BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Table of Contents

Developing Web Applications

Enterprise web applications today can easily contains hundreds, if not thousands of pages. These pages should not only look great visually, but also offer services to customers that require the implementation of complex business logic. Managing a complex web site can be a daunting task, especially where the business logic is implemented directly in the web pages, and changing the logic requires many edits in many locations.

WebLogic Workshop provides you with the tools to manage complex web applications using JavaServer Pages (JSPs) and Page Flows. Separation of presentation and business logic allows for modularity of business logic implementation, such that the impact of changing business logic can be minimal. Furthermore, this separation allows the application developer to concentrate on implementing the business process using Java controls and EJBs, while the web developer can focus on the presentation. Page flows provide the navigational control, allowing a web application architect to easily design the flow between the JSP pages in the web application.

Topics Included in This Section

Guide to Building Page Flows

This development guide explains the key concepts involved in developing web applications using page flows, JavaServer Pages (JSPs), and WebLogic Workshop.

Exception Handling and Validating User Input

Explains how to handle errors at arise in a web application and how to validate data submitted by users.

Working with Struts Applications

Explains how to use your existing Struts applications with Page Flow and Portal applications.

Web Application Reference

The reference section provides details on page flow annotations, JSP tag syntax, and Flow View icons.

Related Topics

Getting Started Tutorial: Web Applications

This tutorial provides an entry−level introduction into the WebLogic Workshop environment for developing web applications that contain page flows and JSPs.

Tutorial: Page Flow

This tutorial provides more advanced tour of the WebLogic Workshop environment for developing web applications. This tutorial will teach you the basics of Page Flow technology, as well as more advanced features such as data binding, Java controls and security.

Page Flow and JSP Samples

This section discusses a number of pre−built sample web applications that demonstrate key concepts in page flow and JSP technology.

How Do I... topics for Page Flows and JSPs

This 'How Do I...?' section presents a number of hands−on examples to building page flows and data binding.

Guide to Building Page Flows

WebLogic Workshop provides you with the tools to develop web applications using JavaServer Pages (JSPs) and Page Flows, separating presentation, business logic, and navigational control to manage complexity. The topics in this section discuss these concepts and provide detailed information on how to use WebLogic Workshop to develop web applications based on page flows and JSP pages.

Topics Included in This Section

Introduction to Building Web Application with Page Flows

Introduces the page flow features in WebLogic Workshop. Walks you through a simple example.

Advantages of Using Page Flows

Outlines the advantages of page flow features, especially in comparison to using "pure Struts."

Using Form Beans to Encapsulate Data

Introduces data binding and describes how forms beans are used to separate data from presentation.

Form Bean Scopings

Explains the different scopings available for Form Beans.

Using Data Binding in Page Flows

Describes the various data binding contexts that are available in WebLogic Workshop.

Designing User Interfaces in JSPs

Discusses the most common presentation elements that you will need to design your web pages, such as buttons, labels, and checkboxes.

Presenting Complex Data Sets in JSPs

Describes the JSP data binding tags used to dynamically display large sets of data in a JSP.

Declaring Page Inputs

Explains how to declare variables that will be passed from the page flow controller to the JSP page.

Calling Web Services and Custom Java Controls From a Page Flow

Describes how to call a web service or Custom Java Control from a Page Flow.

Handling Images and Binary Data in Page Flows

Describes how to handle and display binary data, such as image files, in Page Flows and JSP pages.

Using JavaScript in Page Flow and Portal Applications

Explains how to access Page Flow tags with JavaScript, especially when the Page Flow application is contained in a Portal.

A Detailed Page Flow Example

Discusses how you can use WebLogic Workshop to quickly build the framework of a web application, which implements the major navigational flows and other major business logic, and can be used for early testing.

Best Practices for Page Flows

Describes a number of best practices for building Page Flows and JSP pages.

Updating Libraries with Service Packs

Describes how to update Page Flow libraries after you install a new WebLogic Workshop Service Pack.

Configuring Page Flow Applications

Explains how to optimize performance and avoid exhaustion of server resources.

Related Topics

Getting Started Tutorial: Web Applications

Tutorial: Page Flow

Page Flow and JSP Samples

Introduction to Building Web Applications with Page Flows

As a web application developer, WebLogic Workshop *page flows* are designed to make your job easier. This topic introduces page flow concepts and components, and discusses the following:

- Why Use Page Flows?
- How Does a Page Flow Work?
- Components of the Page Flow Model
- Flow, Action, and Source Views
- Navigation: a Simple Example
- Submitting Data: a Simple Example
- Displaying Data: a Simple Example

If you want to jump right in and build a page flow now, see the introductory tutorial Getting Started: Web Applications or the more advanced tutorial Tutorial: Page Flow. If you want to see samples that demonstrate common page flow and JSP features, see Page Flow and JSP Samples.

Why Use Page Flows?

By using page flows, you can avoid making the typical mistakes that often happen during web application development, by separating presentation, business logic implementation, and navigational control. In many web applications, web developers using JSP (or any of the other dynamic web languages such as ASP or CFM) combine presentation and business logic in their web pages.

As these applications grow in complexity and are subject to continual change, this practice leads to expensive, time−consuming maintenance problems, caused by:

- Limited reuse of business logic
- Cluttered JSP source code
- Unintended exposure of business−logic code to team members who focus on other aspects of web development, such as content writers and visual designers

Page flows allow you to separate the user interface code from navigational control and other business logic. User interface code can be placed where it belongs, in the JSP files. Navigational control can be implemented easily in a page flow's single controller file, which is the nerve center of your web application. A controller file is a special Java file that uses a JPF file extension. Business logic can be implemented in the page controller file, or in Java controls that you call from JPF files.

The separation of presentation and business logic offers a big advantage to development teams. For example, you can make site navigation updates in a single JPF file, instead of having to search through many JSP files and make multiple updates. In WebLogic Workshop you can as easily navigate between page flows as between individual JSP pages. This allows you to group related web pages under one page flow, and create functionally modular web components. This approach to organizing the entities that comprise web applications makes it much easier to maintain and enhance web applications by minimizing the number of files that have to be updated to implement changes, and lowers the cost of maintaining and enhancing applications.

Another advantage of page flows is that an instance of the page flow controller class is kept alive on a per−user−session basis while the user is navigating within the scope of the page flow. This instance ends when the user exits from the page flow. You can use instance member variables in page flow classes to hold user session state.

For more information about the advantages of page flows, especially in comparison to "pure Struts" applications, see Advantages of Using Page Flows.

How Does a Page Flow Work?

A page flow is a Java class, called the "controller" class, that controls the behavior of a web application through the use of specially designed annotations and methods. The directory that contains the controller class also includes the JavaServer Pages (JSPs) used in the page flow. For a JSP to be considered part of a page flow, it must reside within the page flow directory. The JSP files use special tags which help bind to data and business logic actions. The action methods in the controller file implement code that can result in site navigation, passing data, or invoking back−end business logic via controls. Significantly, the business logic in the controller class is separate from the presentation code defined in the JSP files.

The overall purpose of a page flow is to provide you with an easy−to−use framework for building dynamic, sophisticated web applications. WebLogic Workshop provides graphical and code−level tools to simplify the development cycle. While page flows give you access to advanced features of J2EE, you do not have to be a J2EE expert to quickly develop and deploy Java−based applications built on page flows. Wizards can be used to create different types of page flows, generating the Java and JSP files that serve as a starting point for your work. Graphical tools let you draw the relationships between web components in a controller's Flow View. In Source View, syntax completion, validation, and other programmer's aids reduce the amount of work required to get your application running.

Note: WebLogic Workshop's web application functionality is built on Struts, which is an open−source framework for building web applications in a J2EE environment.

Components of the Page Flow Programming Model

Page flows implement user interface control logic, and contain:

- Action Methods
- Form Beans
- Forward Objects
- The <netui...> Tag Library

Action Methods

In the controller class, action methods are methods that are annotated with a @jpf:action tag.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="page_A.jsp"
 */
protected Forward begin()
{
    return new Forward( "success" );
}
```

Action methods can perform several functions. They can (1) implement navigation decisions, (2) move data into and out of JSP pages, and (3) invoke back−end business logic via calls to controls.

Form Beans

Form Beans are Java data structures that correspond to HTML forms. When a user submits data from an HTML form, the data is stored in a Form Bean instance. Once the data is stored in a Form Bean instance, the data is available for processing by the action methods in the controller file. Form Bean instances (containing submitted data) are typically passed as parameters to action methods.

```
/**
 * @jpf:action
 */
protected Forward ProcessData( MyFormBean form )
{

    //Submitted data is processed here...

}
```

Form Beans are simple Java classes contained within the controller file. They consist of some number of fields with setter and getter methods associated with those fields. Below is a Form Bean with one field, the String name, and setter and getter methods for that field. Form Bean must extend the class com.bea.wlw.netui.pageflow.FormData.

```
public static class MyFormBean extends FormData
{
    private String name;

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

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

Forward Objects

Forward objects are returned by action methods. They can be used to control navigation and pass data throughout the application.

The <netui> Tag Library

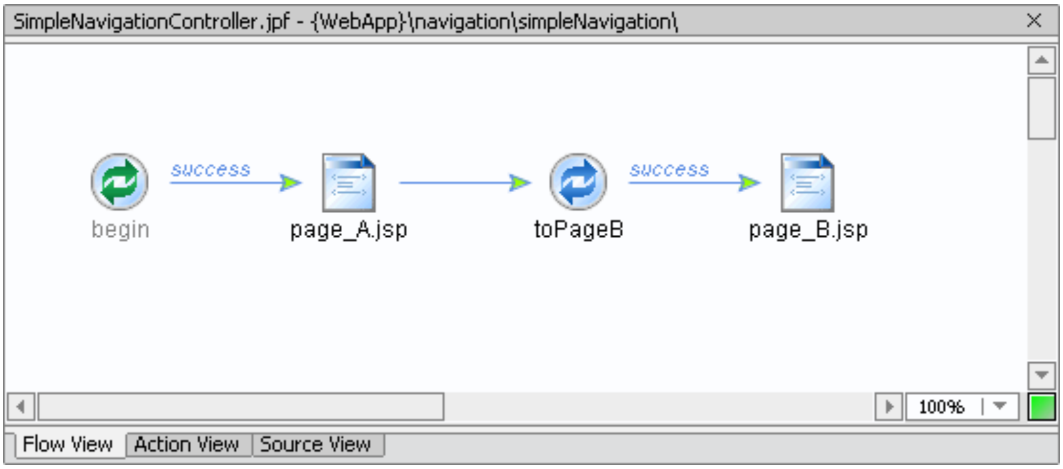
The <netui> tag library contains JSP tags specifically designed to work with the controller class. Tags in the library all begin with the prefixes "netui", "netui−databinding", and "netui−template". Some of these tags perform much like familiar HTML tags, while others perform function particular to page flow web applications. The most important feature of the tag library is its ability to "data bind" to data in the controller file. Data binding allows the JSP pages to both read from and write to Java code in the controller class. This is accomplished without placing any Java code on the JSP pages, greatly enhancing the separation of data presentation and data processing.

Flow, Action, and Source Views

In the WebLogic Workshop IDE, you can switch between the Flow View, Action View, and Source View to create, modify, and view page flow components. Let's start with a simple page flow to understand the basic icons you will encounter in the Flow View, and to find out how the Flow View relates to methods and object in the Source View. In the example, navigation control is forwarded from one JSP to another. You can find this example in WebLogic Workshop at:

```
<BEA_HOME>\weblogic81\samples\workshop\SamplesApp\WebApp\navigation\simpleNaviga
```

Here is the Flow View diagram that we created for this page flow in WebLogic Workshop:

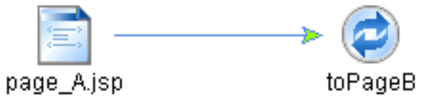


All page flow controller classes have a begin action method to define what happens each time this page flow is first navigated to. For this page flow, page_A.jsp is the first page that the user will see when the page flow's URL is accessed. The begin action is shown with a blue and green circular icon:

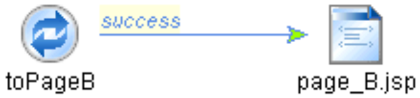
All other actions are represented by a blue circular icon in the Flow View:

Each JSP file that resides in a page flow's directory is shown on the Flow View, and is represented by a rectangular icon with a folded upper−right corner:

An arrow from a JSP page icon to an action icon indicates that the action can be invoked from the JSP page. For example, if there is a link on the JSP that calls the action method, this is depicted in Flow View by the following arrow.

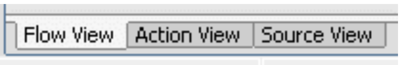


An arrow from an action to a JSP page indicates that the execution of the action will load the target JSP page into the browser. For example:



The name of each action and JSP page is shown below the icon. The name on the arrow next to the action's circular icon corresponds to the logical name of the *Forward* object that is returned by the action method. The Forward object's role will be clearer when we look at the source code below.

In the WebLogic Workshop IDE, use the tabs at the bottom of the main window to switch between the graphical views and source view. When a page flow is open, its graphical representation is displayed in the Flow View window. You can switch to the page flow's graphical Action View or to its code−level Source View:



The Action View allows you to focus on a smaller portion of the page flow, for instance to examine a particular action and its form bean. The Source View is where you can customize the generated code and add business logic, or call controls that implement business logic.

Let's turn to a few examples that demonstrate some of the key features of navigational control and data processing.

Navigation: a Simple Example

As shown in the Flow View diagram, the page flow class defines an action method named toPageB. This action can be invoked by a link on the JSP page page_A.jsp.

page_A.jsp

```
<%@ taglib uri="netui−tags−html.tld" prefix="netui"%>
    ...
    <netui:anchor action="toPageB">Link to page_B.jsp</netui:anchor>
```

A special JSP tag library named netui−tags−html.tld is referenced. WebLogic Workshop provides this tag library and several others to help you develop dynamic web applications. The <netui:anchor...> tag used here is simply invoking an action (toPageB) with a hyperlink. (For more information about the page flow tag library, see Designing User Interfaces in JSPs.)

In the controller file SimpleNavigationController.jpf, the toPageB action method is defined as follows:

SimpleNavigationController.jpf

```
import com.bea.wlw.netui.pageflow.Forward;
```

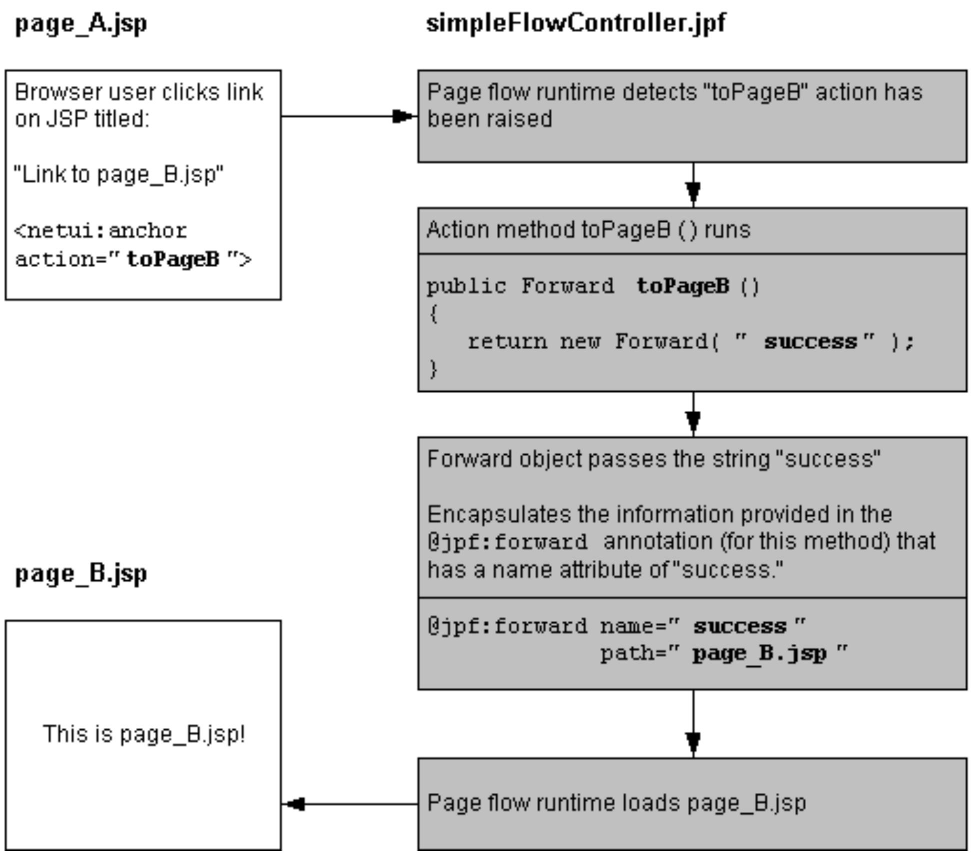
```
        ...

    /**
     * @jpf:action
     * @jpf:forward name="success" path="page_B.jsp"
     */
    public Forward toPageB()
    {
        return new Forward( "success" );
    }
```

When the link on page_A.jsp is clicked, the page flow runtime detects the action and runs the toPageB action method. This action method is coded to return a Forward object which passes the parameter "success". (Notice that this name "success" matches the name on the corresponding action arrow in Flow View.)

