Pro Apache Beehive

KUNAL MITTAL AND SRINIVAS KANCHANAVALLY

Apress®

Pro Apache Beehive

Copyright © 2005 by Kunal Mittal and Srinivas Kanchanavally

ISBN (pbk): 1-59059-515-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Linda Marousek
Compositor: Susan Glinert Stevens
Proofreader: Sue Boshers
Indexer: Carol A. Burbo
Artist: Wordstop Technologies (P) Ltd, Chennai, India
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 4

Dissecting Java Page Flows

BEA originally launched the Java Page Flow technology with WebLogic Workshop. BEA also introduced a related technology, NetUI, which is a set of tag libraries that provide a binding between Java Page Flows (the controller layer) and Java Server Pages (the presentation layer). NetUI in Apache Beehive is a combination of the NetUI tag libraries and Java Page Flows. (In fact, we could have called this chapter "Dissecting NetUI.")

Note In this book, we'll refer to these two components as separate pieces. So, whenever we say *Page Flows*, we simply mean Java Page Flows. Whenever we say *NetUI*, we mean the NetUI tag libraries.

In this chapter, you'll look at the basic architecture of Java Page Flows and NetUI tags. You'll see the original Page Flows in WebLogic Workshop and then look at the Beehive version. You'll learn about the overall architecture, the classes, and the APIs you'll need to use to leverage Java Page Flows and NetUI.

The intent of this chapter is to introduce you to these technologies. You'll actually dig deeper into them in Chapter 5. Even if you've already worked with Page Flows in WebLogic Workshop, we recommend at least skimming through this chapter to get a basic overview of the differences between the two versions (WebLogic Workshop Page Flows and Beehive Page Flows). Even if you're an expert on the WebLogic Workshop version, or even if you're an expert on Beehive itself, you'll find this chapter useful as a ready-to-use reference/refresher.

Introducing Java Page Flows

In the typical Model-View-Controller (MVC) design pattern, Java Page Flows form the controller layer. They're assisted by the NetUI tag libraries in the presentation layer. Java Page Flows are built on top of Struts—which, as you know, is one of the most widely adopted MVC frameworks available today. So, why not just use Struts?

Java Page Flows leverage the core functionality of Struts but remove a lot of the grunt work you have to do with Struts. By *grunt work*, we mean managing the deployment configuration files (such as the `struts-config.xml` file). The original version of Page Flows from BEA introduced a declarative programming language that was automatically generated and maintained by

WebLogic Workshop. The Apache Beehive version of Page Flows uses JSR 175 for its metadata definition. (You saw the details of this in Chapter 3.)

As mentioned, Page Flows leverage all the features of Struts, such as the validation framework. You'll see this in more detail in Chapter 5. You can actually have a single Web application that has a combination of Struts and Page Flows.

So, let's actually look at a Page Flow.

Page Flows in WebLogic Workshop

This book is not about WebLogic Workshop, so we won't go into the details of how you start building Page Flows in WebLogic Workshop. Let's just assume that you built a simple HelloWorld Page Flow using WebLogic Workshop (see Listing 4-1).

■**Note** See the BEA Web site (`http://www.bea.com`) for information on how to download and install BEA WebLogic Workshop 8.1. See the documentation on the BEA developer site (`http://edocs.bea.com`) to learn how to work with BEA Page Flows.

