebWork to build simple web-based applications. Later in the book, you'll expand on what you learned here to do more advanced configurations and take advantage of advanced features including validation, internationalization, scripting, type conversion, and support for display formats other than HTML, such as PDF.

2.1 Downloading WebWork

Before you begin, you must download WebWork. You can find the latest version (2.1.7 at the time of this writing) at http://webwork.dev.java.net/servlets/ProjectDocumentList. Once you've downloaded the distribution binary, such as webwork-2.1.7.zip, you need to unzip it. Inside, you'll find sample applications, documentation, and the source code of the framework so you can see how it works. You'll also find all the JAR files required to get WebWork running.

We highly recommend that you examine all the documentation and sample code, but for now we're only concerned with the required libraries and the WebWork JAR. Let's begin by preparing the basic web application file structure so that you can start building applications.

2.2 Preparing the skeleton

The basic web application file structure, also known as the *skeleton*, is the bare minimum required to begin building the sample applications you'll explore in this chapter. You need the files listed in table 2.1, which are found in the downloadable WebWork distribution.

Table 2.1 Files required to set up a WebWork web application

Filename	Description
xwork.jar	XWork library on which WebWork is built
commons-logging.jar	Commons logging, which WebWork uses to support transparently logging to either Log4J or JDK 1.4+
oscore.jar	OSCore, a general-utility library from OpenSymphony
velocity-dep.jar	Velocity library with dependencies
ognl.jar	Object Graph Navigation Language (OGNL), the expression language used throughout WebWork

Table 2.1 Files required to set up a WebWork web application *(continued)*

Filename	Description
xwork.xml	WebWork configuration file that defines the actions, results, and interceptors for your application
web.xml	J2EE web application configuration file that defines the servlets, JSP tag libraries, and so on for your web application

Some files, especially configuration files, aren't included in the distribution; you'll create them in a moment. Also note that the version numbers of the dependent JAR files (such as oscore.jar and ognl.jar) can be found in the distribution in the file versions.txt, located in the lib directory.

As usual for J2EE web applications, JARs go in the WEB-INF/lib directory and web.xml goes in the WEB-INF directory. As is most often the case, configuration elements such as xwork.xml go in the WEB-INF/classes directory. Having done that, your directory layout should appear as follows:

```
 / (Root)
|---WEB-INF
    |---web.xml
    |---classes
    |     |---xwork.xml
    |---lib
    |     |---webwork-2.1.7.jar, xwork.jar, oscore.jar, ognl.jar, ...
```

NOTE Now that the directory structure is set, you must configure your web application server (such as Resin, Orion, or Apache Tomcat) to deploy the web app starting from the location marked Root. How you deploy it depends on the server you're using and whether you zip the directory layout as a WAR beforehand. Consult your server's documentation for detailed instructions on deploying web apps such as this one. We also suggest that you consult your IDE's documentation to be able to deploy your web application directly from the IDE, either by using a plug-in or by starting the container's main class as an executable.

2.2.1 Creating the web.xml deployment file

In order for WebWork to work properly, it needs to be deployed in such a way that certain URL patterns, such as `*.action`, map to a WebWork servlet that is responsible for handling all WebWork requests. A URL *pattern* is a pattern that is matched against all incoming HTTP requests (such as from web browsers). If the location, also known as a *resource*, matches the pattern, the associated servlet is invoked. Without this servlet, WebWork wouldn't function.

You must first add the following entry to web.xml:

```
<servlet>
    <servlet-name>webwork</servlet-name>
    <servlet-class>
        com.opensymphony.webwork.dispatcher.ServletDispatcher
    </servlet-class>
</servlet>
```

The next step is to map the servlet to a URL pattern. The pattern you choose can be anything you want, but the most typical pattern is `*.action`. You can configure the pattern the servlet will match by adding the following to web.xml:

```
<servlet-mapping>
     <servlet-name>webwork</servlet-name>
     <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

Finally, in order to use WebWork's tag library, you must add an entry to web.xml indicating where the tag library definition can be found:[1]