Look at the two @jpf annotations that appear on the lines above this action method. These annotations are enclosed in Javadoc comments. The @jpf:action tag indicates that the toPageB method is an action method. The @jpf:forward tag describes the behavior of that method.

Putting it all together, a Forward object is returned by an action method. The Forward object passes the string "success", indicating that it should behave according to the directions encoded in the annotation @jpf:forward name="success". That annotation's path attribute has the value "page_B.jsp", which causes the page flow controller to load page_B.jsp into the browser.

The following diagram summarizes the flow in the example:


page_A.jsp
simpleFlowController.jpf
Browser user clicks link on JSP titled:
"Link to page_B.jsp"
<netui:anchor action=" toPageB ">
Page flow runtime detects "toPageB" action has been raised
Action method toPageB () runs
public Forward toPageB ()
{
return new Forward(" success ");
}
Forward object passes the string "success"
Encapsulates the information provided in the @jpf:forward annotation (for this method) that has a name attribute of "success."
@jpf:forward name=" success "
path=" page_B.jsp "
page_B.jsp
This is page_B.jsp!
Page flow runtime loads page_B.jsp

To change the navigation target of this action method, simply change the value of the path attribute. For example, if you want this action method to navigate to page_C.jsp, you would make the following change to the controller file (no change to the JSP page is necessary).

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="page_C.jsp"
 */
public Forward toPageB()
{
    return new Forward( "success" );
}
```

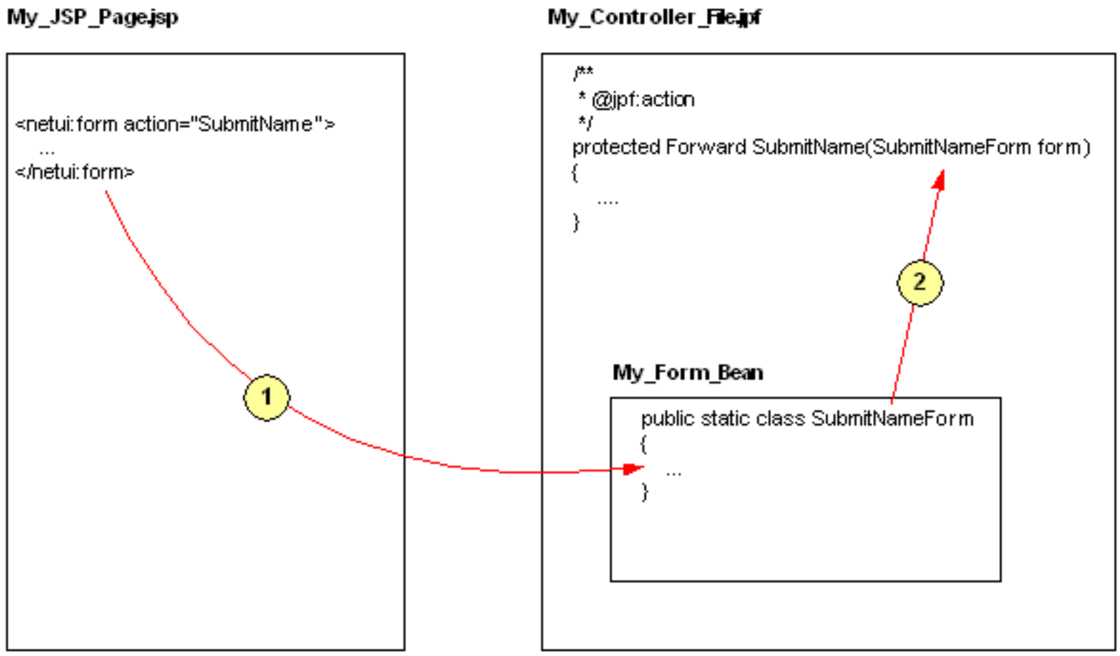
As you will see in later sections, the WebLogic Workshop IDE generates this code for you when you create a new page flow or JSP file from the graphical view. This code generation and subsequent validation of your changes saves you considerable time.

Submitting Data: A Simple Example

Suppose you want to your web application to collect data from users and then process that data in some way. The following example demonstrates how to set up a data submission process using page flows. The sample code referred to in this example can be found at:

`<BEA_HOME>\weblogic81\samples\workshop\SamplesApp\WebApp\handlingData\simpleSub`

Submitting data is a two step process: (1) the data submitted from a JSP page is loaded into a Form Bean instance and (2) the Form Bean instance is passed to an action method for processing.



```
<netui:form action="SubmitName">
   ...
</netui:form>
```

```
/**
 * @jpf:action
 */
protected Forward SubmitName(SubmitNameForm form)
{
    ....
}
```

```
public static class SubmitNameForm
{
    ...
}
```

Form Beans are simple Java classes with fields and setter and getter methods for accessing those fields. Form Beans classes are contained within the controller file. In most cases, Form Beans are designed to accept data submitted from JSP forms. For example, if a JSP page has input elements for name, eye_color, and height, then the Form Bean will have corresponding fields for name, eye_color, and height. The following example

Form Bean can be found in the controller file SimpleSubmitController.jpf. It contains one field, name, and setter and getter methods for that field.

SimpleSubmitController.jpf

```
public class SimpleSubmitController extends PageFlowController
{

...

  public static class SubmitNameForm extends FormData
    {
        private String name;

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

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

The input elements on the JSP page are said to be "data bound" to the fields in the Form Bean. Data binding allows the the data submitted from the JSP page to be loaded into the Form Bean instance. For example, the input element on index.jsp contains a data binding expression that refers to the name field of the Form Bean: {actionForm.name}. The expression "actionForm" refers to the Form Bean SubmitNameForm, the property ".name" refers to the name field of the Form Bean. For detailed information about data binding see Using Data Binding in Page Flows.

index.jsp

```
    <netui:form action="SubmitName">
        Name: <netui:textBox dataSource="{actionForm.name}"/>
        ....
    </netui:form>
```

Finally the Form Bean instance (carrying the submitted data) is passed to the action method for processing.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="showName.jsp"
 */
protected Forward SubmitName(SubmitNameForm form)
{
    //
    // The data is processed here
    //

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

The submitted data can be accessed by calling the getter methods on the Form Bean.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="showName.jsp"
 */
protected Forward SubmitName(SubmitNameForm form)
{
    if( form.getName() != null )
                    // do something here
    else
        // do something else here

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

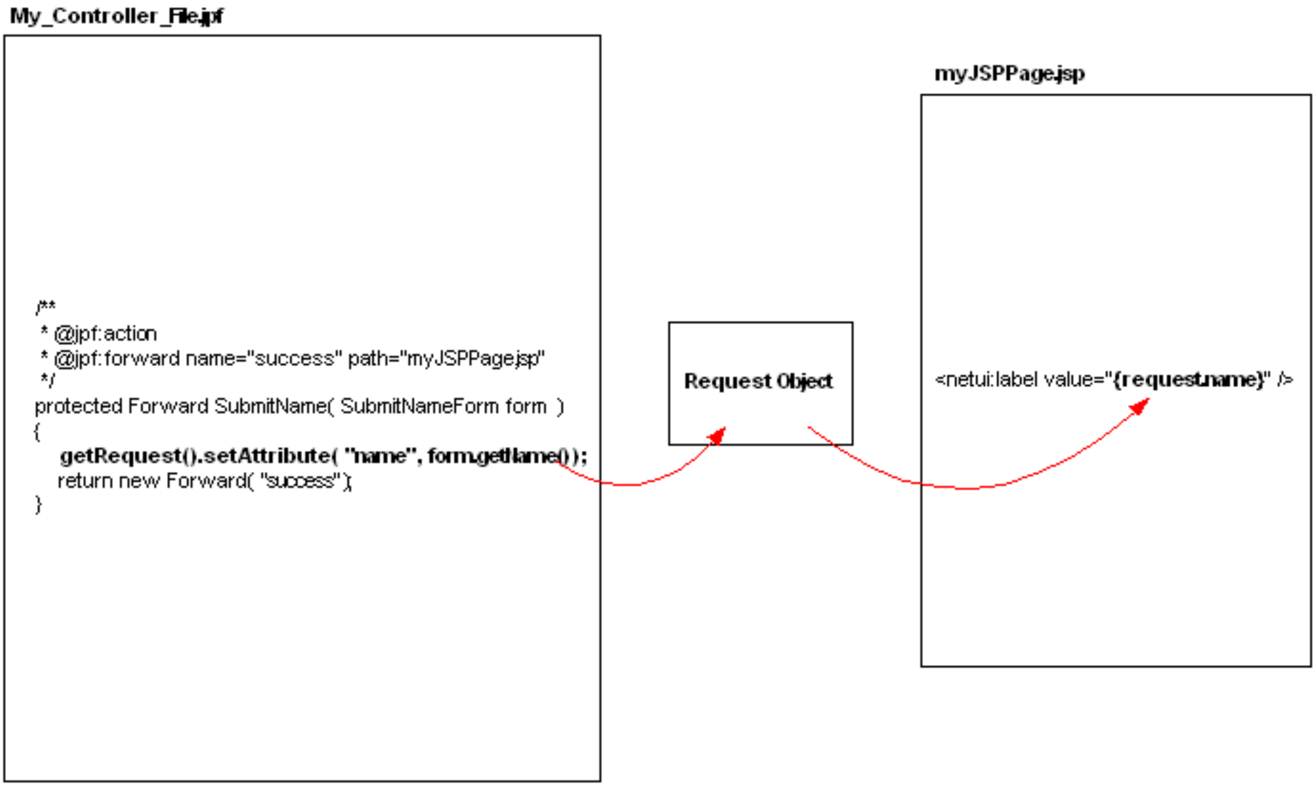
By default the Form Bean instance that is passed to the action method exists only as long as the HTTP request. This is called a "request−scoped Form Bean". When the HTTP request is destroyed, the Form Bean instance, along with the user submitted data, is destroyed. As an alternative, you can use a Page Flow−scoped Form Bean, which has a longer life cycle. For details see Form Bean Scopings.

Displaying Data: A Simple Example

Suppose that once you have collected data, you want to display it back to the user. The following example shows how to use data binding to display data to the user. The sample code referred to can be found at:

`<BEA_HOME>\weblogic81\samples\workshop\SamplesApp\WebApp\handlingData\simpleSub`

Displaying data using data binding requires that (1) the data is located somewhere where it can accessed by the JSP page and (2) the JSP page uses a data binding expression to retrieve the data from that location.



```
/**
 * @jpf:action
 * @jpf:forward name="success" path="myJSPPage.jsp"
 */
protected Forward SubmitName( SubmitNameForm form )
{
   getRequest().setAttribute( "name", form.getName());
   return new Forward( "success");
}
```

```
<netui:label value="{request.name}" />
```

Notice the syntax of data binding expression on the JSP page. (1) It is framed by curley braces, (2) it begins with a data binding context, in this case the request context, and (3) the context is followed by an attribute, in this case "name".

In the following example, an action method places data on the name attribute of the request object.

SimpleSubmitController.jpf

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="showName.jsp"
 */
protected Forward SubmitName(SubmitNameForm form)
{
    getRequest().setAttribute("name", form.getName());
    return new Forward("success");
}
```

After the data has been located on the name attribute of the request object, it is displayed on a JSP page using a data binding expression.

showName.jsp