Listing 4-1. *`helloworld`.jpf in WebLogic Workshop*

```
package helloworld;
import com.bea.wlw.netui.pageflow.Forward;
import com.bea.wlw.netui.pageflow.PageFlowController;

/**
 * @jpf:controller
 * @jpf:view-properties view-properties::
 * <!-- This data is autogenerated.
* Hand-editing this section is not recommended. -->
 * <view-properties>
 * <pageflow-object id="pageflow:/helloworld/HelloWorldController.jpf"/>
 * <pageflow-object id="action:begin.do">
 *   <property value="80" name="x"/>
 *   <property value="100" name="y"/>
 * </pageflow-object>
 * <pageflow-object id="forward:path#success#helloworld.jsp#@action:begin.do@">
 *   <property value="44,20,20,60" name="elbowsX"/>
 *   <property value="92,92,-4,-4" name="elbowsY"/>
 *   <property value="West_1" name="fromPort"/>
 *   <property value="North_1" name="toPort"/>
 *   <property value="success" name="label"/>
 * </pageflow-object>
 * <pageflow-object id="page:helloworld.jsp">
 *   <property value="60" name="x"/>
 *   <property value="40" name="y"/>
 * </pageflow-object>
 * </view-properties>
 * ::
 */
public class HelloWorldController extends PageFlowController
{


    // Uncomment this declaration to access Global.app.
    //
    //     protected global.Global globalApp;
    //

    // For an example of Page Flow exception handling,
    // see the example "catch" and "exception-handler"
    // annotations in {project}/WEB-INF/src/global/Global.app

    /**
     * This method represents the point of entry into the Page Flow
     * @jpf:action
     * @jpf:forward name="success" path="helloworld.jsp"
     */
    protected Forward begin()
    {
        return new Forward("success");
    }
}
```

Notice that this code snippet is mostly full of Java comments. As described in Chapter 3, these are the different annotations that support the execution of the actual Page Flow.

All Page Flows have a `begin` method. This is similar to the `main` method in a Java class. In this example, the `begin` method does only one thing: it directs you to the `helloworld.jsp` page. Listing 4-2 shows this JSP.

Listing 4-2. *`helloworld.jsp` in WebLogic Workshop*

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
    <head>
        <title>
            WebLogic Workshop - Hello World
        </title>
    </head>
    <body>
        <p>
           Hello World !!
        </p>
    </body>
</netui:html>
```

This JSP is simple enough. You can easily compile and deploy this Page Flow from within WebLogic Workshop and see its execution.

■ **Note** BEA WebLogic 9.*x* (`http://e-docs.bea.com`) will be based on the Apache Beehive version of Page Flows rather than the proprietary version of Page Flows you'll find in BEA WebLogic 8.1.

Now, let's see the same Page Flow in Apache Beehive.

Page Flows in Apache Beehive

The HelloWorld example in Beehive looks a little different from the WebLogic Workshop version. Let's first look at the controller in Listing 4-3.

Listing 4-3. *`helloworld.jpf` in Apache Beehive*

```
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;
import org.apache.beehive.netui.pageflow.Forward;

@Jpf.Controller (

        simpleActions= {

                @Jpf.SimpleAction (name="cancel", path="begin.do")
                }
)

public class HelloWorldController extends PageFlowController

{
```

```
@Jpf.Action (
        forwards= {
                @Jpf.Forward (name="success", path="helloworld.jsp")
                }
        )
 public Forward begin()

 {
        return new Forward("success"); }
```

You'll immediately notice that this version of the Java Page Flow is a lot shorter and crisper. All the Javadocs annotations at the beginning of the class code are no longer needed in the Apache Beehive version. Listing 4-4 shows the JSP that goes with this controller.

Listing 4-4. *`helloworld.jsp` in Apache Beehive*

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-html-1.0" prefix="netui"%>
<netui:html>
  <head>
    <title>beehive - hello world</title>
    <netui:base/>
  </head>
  <netui:body>
    <p>
      Hello World !!
      <br>
     </p>
  </netui:body>
</netui:html>
```

The “How to Run the Sample Code” sidebar will show you how to set up and run this example in your own environment.

HOW TO RUN THE SAMPLE CODE

Use the following steps to set up and run the HelloWorld example in your own environment.

Make a Project Folder

First, make sure you've read Appendix A. Then, on your C: drive, create a directory named `beehive-projects`. In the `beehive-projects` directory, create a directory named `helloworld`. Before proceeding, confirm that the following directory structure exists:

```
C: \
  beehive-projects
    helloworld
```

Continued

Copy Runtime JARs to the Project Folder

Copy the folder `BEEHIVE_HOME/samples/netui-blank/resources` into your project folder, `C:\beehive_projects\helloworld`. `BEEHIVE_HOME` is the top-level folder of your Beehive installation, as explained in Appendix A.

Copy the folder `BEEHIVE_HOME/samples/netui-blank/WEB-INF` into your project folder, `C:\beehive-projects\helloworld`.