```
<taglib>
     <taglib-uri>webwork</taglib-uri>
     <taglib-location>/WEB-INF/lib/webwork-2.1.7.jar
     </taglib-location>
</taglib>
```

You can add many other optional configuration items to web.xml, such as support for JasperReports, Velocity, FreeMarker, and other view technologies. Because you're building a skeleton application with no other files or configuration elements, the final web.xml file should look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE web-app PUBLIC
     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/dtd/web-app_2_3.dtd">
 <web-app>
     <servlet>
         <servlet-name>webwork</servlet-name>
         <servlet-class>
             com.opensymphony.webwork.dispatcher.ServletDispatcher
         </servlet-class>
     </servlet>
     <servlet-mapping>
         <servlet-name>webwork</servlet-name>
         <url-pattern>*.action</url-pattern>
```

[1] In newer servlet containers that fully support the JSP 1.2 specification, the taglib should be automatically picked up without any configuration.

```
        </servlet-mapping>
        <taglib>
            <taglib-uri>webwork</taglib-uri>
            <taglib-location>/WEB-INF/lib/webwork-2.1.7.jar
            </taglib-location>
        </taglib>
   </web-app>
```

This is the most basic web.xml file you can use for a WebWork application. Most likely, your web.xml will quickly grow to include additional servlets, tag libraries, event listeners, and so on.

2.2.2 Creating the xwork.xml configuration file

Now that you've configured web.xml correctly, you need to set up a skeleton configuration for WebWork itself. Because WebWork is based on a subproject, XWork, the configuration file is named xwork.xml and is located in WEB-INF/classes, as previously shown. We'll discuss configuration in more detail in chapter 3, "Setting up WebWork," so don't worry too much about the contents of this file. For now, the skeleton setup requires just the following in xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
    <include file="webwork-default.xml"/>

    <package name="default" extends="webwork-default">
        <default-interceptor-ref name="completeStack"/>
    </package>
</xwork>
```

The key thing to note here is that a file, webwork-default.xml, is included. Doing this ensures that all the WebWork additions built on top of XWork are available to you. This file contains the standard configuration for WebWork, so it's very important that it be included. Without this file, WebWork wouldn't function as you'd expect, because it wouldn't be correctly configured. Note that you don't need to have a file named webwork-default.xml—it's already included in the WebWork JAR file.

2.2.3 Creating the webwork.properties configuration file

Just as you placed xwork.xml in WEB-INF/classes, you also need to add a file called webwork.properties to that directory. Like other aspects of WebWork configuration, the contents of this file are discussed in chapter 3. For now, add the following line to webwork.properties:

```
webwork.tag.altSyntax = true
```

This line is required because every example in this chapter (and the rest of the book) is given with the assumption that `webwork.tag.altSyntax` is set to `true`. We did this to let you have the most up-to-date information about a framework that is always evolving. At the time of this writing, WebWork 2.1.7 is the latest released version. However, we already know that as of WebWork 2.2, `altSyntax` will become standard; so, we felt it would be best to cover this syntax now rather than teach something that is on the verge of changing.

2.2.4 Tips for developing WebWork apps

You're now ready to begin building your first example application. In order to do so, you must compile Java sources and copy the resulting .class files to WEB-INF/classes. There are several ways to do this, including executing `javac` by hand, using an Ant build script, or using an IDE such as Eclipse or JetBrains IntelliJ IDEA (formerly IntelliJ IDEA). You should choose whatever method you're most comfortable with. In the CaveatEmptor example used in the rest of the book, you'll find project files for IDEA as well as an Ant build script to help you get started. Feel free to use and modify the supplied build scripts and project files for your own projects.

Using an IDE may be a better approach because you can launch the application server, debug, and edit all within the same environment. Without these features, you have to manually stop and start your application server from the command line whenever you make changes to the code in your applications. If you haven't tried using a full-featured IDE with J2EE application server support, we highly recommend doing so.

2.3 Your first action

Let's start by creating a simple WebWork action. An *action* is a piece of code that is executed when a particular URL is requested. After actions are executed, a *result* visually displays the outcome of whatever code was executed in the action. A result is generally an HTML page, but it can also be a PDF file, an Excel spreadsheet, or even a Java applet window. In this book, we'll primarily focus on HTML results, because those are most specific to the Web. As Newton's Third Law states, every action must have a reaction.[2] Although not "equal and opposite," a result is always the reaction to an action being executed in WebWork.

Suppose you want to create a simple "Hello, World" example in which a message is displayed whenever a user goes to a URL such as http://localhost/

[2] An action doesn't technically have to have a result, but it generally does.

helloWorld.action. Because you've mapped WebWork's servlet to `*.action`, you need an action named `helloWorld`. To create the "Hello, World" example, you need to do three things:

1. Create an action class: `HelloWorld`.
2. Create a result: hello.jsp.
3. Configure the action and result.

Let's begin by writing the code that creates the welcome message.

2.3.1 Saying hello, the WebWork way

Start by creating the action class, HelloWorld.java, as shown in listing 2.1.

Listing 2.1 HelloWorld.java