```
Here is the data you submitted: <netui:label value="{request.name}" />
```

Note that the request object has a relatively short life−cycle. When the user makes a new request, by navigating to a new JSP page or invoking another action method, the current request object is destroyed along with the data it contains. If your application requires the data to be more persistent, then you could use a different data binding context, for example the session object or a Page Flow−scoped Form Bean, which both have longer life−cycles. For detailed information about the different data binding contexts available, see Using Data Binding in Page Flows.

Related Topics

Getting Started: Web Applications

Tutorial: Page Flow

A Detailed Page Flow Example

Advantages of Using Page Flows

This topic outlines the advantages of using page flows in your web applications. You may be familiar already with the Struts framework, which is a part of the Jakarta™ Project by the Apache Software Foundation™. Struts is an open−source framework for building Web applications based on the model−view−controller (MVC) design paradigm. WebLogic Workshop page flows extend the Struts framework to provide a simplified development model with numerous additional features:

- ***Ease of use***
  Typically, native Struts development requires management and synchronization of multiple files for each Action, form bean, and the Struts configuration file. Even in the presence of tools that help edit these files, developers are still exposed to all the underlying plumbing, objects, and configuration details. Page flows provide a dramatically simpler, single−file programming model that allows you to focus on the code you care about, see a visual representation of the overall application flow, and easily navigate between pages, actions, and form beans. Page flows also provide a number of wizards to automate common tasks and visualize tag libraries. Furthermore, page flows provide key programming model benefits like thread safety. As a developer, this means that you can be insulated from some of the complexities that are pervasive in other web development models today like Struts and servlets. The result is that page flows enable you to become immediately productive in building sophisticated web applications, without the learning curve typically associated with Struts.
- ***Nested page flows***

  Struts provides a useful framework for smaller applications, but has trouble scaling to larger applications. Every page and action is referenced from a single configuration file, and the manageability of applications and projects quickly degrades as the size of the application increases. But with the nesting feature of page flows, you can easily create smaller nested applications that are linked together, helping to enforce a modular design paradigm and supporting larger team development projects. Importantly, nested page flows automatically manage session state as a user moves between them, ensuring that the minimum possible level of system resources is used.

- ***State management***

  Page flows make state management easy. You can put data in session state simply by using a variable in your JPF controller class, unlike with Struts action classes. Or you can easily pass data (as Java beans) to individual pages. Data stored in the session state is automatically freed as a user leaves the page flow for efficient use of session data.

- ***Rich Data binding features***

  The WebLogic Workshop data binding tags and wizards make it easy to create rich, dynamic pages within a page flow. These features enable binding of user interface (UI) components to numerous data contexts (not just limited to form beans) and the data may come from any source: a database, a web service, an EJB, a custom application, and so on. In the IDE, you can drag−and−drop the data binding tags to your JSP page and bind to data, including complex types such as method invocations, repeating data, and grids. For more information, see Using Data Binding in Page Flows and Presenting Complex Data Sets in JSPs.

- ***Service−oriented design with Java controls***

Page flows have full support for Java controls, a simple programming model abstraction for accessing enterprise resources and packaging business logic. Java controls give developers access to the powerful features of the J2EE platform (security, transactions, asynchrony) in a simple, graphical environment with methods, events and properties, and enable developers to build Web applications conforming to best practices for service−oriented architecture. For more information, see Tutorial: Java Control.

- ***Integration with Portal***

  Have a page flow? You can easily also have a portlet! Page flow applications can be instantly turned into portlets via a wizard in WebLogic Workshop. For more information, see How Do I: Create a Portlet? and How Do I: Add Portal Functionality to an Existing Page Flow Application?
- ***Strong data typing***

  Struts treats all data as a String, making it difficult to work with rich form data. Page Flows, on the other hand, allow developers much easier access to strongly−typed data, making it a lot easier to work with form information.
- ***Powerful exception handling***

  You can handle exceptions by pointing to local actions in your page flows, Page Flows, and handle not−logged−in errors globally across a set of page flows. This makes it much easier to manage errors and centralized login processes for your entire application versus managing a cloud of exception handler classes in Struts. WebLogic Workshop provides a number of exception types that represent common exception scenarios for page flow applications, such as `ActionNotFoundException`, `LoginExpiredException`, and `UnfulfilledRolesException`. For more information, see Handling Exceptions in Page Flows.
- ***Iterative development experience***

  With the WebLogic Workshop IDE and page flows, you do not need to go through the tedious development cycle of edit, build, deploy, and test. Simply make a change in the IDE and click the Run button. The WebLogic Workshop model for automated deployment and integrated testing displays the results of your code change immediately. Plus, more errors are caught up−front through the WebLogic Workshop page flow compiler, instead of discovering the errors at runtime through exceptions or unintended behavior.
- ***Built on the open−source Struts framework***

  While page flows provide a number of compelling advantages relative to Struts, keep in mind that page flows are based on Struts. This means that page flows can interoperate with existing Struts applications. You can always use the Struts Merge feature of page flows to obtain full access to underlying Struts artifacts and their configuration details. And page flows can interoperate with existing Struts applications, with both residing in the same web project. Also, with the page flow portability kit, you can run a page flow application on Apache Tomcat" and any J2EE−compliant Servlet/JSP 2.3 container. For more information about this kit, see the BEA dev2dev.bea.com web site.

Related Topics

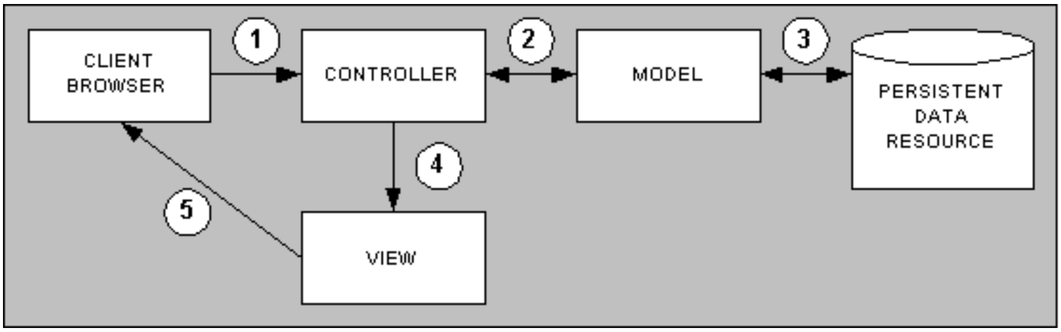
Using Form Beans to Encapsulate Data

A Detailed Page Flow Example

Page Flow and JSP Reference

Using Form Beans to Encapsulate Data

The concepts behind page flows − separation of business logic and presentation, and centralized navigational control − reflect a design pattern that may be familiar to you: Model−View−Controller (MVC). The following diagram shows a typical sequence of events involving the various MVC components:



1. The client browser issues an HTTP request to the web application.
2. The Controller component receives the request. The Controller makes decisions on the processing that will occur next, based on its business rules. In the case of WebLogic Workshop page flows, the Controller is the page flow's JPF file. In the source code, simple tags that appear in Javadoc comments are used to annotate the purpose of the Controller's methods.
3. The Model components interact with the persistent data resources, such as a relational database. In WebLogic Workshop, the Model components can for instance be implemented as controls.
4. The Controller decides which View component should be used, based on the results of processing and the data returned by the Model components.
5. The View component (the selected JSP file) renders the HTTP response, which the server displays to the client browser.

The separation of JSP (view) presentation code from navigation decisions and business logic is an important principle of page flows. Separating the presentation of data on a JSP page from the processing of the data by the controller allows for the encapsulation of the data itself and its handling. For example, to handle customer registration you can create a separate Java class − embedded in the controller or in a separate Java class or custom Java Control− that contains the customer's registration data, such as name, address, login name and password, as well as the business logic necessary to register a new custom, change an existing registration, or remove the customer account. Updates to the business logic can be made in one class instead of in multiple web pages throughout your web application.

Data binding is the process of associating the presentation of the data on a JSP page with the data itself. There are various data binding contexts in WebLogic Workshop, which are described in Using Data Binding Tags in Page Flows. A very common type of data binding is done using forms and form beans, which is described in the following example. The various presentation elements that can be used in the design of a JSP page, such as textboxes, radiobuttons and checkboxes, are discussed in Designing User Interfaces in JSPs.

A Page Flow Action that Uses Form Beans

A typical way of eliciting information from a user is through a form. The fields in the form can be empty when the form is first presented to the user, for instance when entering new information, or can be prepopulated, for example when a user modifies information that he/she entered on an earlier occasion. In either case a form bean is used to encapsulate the data.

A form bean class is typically defined as an inner class within a page flow class, and in most cases extend com.bea.wlw.netui.pageflow.FormData. When you define a new action method with a form bean in the page flow's Flow View, the relevant import statement is automatically included:

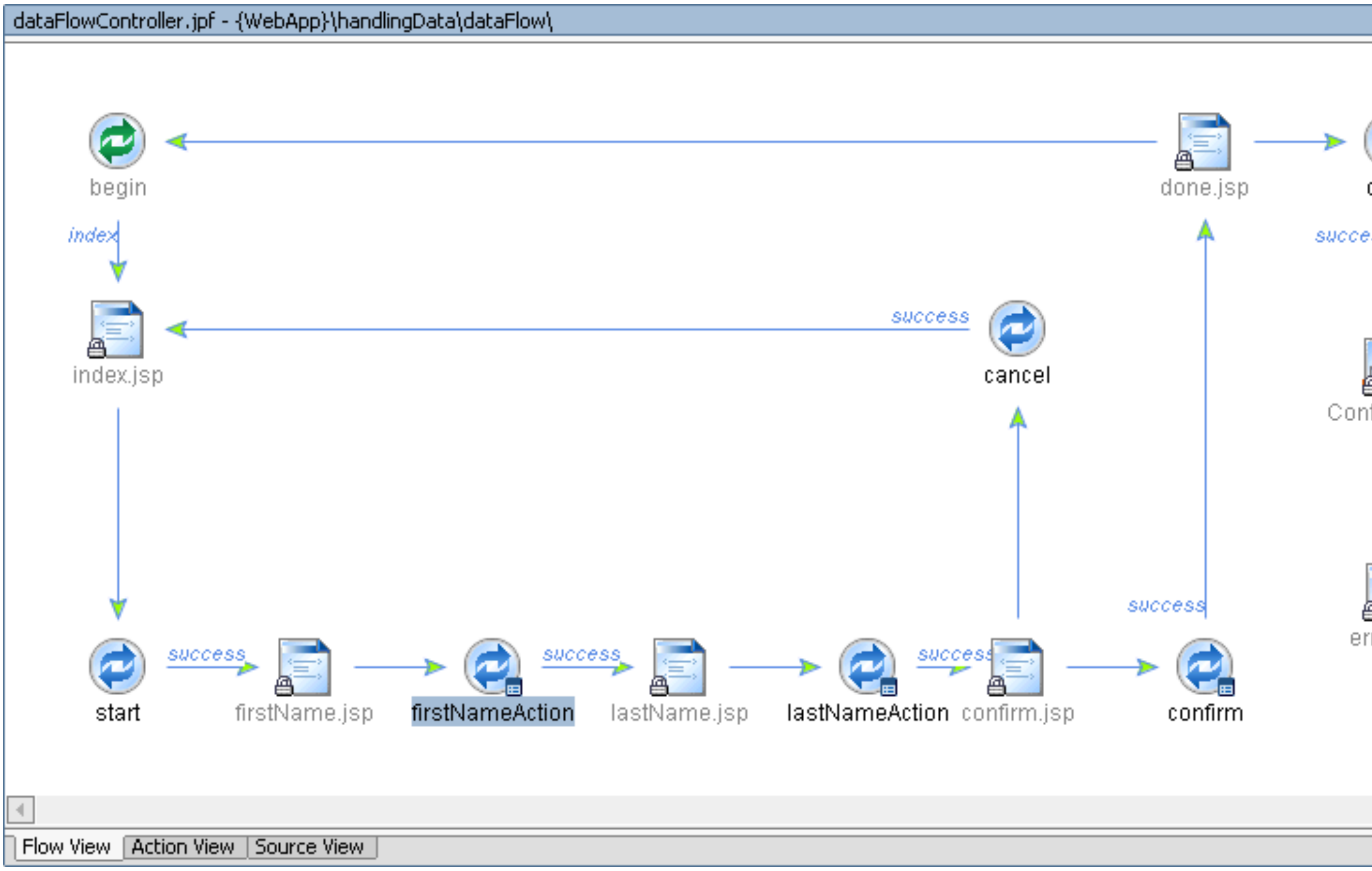
```
import com.bea.wlw.netui.pageflow.FormData;
import com.bea.wlw.netui.pageflow.Forward;
import com.bea.wlw.netui.pageflow.PageFlowController;
import javax.servlet.http.HttpServletRequest;
```

In the page flow, an action method is used to receive and process a form bean from one JSP to the next. In the controller's Flow View, an action method that uses a form bean displays a slightly different icon compared to an action method without a form bean argument:

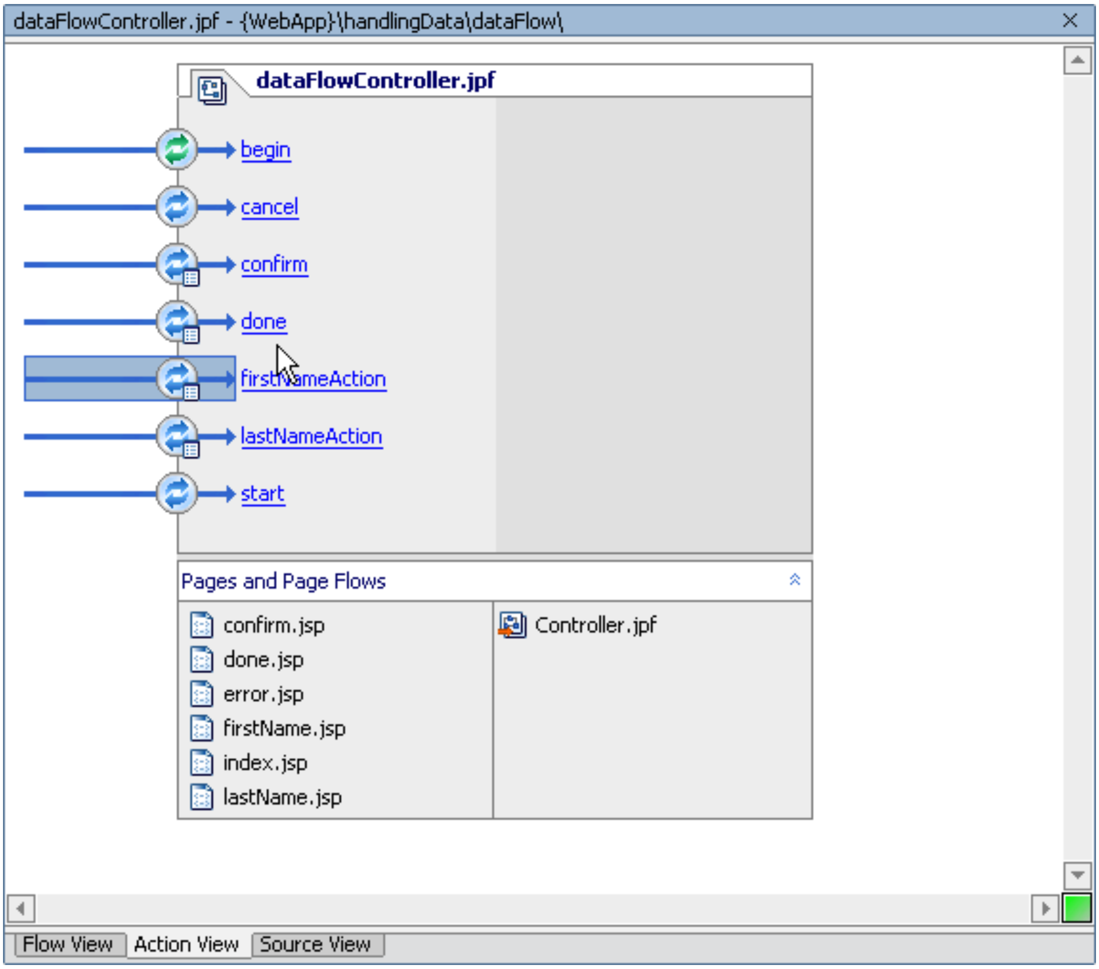
Let's look at an example of a page flow that does simple navigation and uses a form bean. The page flow's purpose is to pass data between JSP pages that participate in this page flow. The example, named "dataFlowController", defines and uses a form bean named NameActionForm within the dataFlowController.jpf class. To run this example and examine the code, go to:

```
<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\handlingData\dataFlow\
```

The complete Flow View for dataFlowController.jpf is given here. To demonstrate the use of form beans, we'll focus on the ***firstNameAction*** action method:

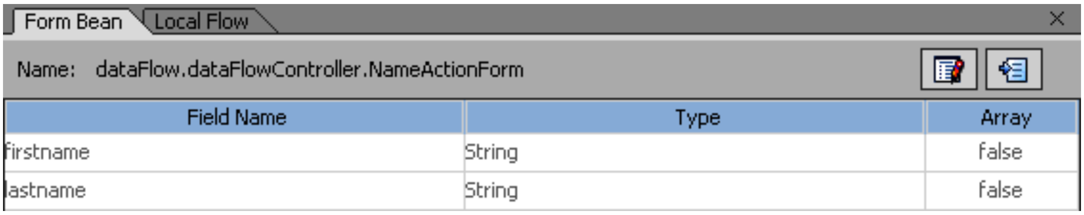


Let's switch to Action View to examine the local flow of the firstNameAction method. After clicking the Action View tab near the bottom of the window, the IDE initially displays all the actions defined in the page flow, and shows a message that "No action is currently selected." The following diagram shows a portion of the display, after firstNameAction has been selected:



The Local Flow indicates that the firstName.jsp file raises an action named firstNameAction. This action method is defined in the page flow controller file dataFlowController.jpf. When the firstNameAction action method runs, its Forward object returns a logical name of "success", and the result is that the lastName.jsp page is loaded.

Now click the Form Bean tab to examine the form bean used by this firstNameAction method:



Name: dataFlow.dataFlowController.NameActionForm

Field Name	Type	Array
firstname	String	false
lastname	String	false

The Form Bean details section shows that the firstNameAction method uses a form called NameActionForm, which contains two String variables, lastname and firstname. There are two icons to the right of the form bean name. When you click the left−most icon you can add and remove fields using the form bean editor, while clicking the other icon will take you to the source view of the form bean. Let's do the latter by clicking this icon:

This is the source code for NameActionForm form bean:

```
public static class NameActionForm extends com.bea.wlw.netui.pageflow.FormData
{
    private java.lang.String firstname;
    private java.lang.String lastname;

    public void setLastname(java.lang.String lastname)
    { this.lastname = lastname; }

    public java.lang.String getLastname()
    { return this.lastname; }

    public void setFirstname(java.lang.String firstname)
    { this.firstname = firstname; }

    public java.lang.String getFirstname()
    { return this.firstname; }
}
```

Now let's turn to the source code of the firstNameAction method:

```
/**
 * This method is passed the page−flow−scoped Form Bean pageFlowScopedBean.
 *
 * If the annotation
 *
 *    jpf:action form="pageFlowScopedBean"
 *
 * were not present, this method would be passed a new instance of the Form Bean
 * and that instance would be request−scoped, not page−flow−scoped.
 *
 * @jpf:action form="pageFlowScopedBean"
 * @jpf:forward name="success" path="lastName.jsp"
 */
public Forward firstNameAction(NameActionForm form)
{
    return new Forward( "success" );
}
```

As shown conceptually on the Flow View and Action View drawings, when the firstNameAction action method runs, its Forward object returns a name of "success", matching the @jpf:forward annotation's logical name, and the result is that the lastName.jsp page is loaded next.

Note also that the firstNameAction method is passed a Page Flow−scoped Form Bean instance. Page Flow−scoped instances of Form Beans have a life−cycle that is as long as the Page Flow instance. Because they persist as long as the Page Flow instance, they are a good place to store data accumulated by the wizard.

Finally, the actual databinding between a form element and a field in the form bean is done using the reserved keyword actionForm on the form element. You can reference fields in the form bean using the 'dot' notation. You must always enclose the entire expression in curly braces, like is shown in this example taken from firstName.jsp:

```
<netui:form action="firstNameAction">
     ...
     <netui:textBox dataSource="{pageFlow.pageFlowScopedBean.firstname}" />
     ...
```

```
</netui:form>
```

The textbox for the first name that is part of the form is tied to the variable firstname in the form bean. Notice that when the form is submitted, the action method firstNameAction is called. Databinding is discussed in detail in Using Data Binding Tags in Page Flows. To learn how to check the information a user entered in a form, see Validating User Input.

Related Topics

Designing User Interfaces in JSPs

Presenting Complex Data Sets in JSPs

A Detailed Page Flow Example

Page Flow and JSP Sample Applications

Form Bean Scopings

This topic explains the different kinds of scoping that Form Beans can possess in a Controller file. It also describes the different data−accumulation behaviors inherent in each sort of scoping.

There are two different scopings available for Form Beans: (1) request−scoping and (2) page flow−scoping.

Request−scoped Form Beans have the same life cycle as a request to the page flow servlet. Each time a new request is made of the servlet, a new Form Bean instance is created. When the request is destroyed, the Form Bean instance is destroyed as well.

Page Flow−scoped Form Beans have the same life cycle as the Page Flow's Controller file instance. Data stored in a Page Flow−scoped Form Bean will be maintaned until the Controller file instance is destroyed.

Request−Scoped Form Beans

By default, Form Bean instances that are passed to action methods are request−scoped instances.

```
    /**
     * @jpf:action
     * @jpf:forward name="success" path="displayData.jsp"
     */
    protected Forward submit( MyFormBean requestScopedBean )
    {
        return new Forward( "success" );
    }
```

"Request−scoped" means that the Form Bean instance has the same life−cycle as the HTTP request. The Form Bean instance is created at the same time as the request and it is destroyed, along with any data within it, when the request is destroyed. Request−scoped Form Bean instances are useful when you submitting data from a single JSP page to a single action method. But they are cumbersome when your application needs to accumulate data over many different JSP pages and action methods. This is because accumulating data from many different JSP pages entails the creation and destruction of many different request objects, along with the creation and destruction of many different request−scoped Form Bean instances. To preserve the accumulated data, you will need to pass the data from the previous Form Bean instance into the new Form Bean instance at each stage in the process. As an alternative to a string of request−scoped Form Beans, you could use a single Page Flow−scoped Form Bean instance to hold the accumulated data.

Page Flow−Scoped Form Beans

Page Flow−scoped Form Bean instances have the same life−cycle as the Controller file instance. They are created and destroyed when the Controller file instance is created and destroyed. This makes Page Flow−scoped Form Beans useful for storing data that has been accumulated across many different JSP pages.

To create a Page Flow−scoped Form Bean instance, construct a public member variable of the Form Bean in the Controller file.

```
public class myController extends PageFlowController
{
    public MyFormBean pageFlowScopedBean = new MyFormBean();
    .
    .
    .
}
```

Once you have created a Page Flow−scoped instance of a Form Bean, you can pass the instance to action methods by using the @action form="*form_bean*" annotation.

```
public class myController extends PageFlowController
{
    public MyFormBean pageFlowScopedBean = new MyFormBean();

    /**
     * @jpf:action form="pageFlowScopedBean"
     * @jpf:forward name="success" path="displayData.jsp"
     */
    protected Forward submit( MyFormBean form )
    {
        return new Forward( "success" );
    }
}
```

Each time the submit() method is invoked, it is passed the same instance of the Form Bean, namely, pageFlowScopedBean, the instance that was created when the Controller file instance was created.

Interactions Between Request−Scoped and Page Flow−Scoped Form Bean Instances

The following section details what happens when one action method passes Form Bean data to another action method. In the following list, assume that A is an action method that invokes action method B.

- If action A takes a Page Flow−scoped Form Bean and action B does not, B receives a new request−scoped instance of the Form Bean.

  ```
  /**
   * @jpf:action form="pageFlowScopedBean"
   * @jpf:forward name="success" path="B.do"
   */
  protected Forward A(MyFormBean form)
  {
      return new Forward("success");
  }

  /**
   * @jpf:action
   */
  protected Forward B(MyFormBean form)
  {
      return new Forward("success");
  }
  ```
- If A takes a Page Flow−scoped Form Bean and B does not, ***and*** if A explicitly passes its Form Bean on the Forward to B, B will use A's Page Flow−scoped instance of the Form Bean.

  ```
  /**
   * @jpf:action form="pageFlowScopedBean"
   * @jpf:forward name="success" path="B.do"
   */
  ```

```
protected Forward A(MyFormBean form)
{
    return new Forward("success", form);
}

/**
 * @jpf:action
 */
protected Forward B(MyFormBean form)
{
    return new Forward("success");
}
```

- If A takes a request−scoped Form Bean and B takes a Page Flow−scoped Form Bean, B will not use the (request−scoped) Form Bean instance that was created for A, instead B will use the Page Flow−scoped Form Bean.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="B.do"
 */
protected Forward A(MyFormBean form)
{
    return new Forward("success");
}

/**
 * @jpf:action form="pageFlowScopedBean"
 */
protected Forward B(MyFormBean form)
{
    return new Forward("success");
}
```

- If A takes a request−scoped Form Bean and B takes a Page Flow−scoped Form Bean, ***and*** if A explicitly passes its Form Bean instance on the Forward to B, B will not only use A's (request−scoped) Form Bean instance, *but the Page Flow−scoped Form Bean will be re−set to carry the same data as the instance that A passed*.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="B.do"
 */
protected Forward A(MyFormBean form)
{
    return new Forward("success", form);
}

/**
 * @jpf:action form="pageFlowScopedBean"
 */
protected Forward B(MyFormBean form)
{
    return new Forward("success");
}
```

- If A takes a Page Flow−scoped Form Bean and B takes a Page Flow−scoped Form Bean, then B will use A's Form Bean instance.

```
/**
 * @jpf:action form="pageFlowScopedBean"
 * @jpf:forward name="success" path="B.do"
 */
protected Forward A(MyFormBean form)
{
    return new Forward("success");
}

/**
 * @jpf:action form="pageFlowScopedBean"
 */
protected Forward B(MyFormBean form)
{
    return new Forward("success");
}
```

Related Topics

@jpf:action Annotation

@jpf:forward Annotation

Using a Wizard to Collect User Infomation Sample

Using Data Binding in Page Flows

Data binding is the process that ties data to presentation tags in JSPs, thereby creating dynamic web applications that are easy to build and maintain. Various JSP tags can tie data from various databinding contexts. The topic Using Form Beans to Encapsulate Data showed an example of a databinding context, called actionForm, which ties form fields to variables of a form bean. The current topic discusses the various databinding contexts, how to add data to a context, and how to retrieve data from a context.

Data Binding Contexts

To tie data to a JSP using a JSP tag, you refer to a data binding context name, as in the following example:

```
<netui:form action="ChangeZipCode">
     <netui:textBox dataSource="{actionForm.postalCode}"/>
     <netui:button>Submit</netui:button>
<netui:form>
```

In this example, the dataSource attribute on the netui:textBox tag references the variable postalCode in a form bean. Form beans are associated using the keyword actionForm. All data binding context elements can be accessed using the 'dot' notation, and the entire expression must always be enclosed in braces. For more information on accessing these elements, see Accessing Data Binding Context Properties below.

Most NetUI tags can reference a data binding context. These include all ***NetUI Data Binding*** tags and most tags in the ***NetUI*** library, in particular form field tags and other tags with a value or defaultValue attribute. For more information on the ***NetUI*** library, see Designing User Interfaces in JSPs. For more information on ***NetUI Data Binding*** tags, see Presenting Complex Data Sets in JSPs. For a detailed description of a tag and its attributes, see JSP Tags Reference, or in Source View, place your cursor inside the tag and press F1.

An overview of the various data binding contexts is given in the following table. The databinding contexts are discussed in more detail below, with the exception of application. For information on this context, consult your favorite JSP documentation. A summary of the lifetime and scalability of these contexts is described in separate section called Lifetime and Scalability of Data Binding Contexts:

Context Name	***Object the Context References***	***Comments***
`pageFlow`	The current page flow.	Properties that are defined in the current page flow controller class can be referenced. Values are read−write.
`globalApp`	The Global App object for the web application.	Properties that are defined in Global.app, located in the web project's WEB−INF/src/global folder, can be referenced anywhere in the web application. Values are read−write.
`actionForm`	The form bean used in an action referenced by a netui:form tag in a JSP.	Properties that are defined in the form bean can be referenced. Values are read−write.
`request`	The request that triggered the loading of the JSP.	Attributes that are defined as part of a request inside an action method can be

		referenced in the JSP called by the action method. Values are read only.
`url`	Query parameters on the current JSP's URL.	Query parameters that are part of the URL can be referenced. Values are read only.
`session`	Attributes in the JSP's session context.	The session refers to the JSP user session object (javax.servlet.http.HttpSession) representing an individual's user session. Values are read only.
`container`	An item in a complex data set.	Refers to data referenced by many complex data binding tags, such as repeater tags. Values are read only.
`pageContext`	The JSP's PageContext object.	The pageContext references the PageContext's page−scope attribute map of a JSP. Values are read only. In WebLogic Workshop the pageContext is also used to access actionForm, container, globalApp and pageFlow data in scriptlet when it is placed there using the netui−data:getData tag.
`bundle`	Properties that are defined in a message resources file.	The bundle context references properties that you define in a message resources file, which allows you to implement internationalized web applications by ***not hard−coding*** text labels in your JSP pages. Values are read only.
`pageInput`	Member variables defined in the page flow controller class.	When you have declared page inputs by using a <netui−data:declarePageInput> tag in your JSP, you can use the pageInput context to display the value of member variables that were defined in the page flow controller class. Values are read only.
`application`	The JSP's application context.	The application refers to the JSP application object (javax.servlet.ServletContext) representing the application to which the JSP belongs. Values are read only.

pageFlow

When you define a variable in your controller class, you can access this variable from any JSP page that is part of that page flow. For example, if the JPF file contains this code:

```
public class SimpleflowController extends PageFlowController
{
   public String labelName = "This is a label"
   ...
```

You can access this variable on a JSP, for instance by using a netui:label:

```
<netui:label value="{pageFlow.labelName}" />
```

The variable is read−write and can be changed, for instance in the controller's action method as is shown in the next example:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="nextPage.jsp"
*/
public Forward labelAction(NameActionForm form)
{
   ...
   labelName = "Default Label";
   return new Forward( "success");
}
```

It can also be changed in a JSP by using a form, as is shown next:

```
  <netui:form action="submitAction">
      <netui:textBox dataSource="{pageFlow.labelName}"/>
      ...
      <netui:button value="submit"/>
  </netui:form>
```

In this example, the text box will display the current value of the pageFlow's labelName variable when the form is first loaded, and upon submit the variable will hold the new value the user entered.

Accessing Page Flow Properties

In the example used above, the page flow property was defined as a public member variable in the page flow controller class. Alternatively you can define a private variable and provide access with a getter and setter method, as is shown in the following example:

```
  private String firstName;

  public String getFirstName() {
      return firstName;
  }

  public void setFirstName(String aStr) {
      firstName = aStr;
  }
```

You can access this variable on a JSP exactly like before. For instance, you can bind the value to a netui:label as is shown next:

```
<netui:label value="{pageFlow.firstName}" />
```

In the example the method getFirstName is used to provide read access to the variable. For more information on getter and setter methods, see Naming Conventions for Form Beans below.

You can also define and access variables that are part of an object created in the controller class. You can provide access by using getter and setter methods, or by referencing the object and property name. The latter option is shown next. For instance, if the JPF file contains the code:

```
public class dataFlowController extends PageFlowController
{
   ...
   public Names myNames = new Names();
   ...
   public class Names implements Serializable {
      public String firstName;
      public String lastName;
   }
   ...
```

You can access the firstName property in a JSP like this:

```
<netui:label value="{pageFlow.myNames.firstName}" />
```

The comments in this section regarding access of page flow properties also apply to properties defined in globalApp, which is discussed next. For more information on accessing properties, see Accessing Data Binding Context Properties.

globalApp

If you need an attribute to be available across all the page flows in a web application's user (browser) session, such as a user's unique session key, you can use the Global class defined in the file Global.app. This file is located in WEB−INF/src/global. In addition to session−wide attributes, you can also use the Global class to define fallback action methods and exception handlers. An instance of this class is included by default in the JPF file, but the reference is commented out; you must uncomment the following line of code in your controller class if you want to use the Global.app instance from within a page flow:

```
protected global.Global globalApp
```

For instance, if you define a variable in the Global class:

```
public class Global extends GlobalApp
{

   public String defaultText = "Please Define";
```

And your controller file references the Global class as described above, you can use it in a JSP file:

```
<netui:textBox dataSource="{actionForm.firstName}" defaultValue="{globalApp.defaultText}"/>
```

Notice that the netui:textBox tag in the example binds to a form bean to post data and binds to globalApp's defaultText to receive its default value.