Now, assemble the runtime resources for your Page Flow application. The runtime JARs include the Page Flow runtime, the `NetUI` tag library, and so on. You can load these resources into your project's `WEB-INF/lib` folder using the following Ant command at the command prompt:

```
ant -f %BEEHIVE_HOME%\ant\buildWebapp.xml
    -Dwebapp.dir=C:\beehive-projects\helloworld deploy.beehive.webapp.runtime
```

This command will copy all JAR files to the `WEB-INF/lib` directory. Next, create the controller file, the central file for any Page Flow. Then, in the directory `C:/beehive-projects/helloworld`, create a file named `HelloWorldController.jpf`. In a text editor (or your IDE of choice), open the file `HelloWorldController.jpf`. In the directory `C:/beehive-projects/helloworld`, create a file named `helloworld.jsp`.

Compile and Deploy the Page Flow

You're now ready to compile the Page Flow and deploy it to Tomcat. Start the Tomcat server. Using the command shell opened in the previous step, at the command prompt, enter the following:

```
ant -f  %BEEHIVE_HOME%\ant\buildWebapp.xml
   -Dwebapp.dir=C:\beehive-projects\helloworld
   -Dcontext.path=helloworld  build.webapp  deploy
```

To undeploy the application, use the following Ant command:

```
ant  -f %BEEHIVE_HOME%\ant\buildWebapp.xml
   -Dwebapp.dir=C:\beehive-projects\helloworld
   -Dcontext.path=helloworld  undeploy
```

Let's now look at a more detailed example. In this example, you'll extend the HelloWorld controller to actually have some basic "login" functionality.

Figure 4-1 shows the basic functionality you'll implement in the HelloWorld controller.

For this example, you'll implement three actions—`begin`, `processLogin`, and `showLogin`—that go to three different JSPs. There's a login form where the user can fill in their username and password. When the user submits the form, they will be directed to `success.jsp`. Listing 4-5 shows the controller code for this simple Page Flow.