```
package ch2.example1;

import com.opensymphony.xwork.Action;

public class HelloWorld implements Action {
    private String message;

    public String execute() {
        message = "Hello, World!\n";
        message += "The time is:\n";
        message += System.currentTimeMillis();
        return SUCCESS;
    }

    public String getMessage() {
        return message;
    }
}
```

The first and most important thing to note is that the `HelloWorld` class implements the `Action` interface. All WebWork actions must implement the `Action` interface, which provides the `execute()` method that WebWork calls when executing the action.

Inside the `execute()` method, you construct a "Hello, World" message along with the current time. You expose the message field via a `getMessage()` JavaBean-style getter. This allows the message to be retrieved and displayed to the user by the JSP tags.

Finally, the `execute()` method returns `SUCCESS` (a constant for the `String` "success"), indicating that the action successfully completed. This constant and others, such as `INPUT` and `ERROR`, are defined in the `Action` interface. All WebWork actions must return a *result code*—a `String` indicating the outcome of the action execution. Note that the result *code* doesn't necessarily mean a result will be executed, although generally one is. You'll soon see how these result codes are used to map to results to be displayed to the user. Now that the action is created, the next logical step is to create an HTML display for this message.

2.3.2 Displaying output to the web browser

WebWork allows many different ways of displaying the output of an action to the user, but the simplest and most common approach is to show HTML to a web browser. Other techniques include displaying a PDF report or a comma-separated value (CSV) table. You can easily create a JSP page that generates the HTML view:

```
<%@ taglib prefix="ww" uri="webwork" %>
 <html>
     <head>
         <title>Hello Page</title>
     </head>
     <body>
     The message generated by my first action is:
     <ww:property value="message"/>
     </body>
</html>
```

The taglib definition in the first line maps the prefix *ww* to the URI *webwork*. (Note that the URI is the same as that in the web.xml file.) A prefix of *ww* indicates that all the WebWork tags will start with `ww:`.

As you can see, this is a simple JSP page that uses one custom WebWork tag: `property`. The `property` tag takes a `value` attribute and attempts to extract the content of that expression from the action. Because you created a `getMessage()` method in the action, a `property` value of `message` results in the return value of a `getMessage()` method call. Save this file in the root of your web application, and call it hello.jsp.

Again, this example is extremely basic. In later chapters, we'll go over many other WebWork tags that can help you create dynamic web sites without using any Java code in your JSPs, using a simple expression language called the Object Graph Navigation Language (OGNL).

2.3.3 *Configuring your new action*

Now that you've created both the action class and the view, the final step is to tie the two together. You do so by *configuring* the action to a particular URL and mapping the `SUCCESS` result to the JSP you just created. Recall that when you created the skeleton layout, you generated a nearly empty xwork.xml file. You'll now add some meaningful values to this file and see the final WebWork action work.

When you're configuring a WebWork action, you must know three things:

- The full action class name, including the complete package
- The URL where you expect the action to exist on the Web
- All the possible result codes the action may return

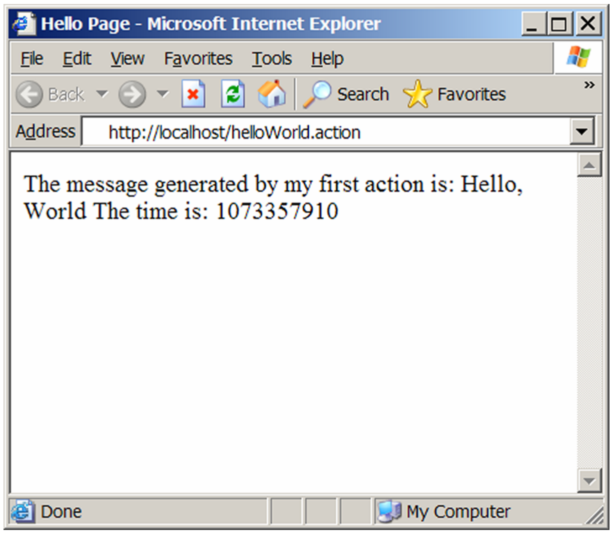
As you know from the previous Java code, the action class name is `ch2.example1.HelloWorld`. The URL can be anything you like; in this case, we choose /helloWorld.action. You also know that the only possible result code for this action is `SUCCESS`.

Armed with this information, let's modify xwork.xml to define the action:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
 "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
 <xwork>
     <include file="webwork-default.xml"/>

     <package name="default" extends="webwork-default">
         <default-interceptor-ref name="completeStack"/>

         <action name="helloWorld"
                 class="ch2.example1.HelloWorld">
             <result name="success">hello.jsp</result>
         </action>
     </package>
</xwork>
```