You can also write to globalApp from a JPF:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="nextPage.jsp"
*/
public Forward writeAction(Form form)
{
    ...
    globalApp.setDefaultText("Altered");
    return new Forward( "success");
}
```

actionForm

To bind to data in a form bean, you use the actionForm context in the JSP's form fields to bind to the form bean properties. You can present a new form to a user and bind to empty variables in a form bean (for an example, see Using Form Beans to Encapsulate Data), or you can prepopulate the form fields, for instance, if you want to update a record stored in a database, as is shown in the next example:

```
/**
* @jpf:action
* @jpf:forward name="update" path="updateItems.jsp"
* @jpf:catch method="sqlExceptionHandler" type="SQLException"
*/
public Forward updateItems(DatabaseForm form) throws Exception
{
    ...
    form.applyValuesToForm(getCurrentRow());
    return new Forward("update");
}
```

This action method reads a record from a database using a built−in DataBase control (this code is omitted in the example), and updates the fields in the form bean with the record before calling the JSP updateItems.jsp. This JSP will display a form field containing the values read from the record:

```
<netui:form action="submitUpdate">
    <netui:content value="{actionForm.itemnumber}"/>
    <netui:textBox dataSource="{actionForm.itemname}"/>
    ...
</netui:form>
```

In the action method called by the form, you can then write these values back to the database:

```
/**
 * @jpf:action
 * @jpf:forward name="updated" path="getItems.do"
 * @jpf:catch method="sqlExceptionHandler" type="SQLException"
*/
public Forward submitUpdate(DatabaseForm aDatabaseForm) throws Exception
{
    getSortFilterService().reset();
    RowSet currentRow = getCurrentRow();
    aDatabaseForm.applyUpdateValuesToRowSet(currentRow);
    javax.sql.RowSet var = myControl.updateItems(currentRow) ;
    return new Forward("updated");
}
```

The action attribute on the netui:form tag determines which action method is called when the user submits the form. The action method in turn determines which form bean is used to store the form fields' values upon form submission. In other words, the action method called by the form determines which form bean is used in the actionForm context. In the example, the action method submitUpdate is called in the action attribute of the netui:form tag. This action method has the form bean DatabaseForm as its argument, so it is this form bean that is used to store the values that the users entered.

When you have a form with prepopulated fields, as is the case in this example, you need to ensure that both the action method loading the JSP and the action method called upon form submission use the same form bean. The actionForm is first used to bind data to the form fields before loading the JSP, and the actionForm is subsequently used to post the user data to a new instance of the form bean. If you want to use one form bean to prepopulate form fields and use another form bean to store the data when the user submits the form, you must use a combination of the actionForm and request data binding contexts. For more information, see request below.

In the above example, the updateItems action method was passed an instance of the form bean DatabaseForm in its argument, presumably filled with data the user entered in a form that called the action method (this JSP code is not actually shown above). The action method updated this instance with a record from a database before passing the form bean object to the JSP updateItems.jsp. Alternatively, it is possible to create a new form bean inside the action method, assign values to its variables, and associate these variables with form fields on the next JSP. This is shown in the next example, which builds on the above example:

```
/**
 * @jpf:action
 * @jpf:forward name="update" path="updateItems.jsp"
 * @jpf:catch method="sqlExceptionHandler" type="SQLException"
 */
public Forward updateItems() throws Exception
{
    DatabaseForm form = new DatabaseForm();
    ...
    form.applyValuesToForm(getCurrentRow());
    return new Forward("update", form);
}
```

Notice that in this case, you need to supply the form as the second argument on the Forward object. The form fields in the JSP that is subsequently loaded bind to the form bean properties in the same manner as shown above through actionForm. If you only want to display the values in the next JSP, that is, you don't need the user to change the property values of the form bean, use request instead (see below).

In the above example, the form that is passed as the argument on the Forward object will be used to prefill the form fields. Similarly, you can have an action method that is passed one form bean in its argument, but loads data in a new form bean instance which is used in the next JSP, as is shown in the following example:

```
/**
 * @jpf:action
 * @jpf:forward name="update" path="updateItems.jsp"
 * @jpf:catch method="sqlExceptionHandler" type="SQLException"
 */
```

```
public Forward updateItems(SomeOtherForm oldForm) throws Exception
{
    DatabaseForm form = new DatabaseForm();
    form.selectiveCopy(oldForm);
    return new Forward("update", form);
}
```

In the example, only the instance of the form bean DatabaseForm is passed to the next JSP. If you want to know how to create a JSP that uses multiple forms and form beans, see How Do I: Use Multiple Forms in a JSP?

Working with Complex Form Beans

In most cases you will likely use form beans with properties that are Strings, primitive data types (such as int, long, boolean, or char), primitive data type wrappers, or arrays of any of these data types. However, in some cases your design might require a form bean that holds an instance of a complex class. When your form bean contains a complex data type, you must add a reset method to your form bean definition to ensure that an instance of the complex class is available to hold the values that the user entered in a form and subsequently submitted (posted).

Note: The reset method is called by the underlying Struts framework used by WebLogic Workshop. You do not need to call this method directly anywhere in the code.

The following example uses a form bean with a Customer property, holding customer data returned by a Java control. The source code for the form bean is shown first. Please note that you must add the reset method to the source code, using the exact signature as shown below:

```
public static class CustomerForm extends FormData
{
    private Customer customer;
    private String couponNumber;

    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }

    public Customer getCustomer()
    {
        return this.customer;
    }

    public void setCouponNumber(java.lang.String couponNumber)
    {
        this.couponNumber = couponNumber;
    }

    public java.lang.String getCouponNumber()
    {
        return this.couponNumber;
    }

    public void reset(org.apache.struts.action.ActionMapping mapping, javax.servlet.http.HttpSe
    {
        this.customer = new Customer();
    }
}
```

Notice that the reset method creates a new instance of the Customer class. The couponNumber property is a String and therefore does not have to be mentioned in this method.

Apart from the definition of the reset method, the form bean is used in exactly the same way as a form bean with simple data types only. In the JSP you reference the various properties through the actionForm context as before:

```
<netui:form action="submitAction" focus="">
    ...
    Customer LastName: <netui:textBox dataSource="{actionForm.customer.lastName}"></netui:textB
    ...
    Customer FirstName: <netui:textBox dataSource="{actionForm.customer.firstName}"></netui:tex
    ...
    Coupon Number: <netui:textBox dataSource="{actionForm.couponNumber}"></netui:textBox>
    ...
    <netui:button value="Submit"></netui:button>
</netui:form>
```

And in the action method called by the form upon submit, you handle the data in the form bean as before:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="showDiscount.jsp"
*/
protected Forward submitAction(CustomerForm form)
{
    customerControl.updateCustomer(form.getCustomer());
    ...
    return new Forward("success");
}
```

Naming Conventions for Form Beans

When you create a form bean, it is recommended that you follow standard naming conventions by defining lowercase form bean properties (such as *item*) and the corresponding getter and setters methods (*getItem* and *setItem*). When you use the form bean editor in the controller's Flow View or Action View, the proper getter and setter methods will be automatically created for you, following the rules given in the below table. Notice that the first row in the table shows the recommended approach to naming form bean properties:

property name (form bean variable name)	***getter name***	***example expression***
item	getItem	{actionForm.item}
ITem	getITem	{actionForm.ITem}
_Item	get_Item	{actionForm._Item}
iTem	getiTem	{actionForm.iTem}
itEm	getItEm	{actionForm.itEm}
_ItEm	get_ItEm	{actionForm._ItEm}
iTEm	getiTEm	{actionForm.iTEm}

Note that you cannot create a variable name that starts with an uppercase letter and is followed by a lowercase letter.

If you create the form bean in Source View, you must add the proper getter and setter methods for the various form bean properties to enable read and write access to these properties through the actionForm context, following the rules given in the above table. For getter methods, you must also make sure that:

- The method name starts with *get*
- The method has a *public* method signature
- The method has a non−void return value
- The method has no parameters

For setter methods, you must also make sure that:

- The method name must start with *set*
- The method returns a *void* type
- The method has a single argument of the same type as the corresponding get method.

The follow example shows a form bean property and its getter and setter method:

```
private java.lang.String item;

public void setItem(java.lang.String item)
{
   this.item = item;
}

public java.lang.String getItem()
{
   return this.item;
}
```

request

If you want to call a JSP and bind it to data that you initialized in the calling action method, you can use the request data binding context. This data can either be contained in a form bean or in the request's attribute map.

Using Form Beans in the Request Context

If you want to display (read−only) the values of a form bean that the user entered in the preceding page, you can add the form bean as the second parameter to the Forward method:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="displayOnly.jsp"
*/
public Forward lastNameAction(NameActionForm form)
{
   // process the form...
   ...
   return new Forward( "success", form );
}
```

In the JSP, you reference the form bean's properties by referring to the *class* name, with the class name's uppercase first letter changed to lowercase (assuming you are following standard class naming conventions):

```
<netui:label value="{request.nameActionForm.firstname}" />
```

Note that the netui:label tag is not used as a form field, that is, it is not part of a form. Also notice again that you are not referring to the form bean's instance name but to the class name, with the first letter in lowercase.

You can also create a new form bean instance in the action method and add it to the request context:

```
/**
 * @jpf:action
 * @jpf:forward name="confirm" path="done.jsp"
*/
public Forward confirm()
{
   NameActionForm myForm = new NameActionForm();
   myForm.firstname = "John";
   return new Forward("confirm", myForm);
}
```

And then call the properties of this form in the same manner (the example would display "John"):

```
<netui:content value="{request.nameActionForm.firstname}" />
```

If you want to load a JSP with a form whose fields are prefilled with data from one form bean, and submit the form to another form bean, you need to use a combination of the request and actionForm binding contexts. The following example illustrates how to do this.

This is the code of the action method that adds the form bean to the request context (by passing it to the Forward constructor) and calls the JSP :

```
  /**
   * @jpf:action
   * @jpf:forward name="success" path="lastNameFilledViaRequest.jsp"
   */
  protected Forward prefillWithRequest()
  {
      ParentNameActionForm form1 = new ParentNameActionForm();
      // read parent's last name from database
      ...
      return new Forward("success", form1);
  }
```

In the JSP, the form field binds to the form bean in the request context through its defaultValue attribute and, upon form submission, posts the data the user entered to another form bean in the actionForm context via the dataSource attribute:

```
  <netui:form action="lastNameAction">
      ...
      <netui:textBox dataSource="{actionForm.lastname}" defaultValue="{request.parentNameAction
```

```
    ...
</netui:form>
```

Note that the actionForm context binds to the form bean of the action method lastNameAction. For more information, see actionForm above.

Using the Attribute Map of the Request Context

Instead of using form beans to store data, you can also create an object of any other class and add it to the request context's attribute map. You do this by using the getRequest and setAttribute methods in your action method, as is shown in the following example:

```
/**
 * @jpf:action
 * @jpf:forward name="confirm" path="done.jsp"
*/
public Forward confirm()
{
   getRequest().setAttribute("firstname", new String("John"));
   return new Forward("confirm");
}
```

And you would bind to this attribute in the JSP by entering:

```
<netui:content value="{request.firstname}" />
```

Notice that in the action method, you set an attribute with a name (firstname) and a value (John) in the request object, which you accessed with the getRequest method. In the JSP, you bind to the data by simply referring to the name of the attribute.

The above example used a simple Java type, but you could define an attribute to refer to an instance of any (serializable) class:

```
getRequest().setAttribute("someAttribute", new MyClass(...));
```

And bind to it from the JSP like this:

```
<netui:label value="{request.someAttribute.someMyClassVariable}"/>
```

url

You can bind to query parameters on a URL using the url data binding context. For instance, a JSP might contain the following code snippet:

```
<netui:anchor action="deleteUser">Delete User
   <netui:parameter name="User_Index" value="1"></netui:parameter>
</netui:anchor>
```

When a user clicks the *Delete User* link, the deleteUser action method is invoked, loading a new JSP page and adding the User_Index attribute with value 1 to its URL, which will look something like this: http://localhost:7001/MyProjectWeb/myPageFlow/deleteUser.do?User_Index=1. On the loaded page, you can bind to this data as follows:

```
<netui:content value="{url.User_Index}"></netui:content>
```

Note: The url data binding context only binds to the first value of a query parameter. If there are multiple values for a query parameter (www.mycompany.com?User_Index=1,45,643), it will only read the first one. If you want to forward multiple−valued attributes to a JSP, use request attributes instead.

session

If you want to work with sessions (instead of with pageFlow or globalApp), you can define attributes in an action method and refer to these attributes in the JSP in very much the same way as discussed above for request. For instance, if an action method contains:

```
getSession().setAttribute("myAttributeKey", myObject);
```

Then, an expression in the JSP might be:

```
<netui:content value="{session.myAttributeKey.someProperty}"></netui:content>
```

You can use the <session−timeout> element of the /WEB−INF/web.xml file to set the session timeout value. The <session−timeout> element is part of the <session−config> element, and sets the number of idle minutes after which sessions in this Web application expire. The value set in this element can override the value set in the TimeoutSecs attribute of the <session−descriptor> element in the WebLogic−specific deployment descriptor weblogic.xml. For details, see the <session−config> section of the web.xml Deployment Descriptor Elements topic on the BEA E−docs web site.

container

The container data binding context is used to refer to the current item in complex data binding expression such as an repeater tag. Complex data binding expressions are discussed in detail in Presenting Complex Data Sets in JSPs. In short, a complex data binding expression ties to a complex data set such as a String array:

```
<netui−data:repeater dataSource="{actionForm.stringArray}">
   ...
   <netui:label value="{container.item}"/>
   ...
</netui−data:repeater>
```

For each item in the referenced String array, the item is displayed on its own line using a netui:label tag. In order to bind to the repeater's current item, the container binding context is used within the body of the repeater. The container binding context has the following properties:

Property Name	Description
item	The current item in the data set. This is used in the example.
index	The integer index of the current item in the data set.
container	A reference to the container holding the current item.

dataSource	A reference to the data set that the current container uses.
metadata	A keyword reserved for implementation in a future release.

In most cases you will probably only use the container's *item* or *index*.

pageContext

The pageContext data binding context references the PageContext's page−scope attribute map of a JSP. This map contains name/value pairs which can be accessed using the pageContext.getAttribute(String) and pageContext.setAttribute(String, Object) methods. Within WebLogic Workshop the pageContext can also be used to access actionForm properties, container data, pageFlow variables and globalApp variables in scriptlet. This is accomplished with the NetUI data binding tag netui−data:getData, which places databound values in the pageContext. The following example demonstrates its use with form bean properties:

```
<netui:form action="lastNameAction" focus="lastname">
   ...
   <netui−data:getData resultId="first" value="{actionForm.firstname}"/>
   ...
   <%
      String firstName = (String) pageContext.getAttribute("first");
      System.out.println("First Name = " + firstName);
      ...
   %>
   ...
</netui:form>
```

In the example, the netui−data:getData tag is used to add an attribute to the pageContext object. The name of the attribute is defined in the tag's resultID attribute and its value is defined in its value attribute. You can subsequently access the attribute through the pageContext's getAttribute method.

The following example shows how to use netui−data:getData and pageContext with containers:

```
<netui−data:repeater dataSource="{pageFlow.strArr}">
   ...
   <netui−data:repeaterItem>
      <netui:label value="{container.item}" />
      <netui−data:getData resultId="item" value="{container.item}"/>
      <%
         String currentItem = (String) pageContext.getAttribute("item");
         System.out.println(currentItem);
         ...
      %>
   </netui−data:repeaterItem>
   ...
</netui−data:repeater>
```

The netui−data:getData tag does not only bind data from the actionForm, container, pageFlow, and globalApp contexts to the pageContext object, but can also be used to bind data from the other data binding contexts. However, the other data binding contexts can be accessed in other ways in scriptlet and therefore generally do not have to be added to the pageContext. For more information, see How Do I: Access Data Binding Contexts in Scriptlet and JavaScript?

bundle

The bundle data binding context references properties that you define in a message resources file, which allows you to implement internationalized web applications ***by not hard−coding*** text labels in your JSP pages.

For example, in your page flow controller class, you can add an annotation such as the following:

```
/**
 * @jpf:controller
 * @jpf:message−resources resources="labels.Messages"
 */
 public class Controller extends PageFlowController
 { ...
```

In this example, the naming convention is that the message resources file must be located in the project's /WEB−INF/classes/labels directory, and the file must be named Messages.properties.

In the Messages.properties file, you might have a property such as:

```
nameLabel=Name
```

Then in your JSP page, you can use a data binding expression such as the following:

```
<netui:label value="{bundle.default.nameLabel}"/>
```

Alternately, you can use the key attribute on the @jpf:message−resources annotation:

In your page flow controller class annotation:

```
/**
 * @jpf:controller
 * @jpf:message−resources key="foo" resources="labels.Messages"
 */
 public class Controller extends PageFlowController
 { ...
```

Assume that the Messages.properties file in the /WEB−INF/classes/labels directory is the same as shown in the previous example. In the JSP page, you could use a tag such as the following:

```
<netui:label value="{bundle['foo/jpfDirectory'].nameLabel}"/>
```

In this case, your page flow controller class file is /jpfDirectory/Controller.jpf.

Another option is to use the <netui−data:declareBundle> tag. For example, in your JSP page:

```
<netui-data:declareBundle name="someMessages" bundlePath="com/foobar/resources/WebAppMessages"/
```

This tag declares a bundle that can be referenced in a data binding expression, such as in the following example:

```
<netui:label value="{bundle.someMessages}"/>
```

For more information, see the topics about the @jpf:message−resources Annotation and the <netui−data:declareBundle> Tag.

pageInput

When you have used one or more <netui−data:declarePageInput> tag(s) in your JSP, you can use the pageInput context to display the value of member variables defined in the page flow controller class. The values are read only. For example, a page flow named PageInputController.jpf contains an action method such as the following:

```
  /**
   * @jpf:action
   * @jpf:forward name="simple" path="simple.jsp"
   */
   public Forward simple()
   {
       Forward f = new Forward("simple");
       f.addPageInput("fooBean", new FooBean());
       return f;
   }
```

In the same page flow, a JavaBean named fooBean is defined as follows:

```
    public static class FooBean
    {
        private String foo = "This is a Foo String";

        public String getFoo()
        {
            return foo;
        }

        public void setFoo(String foo)
        {
            this.foo = foo;
        }
    }
```

A JSP page in this page flow might contain the following tags, which identify the type of data that will be used in the JSP, and displays the read−only value:

```
<netui−data:declarePageInput name="fooBean" type="pageInput.PageInputController.FooBean"/>
    ...
<netui:label value="{pageInput.fooBean.foo}"></netui:label>
```

At runtime in the rendered JSP, the label's value will be: This is a Foo String

For more information and a more interesting example, see Declaring Page Inputs.

Lifetime and Scalability of Data Binding Contexts

The data binding contexts available in WebLogic Workshop differ considerably in scope. The data of some contexts remains available globally for extended periods of time, while other data is short−lived and available locally. Design decisions to store large amounts of data in a context with too wide a scope can greatly impact scalability, and should generally be avoided. The following table summarizes lifetime and scalability considerations for the various data binding contexts, ordered from high impact to relatively low impact:

Context Name	***Lifetime and Scalability***
`globalApp`	Data in the globalApp context is available from the first request to a page flow until the end of the user session. In other words, the data in this context is maintained for each user currently using the web application. Try to limit the data that you store in this context, and consider using database storage instead for continued data persistence. In clustered (distributed) sessions, non−transient globalApp data is serialized across the cluster.
`session`	The lifetime and scalability of session data is comparable to data in the globalApp context object. In other words, for this context you should similarly try to limit the data that you store in this context, and consider using database storage instead for continued data persistence.
`pageFlow`	Data in the pageFlow context is available throughout the span of the user's interactions with a page flow, including all JSP pages. When exiting a page flow, data that is contained in variables defined in the JPF is no longer available (and the associated memory becomes available for renewed use). Although the scope of pageFlow data is more limited than that of globalApp and session data, storing large amounts of data should be done with care. All pageFlow data is stored indirectly in the session object, so any non−transient data will be serialized across the cluster, and (transient and non−transient) data is maintained separately for each user for the duration of the page flow.
`request`	The request object stores all the data pertaining to the request made by a user − for instance, by submitting a form or clicking a link − and the handling of the request via the JPF's action method (and possibly intermediary actions) before rendering the next JSP back to the browser. Within the action method, attributes can be added to the request and, depending on some decision logic, one of various JSPs can subsequently be called to bind to the data. Once the page is loaded on the browser, the data is no longer available.
`actionForm`	The actionForm object makes form bean properties available to form fields in a JSP. When this JSP calls a action method upon form submission, that in turn loads a JSP with a form, the actionForm object binds to the same form bean properties on the next JSP (unless you explicitly override this by calling a different form bean). The amount of data available in an actionForm is determined by the form bean. A properly designed form bean, as a specific instance of good class design, should contain all and only related properties.
`pageContext`	Data stored in the pageContext are only available for the duration of a single JSP page.
`url`	The url context binds data to the parameters in the URL in a read−only fashion. It is not possible to bind multiple values to a single parameter.

`container`	The container context only stores the current item (record) of a larger data set. Its lifetime is bound by the lifetime of the complex databinding tag using the container, such as a netui−data:Repeater tag.

Good state management is only one aspect of scalability for J2EE−based web applications. The design and use of EJBs, communication with databases, and the handling of query results are just a few other factors that need to be carefully considered when considering scalability.

Accessing Data Binding Context Properties

Data binding context elements referenced in a JSP are accessed using the 'dot' notation, and the entire expression must always be enclosed in braces:

```
<netui:textBox dataSource="{actionForm.postalCode}"/>
```

In the example, the netui:textBox binds to a form bean's postalCode property, which is a simple Java type. In general, you will want to bind to simple types such as a String, int, float, or Boolean. To include braces as part of the expression itself, you should offset a brace character with a backslash, "\". The property can also be a variable of an object that is for instance defined in the JPF file, or is part of a request. For more information, see pageFlow and request above.

You can also bind to a data element that is part of a Java array, List, or Map. References to elements in an array or List are made through standard Java array indexing syntax. For example:

```
{actionForm.purchaseOrder.lineItem[4].unitPrice}
```

The data binding expression references the unit price of the fifth line item in a customer's purchase order.

To bind to a value contained in a Java Map class, such as a HashMap, you can reference its key:

```
{pageFlow.inhabitantsMap['San_Francisco']}
```

In the example, the value associated with the key San_Francisco which is contained in inhabitantsMap is accessed.

If you want to bind to multiple elements in a complex data set, you should use a Netui Data Binding tag such as a repeater. For more information, see Presenting Complex Data Sets in JSPs.

In addition to binding to variables that are part of a form bean or a Java object, you can also bind to an XML document's elements and attributes using the same 'dot' notation and array indexing. A perhaps easier and more powerful way to access and manipulate XML data is through XML beans. You can then bind to XML elements through the XML bean in the exact same way as you would reference a form bean or Java object. For more information, see Getting Started with XMLBeans. You can also display the XML elements by using NetUI Data Binding tags in the same manner as you would with any complex data set. For more information on complex data sets, see Presenting Complex Data Sets in JSPs.

Binding to Properties with Periods in Their Names

If the name of the property being accessed contains a period (.), enclose the property name in square brackets. For example, assume that a property is added to the Session object where the property has periods in its name.

```
getSession().setAttribute("property.with.periods.in.its.name", "StringValue");
```

To bind to this property, enclose the property name in square brackets.

```
<netui:label value="{session['property.with.periods.in.name']}"/>
```

Related Topics

Using Form Beans to Encapsulate Data

Designing User Interfaces in JSPs

How Do I: Customize Message Formats in Page Flows?

How Do I: Use Multiple Forms in a JSP?

JSP Tags Reference

Samples

Data Binding Contexts Sample

Using a Wizard to Collect User Infomation Sample

<netui−data:repeater> and related Tags Sample

Multiple Forms on a Page Sample

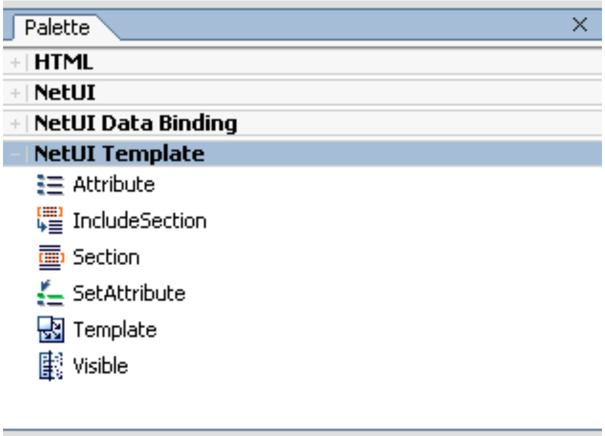
Designing User Interfaces in JSPs

When you create a page flow, WebLogic Workshop provides various tag libraries containing tags you can use to design a JSP. Each JSP file generated by the Page Flow Wizard is defined to include <% taglib...> statements, which reference the tag libraries. For example:

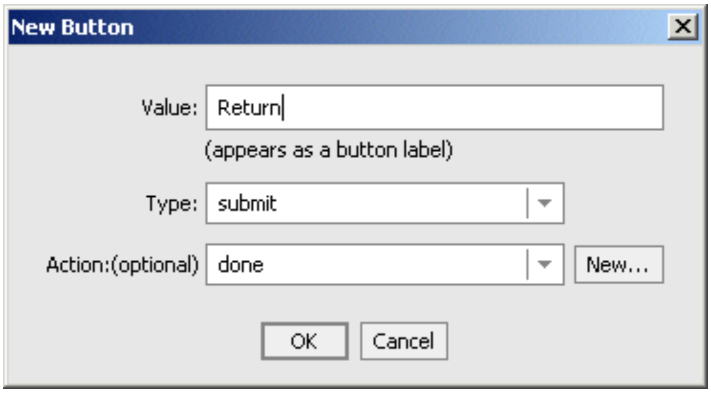
```
<!--Generated by WebLogic Workshop-->
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>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

Note: WebLogic Workshop puts TLD and TLDX library files in your web projects /WEB−INF directory. The TLD files are J2EE−standard tag library descriptor files. The TLDX files are WebLogic Workshop specific files that contain information on how the tags are supposed to behave within the IDE. This includes display, property sheet, and other information.

In the IDE, the tags are available in the ***Palette*** window when a JSP file is open in Design View or Source View. For example:



You can drag and drop these tags from the ***Palette*** window to the Design View. In some cases, a dialog window will appear that allows you to assign actions to tags that work with form beans. For example:



Value: Return

Type: submit

Action:(optional) done

Note: As you move a tag's icon from the Palette pane to the Design View canvas, you will notice on the Design View canvas that the insertion point is shown by a vertical line. Keep this insertion point in mind because it will determine the initial location of the tag's code in the JSP.

The JSP Designer is intended for developers who need to quickly assemble JSP pages based on the NetUI, NetUI Data Binding, and NetUI Template tags. As a developer, you can can create initial drafts of these pages and determine the form actions and other contexts to display simple or complex data to users. In the JSP Designer, you can take advantage of "drop wizards" and (in Source View) additional code completion tools to assist your development work. The JSP Design View is not intended to serve as a full JSP graphical editor, and you may switch into Source View to enter text, or use your favorite JSP editing product to complete the page, adding more polish to the presentation.

This topic describes the various elements in the ***NetUI*** and ***NetUI Template*** libraries. These libraries contain typical components used to design a user interface such as labels, radio buttons, and links. Many of the components in these libraries, in particular form components, allow you present data to the user or elicit data from the user.

In addition, the ***NetUI Data Binding*** library contains specialized components used to display complex data to users. The NetUI Data Binding tags are discussed separately in Presenting Complex Data Sets in JSPs.

Note: The ***Palette*** window also provides a tag library called ***HTML***, which contains several standard HTML tags. These HTML tags do not have any special attributes that interact with the page flow runtime.

NetUI Tags

The NetUI components in the ***Palette*** window are stored in the netui−tag−html.tld library, which is imported into the JSP by default:

<%@ taglib uri="netui−tags−html.tld" prefix="netui"%>

When you add a NetUI component to your JSP, you will notice that the component name is prefixed with "netui", for example:

```
<netui:button value="Start" action="letsGo" type="submit"/>
```

The following table summarizes common use of the NetUI tags. For a detailed description of a tag and its attributes, see JSP Tags Reference, or in Source View, place your cursor inside the tag and press F1:

Typical Tag Use	***Tag name***	***Summary***
Navigation	anchor	Generates a URL−encoded hyperlink to a specified URI.
	imageAnchor	Generates a URL−encoded hyperlink to a specified URI with an image enclosed as the body.
	button	Generates a button with a read−only name. Typically used to reset or submit a form, or to trigger a specific action (such as "signing out").
	imageButton	Generates an image acting as a button. Includes the URL of the image and (optionally) a rollover image. Typically used to submit a form.
Read / Write Data (uses dataSource, defaultValue attributes)	form	Represents an input form that is associated with a form bean whose properties correspond to the various fields of the form.
	checkBox	Generates a checkbox which binds to a form bean property.
	checkBoxGroup	Groups a collection of CheckBoxOptions, and handles databinding of their values. For an example, see A Detailed Page Flow Example.
	checkBoxOption	A checkbox whose state is determined by its enclosing checkBoxGroup.
	hidden	Generates a hidden form field.
	radioButtonGroup	Groups a collection of RadioButtonOptions, and handles databinding of their values. For an example, see A Detailed Page Flow Example.
	radioButtonOption	A radio button whose state is determined by its enclosing RadioButtonGroup.
	select	Renders a dropdown list. For an example, see A Detailed Page Flow Example.
	selectOption	An option whose state is determined by its enclosing select component.
	textArea	Renders a databound TextArea.
	textBox	Renders a databound TextBox.
Error Reporting	bindingUpdateErrors	A development−time aide that displays messages for data binding update errors that occurred when a form was posted. The messages are displayed on the command window. By default, this tag is disabled when the server is running in Production mode.
	error	Renders a single error message with a given error key value. For more information, see Validating User Input.

	errors	Renders the set of error messages found. For more information, see Validating User Input.
	exceptions	Renders formatted exception messages.
Parameters	parameter	Writes a URL parameter to a URL on its parent tag. For an example, see the url data binding context in Using Data Binding in Page Flows.
	parameterMap	Writes each parameter in a map of URL parameters to a URL on its parent tag.
Other	base	Provides the base for every URL on this page.
	content	Display read−only standard or dynamically−generated text.
	image	Generates an image.
	label	Generates styled read−only text based on an data binding expression or a literal value.
	node	Instantiates a TreeNode object that will get added to the parent tag (either a Tree or another Node).
	tree	Renders a tree control represented by a set of TreeNode objects.

Netui Template Tags

The NetUI Template components in the ***Palette*** window are stored in the netui−tag−template.tld library, which is imported into the JSP by default:

<%@ taglib uri="netui−tags−template.tld" prefix="netui−template"%>

When you add a NetUI Template component to your JSP, you will notice that the component name is prefixed with "netui−template", for example:

<netui−template:visible visible="true">Is Visible</netui−template:visible>

The tags in this library are typically used to define site templates. A template page is a JSP page defining the overall look and feel of a set of pages. It does this by providing an overall layout structure, style, and design of the page, and defining placeholders for content. These placeholders are called sections. A content page is a JSP page containing the content. The content page provides the sections of content plugged into the template. Combining the content page and the template page creates the overall look and feel of the content. The big advantage of templates is the ability to change the look and feel of a set of pages without being forced to modify each page.

The NetUI Template library defines two sets of tags. The first set is used in a JSP template page to define placeholders where content pages provide content. The second set of tags is used within a content page to indicate the template to use and to define the content that will be presented within the template.

The following table summarizes common use of the NetUI Template tags. For a detailed description of a tags and its attributes, see JSP Tags Reference, or in Source View, place your cursor inside the tag and press F1:

Tag Set	***Tag name***	***Summary***
Template Page	attribute	Defines an attribute in a template file. Used in conjunction with setAttribute in a content JSP.
	includeSection	Defines a section in a template file that can be further filled in by a content JSP, using the section tag.
Content Page	setAttribute	Set the value of an attribute declared in a template file.
	section	References an includeSection and defines the content to be included.
	template	Specifies the template page to use.
Either	visibility	Makes a section visible or invisible.