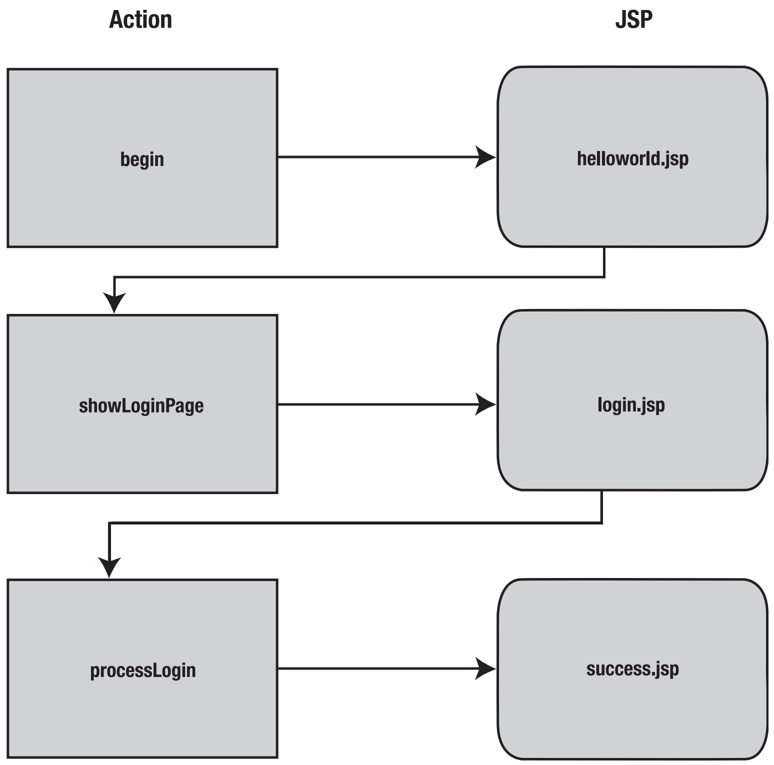

Figure 4-1. *Basic login process in the HelloWorld controller*

Listing 4-5. `helloworld.jpf` *Extended for Login Functionality*

```
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;
import org.apache.beehive.netui.pageflow.Forward;

import helloworld.forms.LoginForm;


@Jpf.Controller (

        simpleActions= {

                @Jpf.SimpleAction (name="cancel", path="begin.do")
                }
)
```

```
public class HelloWorldController extends PageFlowController

{

@Jpf.Action (
        forwards= {
                @Jpf.Forward (name="success", path="helloworld.jsp")
                }
        )
 public Forward begin()

 {
        return new Forward("success");
 }


@Jpf.Action (
        forwards= {
                @Jpf.Forward (name="success", path="login.jsp")
                }
        )
  public Forward showLoginPage()

 {
        return new Forward("success"); }

 @Jpf.Action(
         forwards = {
             @Jpf.Forward(name = "success", path = "success.jsp")
         }
     )
     public Forward processLogin(LoginForm form)
     {
         System.out.println("User Name: " + form.getUsername());
         System.out.println("Password: " + form.getPassword());
         return new Forward("success");
     }


}
```

To make this work, add just one line of code to `helloworld.jsp`:

```
<netui:anchor action="showLoginPage">Login</netui:anchor>
```

This translates to a link that the user can click in `helloworld.jsp`. This will trigger the `showLoginPage` action and take the user to `login.jsp`. Listing 4-6 shows `login.jsp`.

Listing 4-6. *`login.jsp`*

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-html-1.0" prefix="netui"%>
<netui:html>
  <head>
    <title>Login</title>
    <netui:base/>
  </head>
  <netui:body>
    <p>
     <p>
          <netui:form action="processLogin">
          <p>User Name:
                <netui:textBox dataSource="actionForm.username"/>
             <p>Password:
             <netui:textBox dataSource="actionForm.password"
                           password="true" size="20"  />
            <dataSource="actionForm.name"/>
            <p><netui:button type="submit">Submit</netui:button>
            <netui:button  action="cancel">Cancel</netui:button>
          </netui:form>
        </p>

    </p>
  </netui:body>
</netui:html>
```

The login JSP introduces the concept of a form. This is a basic form that looks a lot like a JavaBean or a Struts form class. It has basic getters and setters for the fields you've displayed in the JSP. NetUI and Page Flows provide automatic binding between the form variables and the JSP fields. (You'll learn more about this in Chapter 5.) Listing 4-7 shows the `LoginForm` class.

Listing 4-7. *`LoginForm.java`*

```
package helloworld.forms;

import org.apache.beehive.netui.pageflow.FormData;

public class LoginForm extends FormData
{

    private String username;
    private String password;
    public void setUsername(String name)
    {
        this.username = name;
    }

    public String getUsername()
    {
        return this.username;
    }

    public void setPassword(String password)
    {
        this.password= password;
    }