In this file, you've now made a direct correlation between an action name (`helloWorld`) and the class you wish to be executed. So, any HTTP request to /helloWorld.action will invoke your new action class. You also made a direct correlation between the result code `SUCCESS` (a `String` constant for `"success"`) and the JSP that you just created to display the message.

With xwork.xml saved, the action class compiled and copied to WEB-INF/classes, and hello.jsp added to the root of the web application, you're ready to fire up the application server and try this new action. Consult your application server's documentation for detailed instructions on how to start, stop, and deploy web applications like this one.


Hello Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/helloWorld.action
Done
My Computer

The message generated by my first action is: Hello, World The time is: 1073357910

Figure 2.1
The first "Hello World" action

And that's it! You can now point your web browser to the action URL, such as http://localhost/helloWorld.action,[3] to see the final product shown in figure 2.1.

NOTE The message returned by this action isn't very friendly to the eye: The time (displayed as 1073357910) is pretty hard to read. Don't despair—in chapter 14, "Internationalization," we'll show you how easy it is to display locale-specific dates to the user.

As you can see, this isn't the most exciting web page, so let's spice it up. You'll make the greeting generated by this action customizable by letting users enter their name and be personally greeted. Up to this point, you've seen an action that is read-only; now you'll learn how to handle inputs and read-write actions.

2.4 Dealing with inputs

Now that you know how to build a simple action, let's take it up one notch and add the ability to personalize the message. You'll build on the existing code. First, create another HTML page that asks for the user's name. Create the following file, name.jsp, in the same directory as hello.jsp:

[3] Depending on the servlet container and the configuration, this URL could include a port number like 8080.

```
<html>
     <head>
         <title>Enter your name</title>
     </head>
     <body>
          Please enter your name:
          <form action="helloWorld.action">
              <input type="textfield"
                     name="name"/>
              <input type="submit"/>
          </form>
     </body>
</html>
```

Note that the form is being submitted to helloWorld.action—the same location you used to display the previous example. Since you're expanding on the previous example, you'll continue to use this location. Another important point is that the textfield input is named `name`. Just as `message` was the property you used to display (get) the message, `name` is the property you use to write (set) the user's name.