    public String getPassword()
    {
        return this.password;
    }
}
```

The example you've just seen is very basic. However, it will help you identify the different pieces of the Page Flow architecture.

Introducing Page Flow Architecture

In the following sections, we'll talk about the basic architecture and components that make up Java Page Flows and the NetUI tags.

Page Flow Components

The different components of a Page Flow are as follows:

- Controllers
- Form classes

Controllers

The `Jpf.Controller` annotation is the meat of a Page Flow. It's just a file that contains Java code and annotations. The extension of this file is `.jpf`. As you saw in the previous example, a controller consists of several actions. This is unlike Struts—where one action is one class. You can think of a controller as a collection of action classes. The different annotations of a Page Flow are as follows:

- `Jpf.Catch[]`: Exceptions that the controller catches. We always recommend catching at least the `Exception` class to handle any unexpected/unhandled exceptions.
- `Jpf.Forward[]`: The different forwards. Each action has one or many forwards.
- `global forwards`: Any global forwards. For example, when an exception is caught, you might need a global forward. We always recommend having at least one of these go to some error page when an exception is thrown, as described in the `Jpf.Catch` item.
- `boolean loginRequired`: Does this controller require the user to be logged in to execute the actions defined in this Page Flow?
- `Jpf.MessageResource[] messageResources`: Which message resources to use for error messages. This is similar to the Struts message resources.
- `Jpf.MultipartHandler multipartHandler`: Does this controller need to access multipart forms?
- `boolean nested`: Is this Page Flow a nested Page Flow?
- `boolean readOnly`: The actions do not modify any member variables.
- `String[] rolesAllowed`: The roles that can access actions in this Page Flow.
- `Jpf.SimpleAction[] simpleActions`: The simple actions in this Page Flow.
- `boolean singleton`: Is this Page Flow a singleton?
- `String strutsMerge`: The location of the Struts merge file.
- `Jpf.ValidatableBean[] validatableBeans`: The validation rules for the beans.
- `String validatorMerge`: The location of the `ValidatorPlugIn` merge file.

■ **Caution** Since Apache Beehive is still in early development, we recommend looking at the Javadocs online for the latest and greatest list of methods and functionality. See `http://incubator.apache.org/beehive/apidocs/classref_pageflows/index.html`.

The `PageFlowController` class provides more than just actions. Figure 4-2 shows the basic relations between the `FlowController` parent class and the `PageFlowController` class. (Note that the figure shows only some of the methods in the classes; see the Javadocs for a complete listing of all the methods available in these classes.)

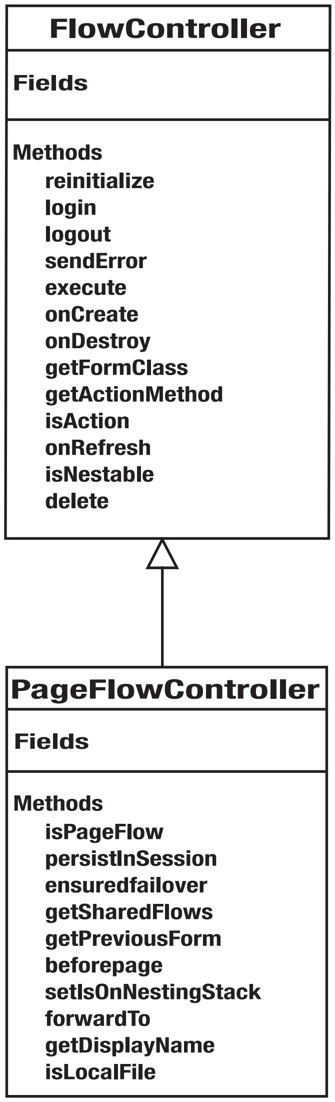

Figure 4-2. *Class diagram of `FlowController` and `PageFlowController`*

Let's look at a few of the methods that you might use more regularly than others:

- `afterAction`: This method is a callback that occurs after any user action method is invoked.
- `beforeAction`: This method executes before any action executes. It's sort of a `preProcess` method for an action.
- `onCreate`: This executes when the Page Flow is created; you can use it to initialize any instance variables for the Page Flow.

- `onDestroy`: This executes when the Page Flow is destroyed; you can use it to clean up any variables.
- `onRefresh`: This is specifically important in a portal environment when no action needs to be executed and you'd rather just render a previously displayed JSP.
- `isNestable`: This determines whether this Page Flow can be nested.
- `isSingleton`: This determines whether this Page Flow is a singleton.

■**Tip** As you start working with Page Flows, refer to the `PageFlowUtils` class. It provides a bunch of helper methods that you'll find useful.

Action and Forward Classes