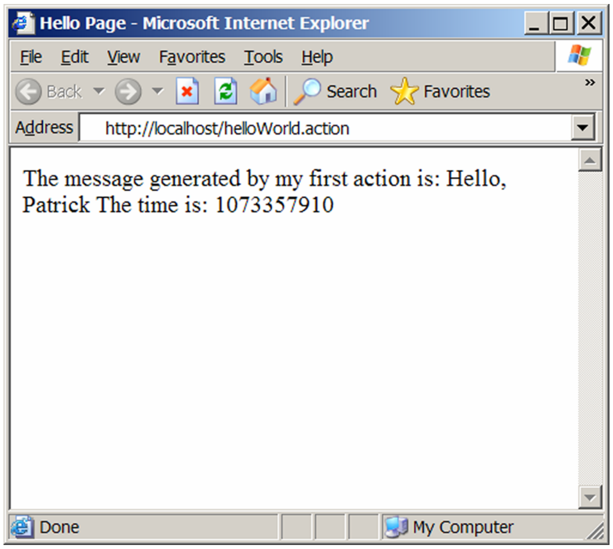
Next, you need to tweak the `HelloWorld` action to construct the personalized message. The new action code looks like this:

```
package ch2.example1;

 import com.opensymphony.xwork.Action;

 public class HelloWorld implements Action {
     private String message;
     private String name;

     public String execute() {
         message = "Hello, " + name + "!\n";
         message += "The time is:\n";
         message += System.currentTimeMillis();
         return SUCCESS;
     }

     public String getMessage() {
         return message;
     }

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

This code adds two things to the previous example. The first new item is a field and corresponding JavaBean-style set method named `name`. This must match exactly the name of the textfield you used in name.jsp. You also personalize the message that is constructed by including the name in the message during the `execute()` method. In WebWork, values are always set (via the `setXxx()` methods such as `setName()`) before the `execute()` method is called. That means you can use the variables in the `execute()` method while assuming they have already been populated with the correct value.

That's it! Recompile the action class, and start your application server. Now point your web browser to http://localhost/name.jsp, enter a name, and see that the message (shown in figure 2.2) is now personalized.

As easy as that was, a few problems can result. For instance, what if the user doesn't enter any data? The greeting will end up saying "Hello, !" Rather than show an ugly message, it might be better to send the user back to the original page and ask them to enter a real name. Let's add some advanced control flow to this action.


Hello Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/helloWorld.action
Done
My Computer

The message generated by my first action is: Hello, Patrick The time is: 1073357910

Figure 2.2
The new greeting is personalized. In this case, we used the name Patrick as the input.

2.5 Advanced control flow

Because you want the action to show either the message result (hello.jsp) or the original input form (name.jsp), you have to define another result in xwork.xml. You do this by changing the action entry to the following:

```
<action name="helloWorld"
         class="ch2.example1.HelloWorld">
     <result name="success">hello.jsp</result>
     <result name="input">name.jsp</result>
</action>
```

Now, if the `execute()` method in the `HelloWorld` action returns the `String` "input" (also defined as a constant, `INPUT`, in the method), the result of the action will be name.jsp rather than hello.jsp. In order to spice up this example a little more, let's also not allow the `String` "World" to be entered as a name. Editing the action to support these checks results in the following `execute()` method:

```
public String execute() {
     if (name == null || "".equals(name)
         || "World".equals(name)) {
         return INPUT;
     }