Integral parts of using a Page Flow controller are the action classes and the forward classes. Let's quickly take a look at what they offer.

Action

The following are the annotations that are available for an action class:

- `Jpf.Catch[]`: The different exceptions caught by this action.
- `Jpf.Forward[]`: The different forwards defined by this action.
- `boolean loginRequired`: Does this action require that the user be logged in?
- `boolean readOnly`: A guarantee that this action does not change any Page Flow variables.
- `String[] rolesAllowed`: The roles that can access this action.
- `String useFormBean`: The form bean that this action class uses.
- `Jpf.ValidatableProperty[] validatableProperties`: The properties of the form bean that need to be validated.
- `Jpf.Forward validationErrorForward`: The forward to use when there is any validation error.

Forward

The following are the annotations offered by the forward classes:

- `Jpf.ActionOutput[] actionOutputs`: List of action outputs.
- `boolean externalRedirect`: Redirect to some external action
- `Jpf.NavigateTo navigateTo`: The page or action to navigate to
- `String outputFormBean`: Output form bean

- `Class outputFormBeanType`: Output form bean type
- `String path`: The path to forward too, usually a JSP
- `boolean redirect`: Redirect or not
- `boolean restoreQueryString`: Whether the original query string will be restored on a rerun of a previous action
- `String returnAction`: The action to be executed on the original Page Flow

■ **Tip** We recommend you look at the different methods on the `Forward` object. They will prove to be useful as you start building complex Page Flow applications.

NetUI Components

NetUI is a set of tag libraries that you will use as part of your JSPs. These tag libraries are JSP 2.0 complaint. Three tag libraries make up NetUI:

- `NetUI`
- `NetUI-data`
- `NetUI-template`

■ **Caution** The `NetUI-data` and `NetUI-template` tag libraries depend on the `NetUI` tag libraries. All the base classes for the three tag libraries are provided as part of the `NetUI` (HTML) tag library.

The basic functionality in these tag libraries is to simplify JSP development and provide automatic data binding between the view and controller layers. These tags come with JavaScript support, so you can work with them like you would the standard HTML tags (input, select, and so on).

■ **Note** You will see examples of how to use each of the tags in the next chapter.

NetUI

The `NetUI` name is a little misleading. Think of this tag library as `NetUI-html`. That makes it clearer, doesn't it? This tag library contains the tags similar to the `struts-html` tag library. Table 4-1, which comes straight from the Javadocs, shows the tags in this library. As you'll see, this library contains the standard tags that you might use with vanilla HTML development.

Note The reason I've simply cut and paste the information from the Javadocs is because, at the time of writing this book, Apache Beehive is still in the beta stage. Therefore, some of these methods might change. Visit the Beehive documentation page for the latest Javadocs at `http://incubator.apache.org/beehive/reference/taglib/index.html`.

Table 4-1. *NetUI Tag Library*

Tag	Description
`<netui:anchor>`	Generates an anchor that can link to another document or invoke an action method in the controller file
`<netui:attribute>`	Adds an attribute to the parent tag rendered in the browser
`<netui:base>`	Provides the base for every URL on the page
`<netui:bindingUpdateErrors>`	Renders the set of error messages found during the process of resolving data binding expressions (`{pageFlow.firstname}`, `{request.firstname}`, and so on)
`<netui:body>`	Renders an HTML `<body>` tag with the attributes specified
`<netui:button>`	Renders an HTML button with the specified attributes
`<netui:checkBox>`	Generates a single HTML checkbox
`<netui:checkBoxGroup>`	Handles data binding for a collection of checkboxes
`<netui:checkBoxOption>`	Renders a single HTML checkbox within a group of checkboxes
`<netui:content>`	Displays text or the result of an expression
`<netui:error>`	Renders an error message with a given error key value if that key can be found in the `ActionErrors` registered in the `PageContext` at `org.apache.struts.action.Action.ERROR_KEY`
`<netui:errors>`	Renders the set of error messages found in the `ActionErrors` registered in the `PageContext` at `org.apache.struts.action.Action.ERROR_KEY`
`<netui:exceptions>`	Renders exception messages and stack traces inline on the JSP
`<netui:fileUpload>`	Renders an HTML input tag with which users can browse, select, and upload files from their local machines
`<netui:form>`	Renders an HTML form that can be submitted to a Java method in the controller file for processing
`<netui:formatDate>`	Renders a formatter used to format dates
`<netui:formatNumber>`	Renders a formatter used to format numbers
`<netui:formatString>`	Renders a formatter used to format strings

Table 4-1. *NetUI Tag Library (Continued)*

Tag	Description
`<netui:hidden>`	Generates an HTML hidden tag with a given value
`<netui:html>`	Renders an `<html>` tag
`<netui:image>`	Renders an HTML `<image>` tag with the specified attributes
`<netui:imageAnchor>`	Generates a hyperlink with a clickable image
`<netui:imageButton>`	Renders an `<input type="image">` tag with the specified attributes
`<netui:label>`	Associates text with an input element in a form
`<netui:parameter>`	Writes a name-value pair to the URL or the parent tag
`<netui:parameterMap>`	Writes a group of name-value pairs to the URL or the parent tag
`<netui:radioButtonGroup>`	Renders a collection of radio button options and handles the data binding of their values
`<netui:radioButtonOption>`	Generates a single radio button option in a group of options
`<netui:rewriteName>`	Allows a name, typically either an `id` or `name` attribute, to participate in URL rewriting
`<netui:rewriteURL>`	Allows a tag name, typically either an `id` or `name` attribute, to participate in URL rewriting
`<netui:scriptContainer>`	Acts as a container that will bundle JavaScript created by other `<netui...>` tags and outputs it within a single `<script>` tag
`<netui:scriptHeader>`	Writes the `<script>` that JavaScript will include in the HTML `<head>` tag
`<netui:select>`	Renders an HTML `<select>` tag containing a set of selectable options
`<netui:selectOption>`	Renders a single `<option>` tag
`<netui:span>`	Generates styled text based on a `String` literal or data binding expression
`<netui:textArea>`	Renders an HTML `<input>` tag of type `"text"`
`<netui:textBox>`	Renders an HTML `<input type="text">` tag
`<netui:tree>` `<netui:treeContent>` `<netui:treeHtmlAttribute>` `<netui:treeItem>` `<netui:treeLabel>` `<netui:treePropertyOverride>`	Renders a navigable tree of `TreeElement` tags

NetUI-data

The `NetUI-data` tag library is used to bind data from forms and the controller to the JSP. It allows you to quickly display lists of data (such as search results). See Table 4-2, which shows the Javadocs information about this tag library.

Table 4-2. *`NetUI-data` Tag Library*

Tag	Description
`<netui-data:anchorColumn>`	
`<netui-data:callMethod>`	Calls methods on any Java classes
`<netui-data:callPageFlow>`	Calls methods on the controller file (which is a JPF file) in the same directory as the JSP
`<netui-data:caption>`	
`<netui-data:cellRepeater>`	Renders individual cells of an HTML table
`<netui-data:columns>`	
`<netui-data:configurePager>`	
`<netui-data:dataGrid>`	
`<netui-data:declareBundle>`	Declares a `java.util.ResourceBundle` as a source for displaying internationalized messages
`<netui-data:declarePageInput>`	Declares variables that are passed from the controller file to the JSP
`<netui-data:footer>`	
`<netui-data:getData>`	Evaluates an expression and places the result in the `javax.servlet.jsp.PageContext` object, where the data is available to JSP scriptlets
`<netui-data:imageColumn>`	
`<netui-data:literalColumn>`	
`<netui-data:message>`	Provides a message schema, which can be parameterized to construct customizable messages
`<netui-data:messageArg>`	Provides a parameter value to a message schema
`<netui-data:methodParameter>`	Provides an argument to a method-calling tag