     message = "Hello," + name + "!\n";
     message += "The time is:\n";
     message += System.currentTimeMillis();
}
```

If the name doesn't pass your validation rules, you return a result code of `INPUT` and don't prepare the message. With just a couple lines of code, the control flow of this action has doubled the number of possible results the action can display. However, this still isn't as interesting as it could be, for two reasons:

- When you return to the `INPUT`, users can't see why they're back on this page. Essentially, there is no error message.
- Users can't tell what they originally entered as the name value. It might be nothing, or it might be *World*. It would be better if the input box displayed what the user original entered.

In order to address both concerns, let's modify the input result to display an error message. You can reuse the `message` property from the success result. Then, add logic to display an error message as well as make the textfield input display the previous name in the event of an error. With the modifications in place, name.jsp now looks like this:

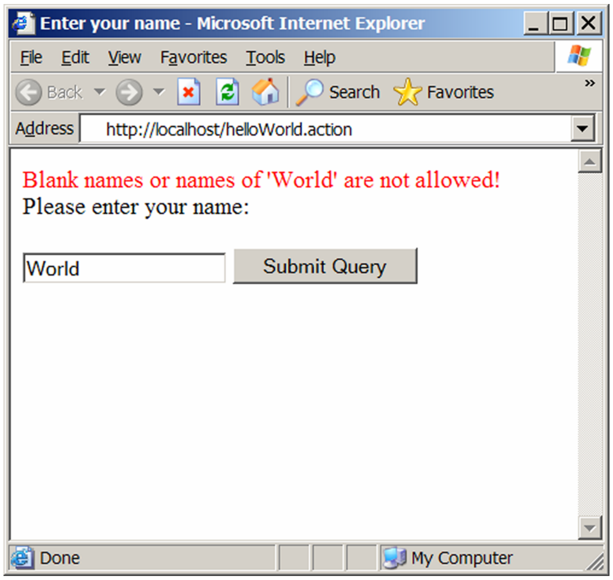
```
<%@ taglib prefix="ww" uri="webwork" %>
<html>
    <head>
        <title>Enter your name</title>
    </head>
    <body>
         <ww:if test="message != null">
             <font color="red">
                 <ww:property value="message"/>
             </font>
         </ww:if>
         Please enter your name:
         <form action="/helloWorld.action">
             <input type="textfield"
                    name="name"
                    value="<ww:property value="name"/>"/>
             <input type="submit"/>
         </form>
    </body>
</html>
```

This code adds two significant things to the JSP. First, if an error message exists, it's printed in a red font. You use the `ww:if` tag to see whether the `message` property exists; if it does, you print it. If a user goes to this page directly, the test fails, and no error message is reported—exactly the behavior you're striving for.

Second, you add a `value` attribute to the input HTML element. This attribute defines the default value to be displayed when the page is first loaded. Because the `ww:property` tag returns an empty `String` if a property isn't found, it also results in the desired behavior. After the action has been submitted once, the property exists. As such, if the `INPUT` result occurs, the value previously entered is displayed.

Finally, let's modify the action's `execute()` method one more time. This time, you'll make sure an error message is set in the `message` property just before the action returns with the `INPUT` code. The new method looks like this:

```
public String execute() {
    if (name == null || "".equals(name)
        || "World".equals(name)) {
        message = "Blank names or names of 'World' are not allowed!";
        return INPUT;
    }

    message = "Hello," + name + "!\n";
    message += "The time is:\n";
    message += System.currentTimeMillis();
    return SUCCESS;
}
```


Enter your name - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/helloWorld.action

Blank names or names of 'World' are not allowed!
Please enter your name:

World

Submit Query
Done
My Computer

Figure 2.3
The "Hello World" example when an invalid name is entered

The most important thing to note here is that the `execute()` method is prevented from finishing and returns with `INPUT` if the name fails the validation check.

With these small changes, you're ready to try the new behaviors. Start your application server, and point your browser to http://localhost/name.jsp. Enter in the value `World` for the name textfield and submit the form, and you should see the screen shown in figure 2.3.

This type of control flow, validation, and error reporting is often required for forms in web applications. Rather than leave the developer to handle these tasks, WebWork provides help that does almost all the work for you. In the next section, we'll explore how you can convert this example to use the reusable components that WebWork provides.

2.6 Letting WebWork do the work

One of the most common tasks developers want to perform when building web applications is to build input widgets, such as drop-down selection boxes or textfields, that all have a standard behavior. This includes displaying error messages about data as well as ensuring the original value is displayed in the case of an error. In addition, developers almost always want the widgets to have a common look and feel.

Because this is such a common need, WebWork provides support for it. Rather than code all the if-else error message logic in your JSPs and actions, you can take advantage of WebWork's helper classes and JSP tags to do the work for you. You'll now convert the previous example to use these classes and tags so you can see the most common way WebWork applications are built.

2.6.1 *Taking advantage of ActionSupport*

You'll start by converting the `HelloWorld` action to take advantage of a helper class called `ActionSupport`. Rather than implement the `com.opensymphony.xwork.Action` interface, you'll modify your class to extend `com.opensymphony.xwork.ActionSupport`. `ActionSupport` provides a method called `addFieldError()` that you can use to report error messages. Listing 2.2 shows the modified class.

Listing 2.2 HelloWorld.java, modified to extend `ActionSupport`