`<netui-data:pad>`	Sets the number of items rendered by a tag
`<netui-data:renderPager>`	
`<netui-data:repeater>`	Iterates over a data set to render it as HTML
`<netui-data:repeaterFooter>`	Renders the footer of a `Repeater` tag
`<netui-data:repeaterHeader>`	Renders the header of a `Repeater` tag
`<netui-data:repeaterItem>`	Renders an individual item in the data set as it's iterated over by the `Repeater` tag
`<netui-data:serializeXML>`	Serializes an XMLBean into the output of a JSP in order to move data to the browser for data binding

NetUI-template

The `NetUI-template` tag library is used to create subsections (or templates) from your JSPs. See Table 4-3, which displays the Javadocs information about this tag library.

Table 4-3. *`NetUI-template` Tag Library*

Tag	Description
`<netui-template:attribute>`	Defines a property placeholder within a template
`<netui-template:divPanel>`	Creates an HTML `<div>` tag that may contain additional tags
`<netui-template:includeSection>`	Defines a content placeholder within a template
`<netui-template:section>`	Sets HTML content inside placeholders defined by an `IncludeSection` tag
`<netui-template:setAttribute>`	Sets a property value in a template page
`<netui-template:template>`	Points a content page at its template page

You've just seen a brief overview of Page Flows and NetUI. Now let's see how all this plays together.

Reviewing Page Flow Architecture

The best way to explain the overall architecture of Page Flows and NetUI is to map these to the standard MVC model, as shown in Figure 4-3.

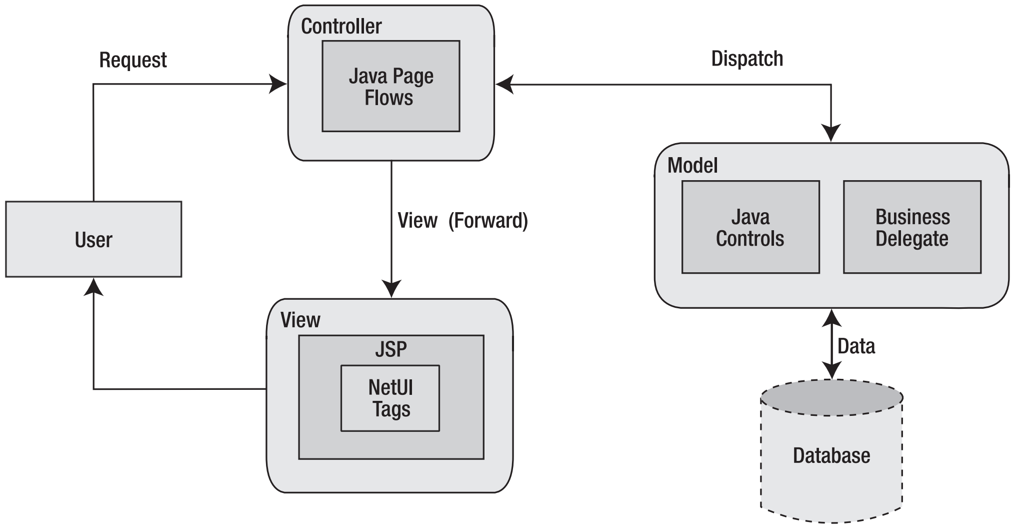

Figure 4-3. *MVC architecture of Page Flows and NetUI*

Think of the controller bucket as being the Page Flow controllers. If you're familiar with Struts, this bucket is filled by Struts actions. The *view* is a collection of JSPs and some tag libraries, in this case the NetUI tag libraries. In Struts, it would be the Struts tag libraries. The model layer is not really predetermined by Page Flows. As part of the Apache Beehive project, there is a technology called *Controls*. This is a model layer technology, which we'll discuss in Chapter 6.

However, for purposes of Java Page Flows, the model layer could be anything. You could obviously use Controls. Or, you could have a set of Java classes that serve as business delegates, which then interact with your EJBs, DAOs, and other classes.

Throughout the chapter, I've alluded to the real advantages of Page Flows over Struts:

- *Ease of use*: The main development savings between Struts and Page Flows is the JSR 175 metadata support. While developing Page Flows, you don't need to manually maintain the `struts-config` files.
- *Data binding*: With Page Flows and NetUI, you get automatic data binding between the form variables and the form fields in the JSP.
- *Exception handling*: This goes back to the annotations. You can define how all your exceptions get handled using the annotations. As a best practice, we recommend always catching the `Exception` class at the Page Flow level. This allows you to handle any otherwise uncaught exceptions.
- *Nested Page Flows*: The whole concept of nested Page Flows is new. We'll explain this in Chapter 5.
- *State management*: Page Flows automatically maintain state. This feature is even more important if you're working with portals.
- *Portal use*: Page Flows were originally developed for portal development. Thus, a lot of the features are targeted toward portal projects.
- *Service orientation*: The integration of Page Flows with Java Controls leads to a more service-oriented approach for application development.

So, What's Next?

In this chapter, you saw the basic components that make up Java Page Flows and NetUI. You also looked at a quick example and then drilled down into the overall architecture of these technologies. Now, let's really see some examples that show you how to work with these technologies and explore all the features. We recommend jumping to Appendix B to learn how to set up the Eclipse Pollinate IDE to build Page Flows and then turning to Chapter 5.