```java
package ch2.example1;

import com.opensymphony.xwork.ActionSupport;

public class HelloWorld extends ActionSupport {
    private String message;
    private String name;

    public String execute() {
        if (name == null || "".equals(name)
            || "World".equals(name)) {
            addFieldError("name",
                "Blank names or names of 'World' are not allowed!");
            return INPUT;
        }

        message = "Hello," + name + "!\n";
        message += "The time is:\n";
        message += System.currentTimeMillis();
        return SUCCESS;
    }

    public String getMessage() {
        return message;
    }

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

As you can see, little has changed. In fact, only two lines have been modified: The action now extends `ActionSupport`, and you no longer set the error message to the `message` property but rather call `addFieldError()`, which is provided by `ActionSupport`. This new method takes two arguments:

- `name`—The property to which this error message relates
- `message`—The error message itself

Because the error message is about the `name` property, you pass the `String` "name" as the first argument. The second argument is the same error message you previously assigned to the `message` property. Although there isn't a significant difference between the old action code and the new code, keep in mind that you're only reporting an error on a single property. Imagine that you have 10 or 15 properties—do you really want to maintain 10 or 15 error message properties as well?

2.6.2 *Intermediate modifications to the JSP*

The next step is to modify the JSP to take advantage of the new `ActionSupport` class structure. Because `ActionSupport` provides a method, `getFieldErrors()`, that returns a `java.util.Map` of error messages, the new JSP looks like this:

```
<%@ taglib prefix="ww" uri="webwork" %>
<html>
    <head>
        <title>Enter your name</title>
    </head>
    <body>
         <ww:if test="fieldErrors['name'] != null">
             <font color="red">
                 <ww:property value="fieldErrors['name']"/>
             </font>
         </ww:if>
         Please enter your name:
         <form action="/helloWorld.action">
             <input type="textfield"
                    name="name"
                    value="<ww:property value="name"/>"/>
             <input type="submit"/>
         </form>
    </body>
</html>
```

The code includes only one change: In the area where the error message is printed, the `value` attribute has changed from `message` to `fieldErrors['name']`. Right about now, you may be asking yourself, "Didn't they say this was supposed to be *better*?" You're right that this code looks more confusing. Fortunately, you're not finished changing the JSP. Don't worry; it gets easier—a lot easier.

We're showing you this JSP so you can begin to understand what's going on in the background. Because `ActionSupport` has a field called `fieldErrors` that is a `Map`, you can reference values in that `Map` by using the notation `map[key]`. Because the key is always the first argument of the `addFieldError()` method call, you pass in a `String` "name". Now that you have a basic understanding of what's going on, let's create the final version of the JSP.

2.6.3 Exploring the UI tag library

At this point, the change you made to the JSP has added more complexity rather than made it easier to work with. But you're not finished with it. WebWork comes with a complete UI tag library that helps you quickly write web applications with a standard look and feel. Let's add a UI tag to take care of the textfield input as well as display the error message:

```
<%@ taglib prefix="ww" uri="webwork" %>
<html>
    <head>
        <title>Enter your name</title>
    </head>
    <body>
         <ww:form action="helloWorld">
             <ww:textfield label="Please enter your name:"
                           name="name"/>
             <input type="submit"/>
         </ww:form>
    </body>
</html>
```

You may be looking at the previous JSP and wondering, "Where did everything go?" Without getting into too much detail (you'll learn all about UI tags in chapter 11), we'll explain what the new `ww:textfield` tag does.

The WebWork UI tag library contains a tag for every HTML form element (select, textfield, checkbox, and so on), and you can even write custom components easily. Each tag provides a standard label, error reporting, font coloring, and more. In this case, you've replaced the hand-coded HTML for error reporting, labeling, and font coloring with a single call to a UI tag that does all this for you.

These tags assume that the action extends `ActionSupport` (or at least provides a `getFieldErrors()` method) to show error messages for that field. This is the primary reason that 99 percent of all WebWork actions extend `ActionSupport` rather than implement the `Action` interface. Using the UI tag library and `ActionSupport`, you can create large, complex forms in almost no time.

2.7 Summary

In this chapter, we showed you how to build a simple web application and then expand on it to introduce more complex flow control and validation rules. By doing this, you discovered what a typical WebWork application looks like—one that takes advantage of `ActionSupport` and the UI tag library. Whereas the first two examples were simple in nature, the third example provided more functionality and a better look and feel, all while containing less code. The idea that less is more and that simplicity can be achieved through component reuse is a theme in WebWork that will reoccur throughout this book.