To see a sample application that uses the template tags, please go to the following page flow and check the JSP files:

`<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\simpleflow`

Also see the template.jsp and header.jsp files in:

`<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\resources\jsp`

Related Topics

Presenting Complex Data Sets in JSPs

A Detailed Page Flow Example

Getting Started with Page Flows

Using Form Beans to Encapsulate Data

JSP Tags Reference

Presenting Complex Data Sets in JSPs

Data binding is the process that ties data to presentation tags in JSPs, thereby creating dynamic web applications that are easy to build and maintain. This topic describes the ***NetUI Data Binding*** tags that are used to display large(r) data sets.

NetUI Data Binding tags are complex databinding tags used to render entire data sets to a web page. These tags are stored in the netui−tag−databinding.tld library, which is imported into the JSP by default:

```
<%@ taglib uri="netui−tags−databinding.tld" prefix="netui−data"%>
```

When you add a NetUI Data Binding component to your JSP, you will notice that the component name is prefixed with "netui−data", for example:

```
<netui−data:grid dataSource="{pageFlow.allRows}" name="{pageFlow.gridName}">


    ...

</netui−data:grid>
```

The NetUI Data Binding tags fall into one of the following categories:

- Method invocation tags
- Repeater tags
- The CellRepeater tag
- Grid tags

The functionality of the various tags are described below. For a detailed description of a tag including its attributes, see JSP Tags Reference, or in Source View, place your cursor inside the tag and press F1.

Note: The netui−data:getData tag, used to bind data to a JSP's page context, is discussed in Using Data Binding in Page Flows. For information on the netui−data:message and netui−data:messageArg tags, see How Do I: Customize Message Formats in Page Flows?

Method Invocation Tags

The method invocation tags netui−data:callControl and netui−data:callPageFlow are used to invoke methods on Controls or PageFlowController instances. The former tag allows access to controls directly from a JSP, while the latter allows controller methods to be called directly from the JSP. While Page Flow actions are not meant to be invoked with the netui−data:callPageFlow tag, calling JPF code from the JSP is useful for localizing code, which is used in many different pages, in a single location.

Repeater Tags

The tags in the repeater tag set are used to render data sets into a JSP page. A reapeater tag is a tag that, given a data set, renders each item or a subset of items in the data set into a page. Let's take a look at an example:

```
<netui−data:repeater dataSource="{pageFlow.strArr}">
   <netui−data:repeaterHeader>
      <ul>
   </netui−data:repeaterHeader>
   <netui−data:repeaterItem>
      <li>
         <netui:label value="{container.item}" />
      </li>
   </netui−data:repeaterItem>
   <netui−data:repeaterFooter>
      </ul>
   </netui−data:repeaterFooter>
</netui−data:repeater>
```

The top level tag is netuid−data:repeater, which binds to the complex data set. In the example the complex data set is an array of Strings, which was added to the page flow. The netui−data:repeaterHeader and netui−data:repeaterFooter tags are not repeated but appear only once as the header and the footer. In the example these are used to start and end a bulleted list. The netui−data:repeaterItem contains the code that is repeated. Typically it will contains some html formatting code − in the example <li> and </li> − plus a netui tag that binds to each item in the dataset using the container data binding context. The example will generate a bulleted list with each item in the String array displayed as a separate bullet.

Notice that the repeater tags use tags from the other tag libraries to create formatted data. This is how you ensure that data is displayed in a meaningful way to the user. Another example of netui−data:repeater, which uses a table to format the data, is shown next:

```
<netui−data:repeater dataSource="{pageContext.vec}">
   <netui−data:repeaterHeader>
      <table border="1">
         <tr>
            <td><b>index</b></td>
            <td><b>name</b></td>
         </tr>
   </netui−data:repeaterHeader>
   <netui−data:repeaterItem>
      <tr>
         <td>
            <netui:label value="{container.index}" />
         </td>
         <td>
            <netui:label value="{container.item}" />
         </td>
      </tr>
   </netui−data:repeaterItem>
   <netui−data:repeaterFooter>
      </table>
   </netui−data:repeaterFooter>
</netui−data:repeater>
```

The previous two examples showed read−only examples of displaying complex data. However, it is also possible to display complex data in a form and allow the user to change the data. For example, to display a

String array and allow the user to change the values, you can use this code:

```
<netui:form action="submit">
   <netui−data:repeater dataSource="{pageFlow.strArr}">
      <netui−data:repeaterHeader>
         <ul>
      </netui−data:repeaterHeader>
      <netui−data:repeaterItem>
         <li>
            <netui:textBox dataSource="{actionForm.name}" defaultValue="{container.item}" />
         </li>
      </netui−data:repeaterItem>
      <netui−data:repeaterFooter>
         </ul>
      </netui−data:repeaterFooter>
   </netui−data:repeater>
   <netui:button type="submit" value="submit"></netui:button>
</netui:form>
```

This example will display a bulleted list of strings, with each string in a separate form field. Notice that the netui:textBox receives its default value from the container context and uses the form bean's name property to submit the value. The form bean variable will have to be a String array:

```
public static class SubmitForm extends FormData
{
   private java.lang.String[] name;
   ...
```

You can easily create a variable array for a form bean in Action View or Flow View by selecting the array checkbox for a form bean variable in the ***Edit Form Bean*** dialog. The necessary getter and setter methods to enable read−write access will then be automatically created for you.

In the action method that is called when submitting the form, you can use standard Java array notation to access the individual elements:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="submitPage1.jsp"
*/
protected Forward submit(SubmitForm form)
{
   System.out.println(form.name[1]);
   ...
   return new Forward("success");
}
```

In the above example, instead of using a form bean to hold the submitted values, we could have used the same String array that we read from to write the values to upon submit. This would require changing the netui:textBox tag in the above JSP code as is shown next:

```
<netui:form action="submit">
   <netui−data:repeater dataSource="{pageFlow.strArr}">
      ...
      <netui−data:repeaterItem>
         <li>
            <netui:textBox dataSource="{container.item}" />
         </li>
      </netui−data:repeaterItem>
      ...
   </netui−data:repeater>
   <netui:button type="submit" value="submit"></netui:button>
</netui:form>
```

Notice that the String array is stored in the pageFlow context, so you can access it accordingly in the action method:

```
  /**
   * @jpf:action
   * @jpf:forward name="success" path="submitPage1.jsp"
   */
  protected Forward submit()
  {
      System.out.println(strArr[1]);
      return new Forward("success");
  }
```

Nested Repeater Tags and Multi−dimensional Arrays

You can use nested <netui−data:repeater> tags to render a multi−dimensional array.

Suppose you have the following multi−dimensional array.

```
        String[][] multiDimArr = {
                                    {"1","2","3","4","5"},
                                    {"a","b","c","d","e"},
                                    {"!","@","#","$","%"},
                                    {"A","B","C","D","E"}
                                 };
```

You can use one eater> tag nested inside another to iterate over every element in the multi−dimensional array.

The outer <netui−data:repeater> tag iterates over the outer array.

```
        <table border="1">
        <netui−data:repeater dataSource="{pageContext.multiDimArr}">
            ...
        </netui−data:repeater>
        </table>
```

The inner <netui−data:repeater> tag iterates over the elements of the inner arrays.

```
        <table border="1">
        <netui−data:repeater dataSource="{pageContext.multiDimArr}">
            <tr>
                <netui−data:repeater dataSource="{container.item}">
                    <td><netui:label value="{container.item}" /></td>
                </netui-data:repeater>
            </tr>
        </netui-data:repeater>
        </table>
```

Notice that the two data binding expressions {container.item} are not synonymous.

The first occurance of {container.item} refers to the elements of the outer array, for example, {"1", "2", "3", "4", "5"}, {"a","b","c","d","e"}, etc. The second occurance of {container.item} refers to the elements of the current inner array, for example, "1", "a", "!", etc. The result is the following HTML table.

```
<table border="1">
    <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
        <td>4</td>
        <td>5</td>
    </tr>
    <tr>
        <td>a</td>
        <td>b</td>
        <td>c</td>
        <td>d</td>
        <td>e</td>
    </tr>
    <tr>
        <td>!</td>
        <td>@</td>
        <td>#</td>
        <td>$</td>
        <td>%</td>
    </tr>
    <tr>
        <td>A</td>
        <td>B</td>
        <td>C</td>
        <td>D</td>
        <td>E</td>
    </tr>
</table>
```

In the browser the table appears as follows.

1	2	3	4	5
a	b	c	d	e
!	@	#	$	%
A	B	C	D	E

Additional Repeater Options

The Repeater also provides more advanced features to customize how the items in the data set are displayed, in particular by using *padding* and *choice*.

Padding is used in the repeater to create a regular display of an irregular data set. The netui−data:pad tag provides attributes such as maxRepeat, minRepeat, and padText that controls how many container items are displayed, and what to display instead if there are less items than expected. For more information, see the netui:pad reference documentation.

The choice feature is used to create a different display for an item depending on its value. This functionality is implemented using two tags, namely netui−data:choiceMethod and netui−data:choice. The former method is used to indicate the item that is being considered, while the latter method defines a possible value for this item and the display to be rendered when this value is found. Both tags must occur within a netui−data:repeaterItem tag.

Also, the related netui−data:methodParameter tag is embedded within the netui−data:choiceMethod tag, as shown in the following example. You can use the netui−data:methodParameter tag to get the value that will be used by the choice tags.

```
<netui−data:repeater dataSource="{pageFlow.cart.lineItemList}">
    <netui−data:repeaterHeader>
        ...
    </netui−data:repeaterHeader>
    <netui−data:repeaterItem>
        <netui−data:choiceMethod object="{pageFlow}" method="getShippingState">
            <netui−data:methodParameter value="{container.item.shipState}"/>
        </netui−data:choiceMethod>

        <netui−data:choice value="inTransit">
            ...
            <netui−html:label value="{container.item.name}"/>
            <netui−html:label value="In Transit"/>
            ...
        </netui−data:choice>

        <netui−data:choice value="arrived">
            ...
            <netui−html:label value="{container.item.name}"/>
            <netui−html:label value="Arrived"/>
            ...
        </netui−data:choice>

        <netui−data:choice value="notShipped">
            ...
            <netui−html:label value="{container.item.name}"/>
            <netui−html:label value="Not Yet Shipped"/>
            ...
        </netui−data:choice>

        <netui−data:choice default="true">
            ...
            <netui−html:label value="{container.item.name}"/>
            <netui−html:label value="Error;status unknown./>
            ...
        </netui−data:choice>
    </netui−data:repeaterItem>
    <netui−data:repeaterFooter>
        ...
    </netui−data:repeaterFooter>
</netui−data:repeater>
```

In the example, the netui−data:choiceMethod tag indicates that the item's shipState must be used to decide which formatting to display. Specifically, its method attribute specifies the method getShippingState, which is a method that is defined in the pageFlow context object, as is indicated in the tag's object attribute. This method getShippingState takes a single argument, the current item's shipState, as is defined in the nested netui−data:methodParameter tag. The various netui−data:choice tags implement what to display given the return value of the getShippingState method for the current item, that is, inTransit, arrived, notShipped, or any other value using the default choice tag. Notice that in the current example some html formatting tags were removed for the sake of clarity. However, you can add html tags to each netui−data:choice tag to render different formatting, for instance to display error values in red, expected values in green, and so forth.

The previous example uses a number of possible static values for the netui−data:choice tag. However, it is also possible to bind its value attribute, as is shown in the following example:

```
<netui−data:repeater dataSource="{pageFlow.cart.lineItemList}">
    <netui−data:repeaterHeader>
        ...
    </netui−data:repeaterHeader>
    <netui−data:repeaterItem>
        <netui−data:choiceMethod object="{pageFlow}" method="getSpecialItems">
            <netui−data:methodParameter value="{container.item.name}"/>
        </netui−data:choiceMethod>

        <netui−data:choice value="{container.item.name}">
            ...
            <netui−html:label value="{container.item.name}"/>
            <netui−html:label value="Special"/>
            ...
        </netui−data:choice>

        <netui−data:choice default="true">
            ...
            <netui−html:label value="{container.item.name}"/>
            ...
        </netui−data:choice>
    </netui−data:repeaterItem>
    <netui−data:repeaterFooter>
        ...
    </netui−data:repeaterFooter>
</netui−data:repeater>
```

In the example a new method getSpecialItems, referenced in the netui−data:choiceMethod tag, is used to evaluate each item's name, and it returns the name for special items only (depending on some business logic). The first netui−data:choice tag's value attribute in the example binds to the current item's name, so for special items this value will match the return value of the method, and the formatting defined in this choice tag will be used. All other items will be formatted as defined in the default choice tag.

CellRepeater

The netui−data:cellRepeater is like the netui−data:Repeater tag in that it is used to display each item in a complex data set. The difference is that the netui−data:cellRepeater displays each item in the cell of a table. All the html tags necessary to create the table are automatically created by the cellRepeater tag. For example:

```
<netui−data:cellRepeater dataSource="{pageFlow.itemArray}" columns="{pageFlow.numColumns}" >
   Item: <netui:label value="{container.item}"/>
</netui−data:cellRepeater>
```

This example creates a table with a certain number of columns as given in pageFlow.numColumns and as many rows as necessary to display all the items in the data set. Each cell in the table will contain *Item: 'the actual item'*. The cellRepeater tag has other attributes to determine the formatting of the table and the individual cells. For more information, see the reference documentation for this tag.

Grid Tags

A final repeating databinding tag set is the Grid tag set, which is used to render data from a javax.sql.RowSet object. A RowSet object is used to hold a typically disconnected data set that contains data from a relational database (for more information, see its API reference at http://java.sun.com). The Grid tag set provides the ability to display, page, sort, and filter data. The Grid tag does not display items using standard html formatting tags, and does not use the *container* data binding context (except for the tag netui−data:expressionColumn). Instead the grid uses specific grid tags to enable table−like formatting and to accomplish data binding, assuming that the same number of columns are rendered for each row in the table.

In WebLogic Workshop you can easily generate a page flow that uses grid tags to display the records in a relational database and contains the action methods to update, insert and delete records. For more information, see How Do I: Create a Database Control Page Flow?

Let's take a look at an example to understand the various grid tags (to see and run the entire example, go to the databasePageFlowController.jpf Sample):

```
<netui−data:grid dataSource="{pageFlow.allRows}" name="{pageFlow.gridName}">
   <netui−data:gridStyle styleClassPrefix="gridStyle"/>
   <netui−data:pager renderInHeader="true" action="begin" renderInFooter="true"/>
   <netui−data:columns filterable="true" filterAction="begin" sortAction="begin" sortable="true">
      <netui−data:anchorColumn action="getItems" addRowId="true" title="Details"/>
      <netui−data:anchorColumn action="updateItems" addRowId="true" title="Edit"/>
      <netui−data:basicColumn title="Itemnumber" name="itemnumber"/>
      <netui−data:basicColumn title="Itemname" name="itemname"/>
      <netui−data:basicColumn title="Quantityavailable" name="quantityavailable"/>
      <netui−data:basicColumn title="Price" name="price"/>
   </netui−data:columns>
</netui−data:grid>
```

The following tags are used in the example:

- The netui−data:grid tag is the top level tag and binds to the rowset data through its dataSource attribute.
- The netui−data:gridStyle tag is used to enable specific formatting for parts of the tag, for example to

create a different display of alternating rows in the table. It references CSS styles that should be defined in the file resources/css/style.css, which is automatically included in your web project.

- The netui−data:pager tag is used to enable paging through the data. Instead of binding and presenting all the data at once in the JSP, you use this tag to bind and display a number of items at a time. You can determine how many items to display at a time and where to display the pager navigation text "*Page # of # Prev Next*". In the example the default number of items to display is used, and the navigation text is placed both in the header and the footer of the table.
- The netui−data:columns tag is used as the containing tag for other netui−data tags that are used to display individual columns. The tags contained inside this tag describe the header of the column followed by the repeating data elements to be displayed in the table.
- The netui−data:anchorColumn tag creates an anchor for each cell in the column. In other words, the items in this column will be linked and clicking the link will trigger the action indicated on the netui−data:anchorColumn tag. The attribute addRowId identifies the current row in the record.
- The netui−data:basicColumn tag creates cells without anchors.
- For both netui−data:basicColumn and netui−data:anchorColumn tags you can specify the title and name attributes. If the title attribute is specified but no name attribute is specified, the column header and all the cells in the column will be given the name specified in the title attribute. In the first anchorColumn in the example, the header and all the cells in the column will say *Details*, which is linked for all the cells (but not the header).

  If the title and name attributes are both specified, the header will have the title name and the cells in the column will be filled with the values of a record field as specified in the name attribute. In our example, the first basicColumn has a header named *Itemnumber*, followed by cells which display the contents of the record field itemnumber in the RowSet. In addition, when both name and title are specified, you can optionally sort the data by value in this column (the default is sortable in ascending order). To sort data by column, you click the linked header for this column (in our example, *Itemnumber* will appear linked). Finally, when both name and title attributes are specified, you can also optionally filter the data (the default is true). If filtering is enabled for a column, a filter icon will appear to the right of the header name. When you click this item, a dialog appears with a large number of filtering options. If the filter icon appears in blue, then the values in this columns are filtered. You can enable filtering in multiple columns at the same time.

Note: When you filter the data on a column, a new SQL query is generated that is run by the page flow, and the matching records are returned to the RowSet object.

In addition to the tags used in the example, there are several additional tags:

- The netui−data:imageColumn tag is used to create an image in the cells of a column. This tag must be contained by the netui−data:columns tag.
- The netui−data:expressionColumn tag is used to combine multiple record fields for display in a single column. This tag must be contained by the netui−data:columns tag. For example, we could add the following tag code to the above example:

```
<netui−data:columns filterable="true" filterAction="begin" sortAction="begin" sortable="true">
  <netui−data:anchorColumn action="getItems" addRowId="true" title="Details"/>
  ...
  <netui−data:expressionColumn title="Itemname, Itemnumber" value=
    'name=<b>{container.item.itemname}</b>
<br>number=<i>{container.item["itemnumber"]}<i/>'/>
</netui−data:columns>
```

> Notice that the tag has a title attribute like the other netui−data:...Columntags. The attribute is used to name the column header. The value attribute is used to determine the contents of the cells. In the example the entire value statement is given on the second bolded line. Notice that you can combine text and actual field values, which are accessed using the container data binding context. Also notice that you can apply html formatting. A cell in the column would look like this:
>
> name=***office chair***
> number=*628*

When you select the Grid icon from the ***Palette*** window and drag and drop this onto your JSP's Design View or Source View, a dialog will assist you in setting up the grid tag table. You can select the data source, whether to use a Pager (that is, define a netui−data:pager tag), the number of data columns (with netui−data:basicColumn tags), and an anchor column (that is, define a netui−data:anchorColumn tag) that you can make filterable and/or sortable. After clicking ***OK***, go to Source View to associate each column with the corresponding record field in your data source through the column's name attribute.

Related Topics

Using Data Binding in Page Flows

Using Form Beans to Encapsulate Data

Designing User Interfaces in JSPs

How Do I: Customize Message Formats in Page Flows?

How Do I: Create a Database Control Page Flow?

Sample Code

<netui−data:repeater> and related Tags Sample

Declaring Page Inputs

The <netui−data:declarePageInput> tag allows you to declare variables that will be passed from the page flow controller to the JSP page. You can assign an initial value to each variable that will be used (for display purposes only) on the JSP.

In the source JSP file, the <netui−data:declarePageInput> tag helps to indicate clearly the type of data that is expected at run time. This information about the incoming data helps your team understand any data dependencies this JSP page may have.

At design time each <netui−data:declarePageInput> tag, once added to a JSP page, enables an additional WebLogic Workshop IDE feature in which the properties on the source object can be discovered. For example, after a database control is referenced in a page flow action, and the <netui−data:declarePageInput> tag is placed on the JSP page, you can use the Data Palette pane's "Page Inputs" section to select one or more properties, such as ITEMNAME and PRICE. When you drag a property from the Page Inputs section to your JSP page, a read−only <netui:label> tag will be added with the appropriate data binding expression already completed for you. For example:

```
<netui:label value="{pageInput.item.itemname}">
```

Declare Page Input Sample

The WebLogic Workshop sample application contains a page flow that demonstrates this page input feature. Start in the following directory:

```
<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/pageInput
```

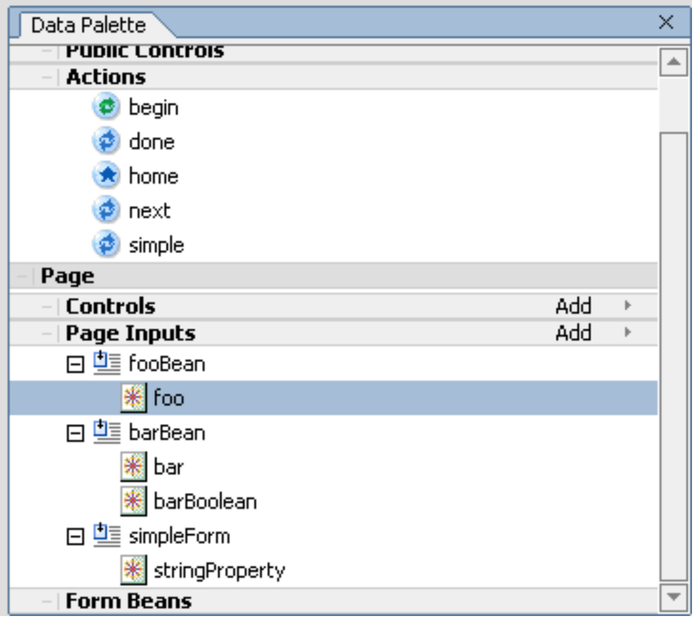
The sample page flow contains two JSP files, simple.jsp and next.jsp, that declare page inputs. The simple.jsp page contains these tags:

```
<netui−data:declarePageInput name="fooBean" type="pageInput.PageInputController.FooBean"/>
<netui−data:declarePageInput name="barBean" type="pageInput.PageInputController.BarBean"/>
<netui−data:declarePageInput name="simpleForm" type="pageInput.PageInputController.SimpleForm"/
```

In the page flow controller class, an action method named simple is defined as follows:

```
 /**
  * @jpf:action
  * @jpf:forward name="simple" path="simple.jsp"
  */
   public Forward simple()
   {
       Forward f = new Forward("simple");
       f.addPageInput("fooBean", new FooBean());
       f.addPageInput("barBean", new BarBean());
       f.addPageInput("simpleForm", new SimpleForm());
       return f;
   }
```

At design time in simple.jsp, because the <netui−data:declarePageInput> tags were present on the page, the Data Palette pane contained a Page Input category. In the following sample screen, we have already expanded each JavaBean's entry to see the variables they contain:



Note that the Data Palette will not display the following data types:

primitive Java types
java.*
javax.*
com.sun.*
org.apache.struts.*
com.bea.*

By selecting the foo property and dragging it to the JSP Source View (taking care to note the insertion point), the IDE will generate the following tag attributes for us:

```
<netui:label value="{pageInput.fooBean.foo}"></netui:label>
```

You can then modify the tag, as we did in the sample application.

The page flow controller class, PageInputController.jpf, contains three JavaBean classes that provide the data for the page inputs in simple.jsp. For details, see the page flow source file.

For the next.jsp page inputs, this page flow uses a database control that retrieves several fields from an ITEMS table and loads them into an array. Here is a portion of the database control JCX file:

```
 * @jc:connection data-source-jndi-name="cgDataSource"
 */
 public interface ItemsDBControl2 extends DatabaseControl, com.bea.control.ControlExtension
 {
     static public class Item
     {
         public int itemnumber;
         public String itemname;
         public int quantityavailable;
         public double price;
     }

     static final long serialVersionUID = 1L;
```

```
    /**
     * @jc:sql statement="SELECT ITEMNUMBER, ITEMNAME, QUANTITYAVAILABLE, PRICE FROM ITEMS"
     */
     Item[] getAllItems() throws SQLException;
}
```

In the page flow controller class, one of the import statements is:

```
import pageInput.ItemsDBControl2.Item;
```

And the page flow calls the control:

```
 /**
  * @common:control
  */
  private pageInput.ItemsDBControl2 itemsDB;
```

The page flow also defines an action method that gets all items in the array.

```
 /**
  * @jpf:action
  * @jpf:forward name="success" path="next.jsp"
  */
  public Forward next()
      throws SQLException
  {
      Item[] items = itemsDB.getAllItems();
      return new Forward("success", "items", items);
  }
```

Notice the new Forward object's parameters:

- "success" is the Forward name, and its value matches the name attribute's value on the @jpf:forward annotation. This match causes navigation to the next.jsp page when the action is run.
- "items" is the name of the page input, as defined in a JSP that is part of this page flow.
- items is the name of the actual object from which data will be obtained. In this case, items is a table referenced by a database control called by the page flow.

In the next.jsp page that is part of this page flow sample, we can use a <netui−data:declarePageInput> tag and a <netui−data:repeater> to display the retrieved values. For example:

```
 <netui−data:declarePageInput name="items" type="ItemsDBControl2.Item[]"/>
   <p>&nbsp;</p>
   <p><b>An array of product items:</b>
   <netui−data:repeater dataSource="{pageInput.items}">
       <netui−data:repeaterHeader>
           <ul>
       </netui−data:repeaterHeader>
       <netui−data:repeaterItem>
           <li>
           <netui:label value="{container.item.itemname}" />
           </li>
       </netui−data:repeaterItem>
       <netui−data:repeaterFooter> </ul> </netui−data:repeaterFooter>
   </netui−data:repeater>
```

See the sample's pre−rendered source files in:

```
<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/pageInput
```

Then run the sample to view the results.

Related Topics

<netui−data:declarePageInput> Tag Sample

Presenting Complex Data Sets in JSPs

Using Data Binding in Page Flows

Calling Web Services and Custom Java Controls From A Page Flow

The following topic explains how call a web service or a Custom Java Control (JCS file) from a Page Flow.

In most cases, you communicate with a web service or Custom Java Control resource, you send a request to the resource, and then listen for a callback containing the response. But one limitation of Page Flows is that that they cannot hear callbacks from web−based resources. For this reason, Page Flows can only communicate with web−based resources that either have *synchronous* methods (methods where the request and the returned result are accomplished within the same method) or that have *polling interfaces* (an interface that allows the page flow to send the request through one method, and then collect the result through another method). Each of these communication techniques are described below.

Synchronous Communication

Suppose you have a web service that communicates synchronously with its clients, such as the HelloWorld.jws web service. Synchronous communication means that an individual method of the web service both takes requests and returns the results of those request, without using another callback method to return the results. The following web service communicates synchronously, because one and the same method Hello() both takes a request and returns the result, the String "Hello World!", to the client.

```
public class HelloWorld implements com.bea.jws.WebService
{
    /**
     * @common:operation
     */
    public String Hello()
    {
        return "Hello, World!";
    }
}
```

To call a web service's methods from a page flow, first declare the web service's control file within the page flow file.

```
public class MyPageFlow extends PageFlowController
{

    /**
     *  @common:control
     */
    private HelloWorldControl myControl;

        .
        .
        .


}
```

Now you can call the methods on the web service through the object myControl.

```
    /**
```

```
     * Action encapsulating the control method : HelloWorld
     *
     * @jpf:action
     * @jpf:forward name="success" path="result.jsp"
     */
    public Forward HelloWorld()
    {
        String result = myControl.HelloWorld();

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

However, it is generally bad design to base page flow/web resource communication on synchronous methods. This is because clients (the page flow) are blocked while they wait for the synchronous methods on the web resource to execute. If the resource takes a substantial amount of time to complete its work, the client will be blocked the same amount of time waiting for the resource to complete. For this reason, it is generally best to use a polling interface to control page flow / web resource communication whenever possible.

Polling Interfaces

In most cases a web service communicates with its clients through the use of *asynchronous callbacks*. This means that a client makes a request of a web service and then receives the result at a later time in the form of a callback message. Using this technique, the client does not have to halt its own processes while it waits for the callback message: it can go on to do other tasks, such as processing data, making other requests, etc.

But not all clients are capable of hearing callback messages from web services: Page Flows are one such client. As a workaround for this limitation, a Page Flow can communicate with a web service through the use of a *polling interface*. A polling interface allows a web resource to communicate with those clients that are unable to hear callbacks from the resource.

Polling interfaces work by provided one method where clients make requests of the web service, and another method where clients collect the result. Note that this technique does not rely on callbacks: the resource never invokes a callback handler on the client. Instead each method call in the polling process is client initiated.

Suppose there is a web service that presents a polling interface with three different methods: requestData(), isDataReady(), and getData(). Clients call requestData() to make a request of the web service, they call isDataReady() to see if the result is ready to be collected, and they call getData() to collect the completed result. To utilize this polling interface from a page flow, you first must declare the web service control file on the page flow file.

```
public class MyPageFlow extends PageFlowController
{

    /**
     *  @common:control
     */
    private MyWebServiceControl myControl;

        .
        .
        .


}
```

The following shows a typical routine for calling the methods of a polling interface from a Page Flow. In a typical routine, the polling is allowed to go on for a set amount of time, not indefinitely. (In the example below, the polling lasts 20 seconds. After that time, the user is forwarded to the error page.) How long you should poll the web service depends on the specifics of the target web service, your client application, and the web environment.

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="response.jsp"
 * @jpf:forward name="failure" path="error.jsp"
 */
protected Forward pollForData()
    throws java.lang.InterruptedException
{
    /**
     * Request data from the web service.
     */
    myControl.requestData();

    /**
     * Poll the isDataReady method 20 times, once a second,
     * to check if the result is ready.
     */
    for(int i=0; i
```

Related Topics

Calling a Synchronous Web Service Sample

Calling an Asynchronous Web Service Sample

Tutorial: Page Flow: Step 2: Processing Data with Page Flows

Handling Images and Binary Data in Page Flows

The following topic explains how to handle binary data, such as BLOB data and image files, in page flows.

Retrieving Binary Data

Uploading Binary Data

You can upload binary data like any other file type, using the <netui:fileUpload> tag. When you use a <netui:fileUpload> tag, the parent <netui:form> tag must have its enctype attribute set to "multipart/form−data".

```
<!−−
When using the <netui:fileUpload> tag, you must set the <netui:form> enctype attribute
−−>
<netui:form action="submitInsert" enctype="multipart/form−data">
    Upload Image File:
    <netui:fileUpload dataSource="{actionForm.contentUp}"/>
    </br>
    <netui:imageButton src="/WebApp/resources/images/insert.gif"/>
</netui:form>
```

The <netui:fileUpload> tag must upload to a Form Bean field of type org.apache.struts.upload.FormFile.

When uploading binary data, it is often convenient to provide two fields in the Form Bean to contain the binary data. One field is of type org.apache.struts.upload.FormFile, which holds the uploaded data. The other field is of type byte[] (or char[]), which holds the data in a form that can be processed by Java code. For instance you can transform the byte[] in a BLOB for insertion into a database, or pass it to a Servlet for display in a web browser.

```
<!−−
    When uploading binary data, it is convenient to have a field
    for uploading the data (of data type FormFile) and a field for handling
    the data in the page flow (of data type binary[]).
    −−>
public static class UploadForm extends FormData
{
    private org.apache.struts.upload.FormFile contentUp;
    private byte[] content;


    public FormFile getContentUp()
    {
        return contentUp;
    }

    public void setContentUp(FormFile theFile)
    {
        this.contentUp = theFile;
    }

    public byte[] getContent()
    {
        return content;
    }
```

```
        public void setContent(byte[] theFile)
        {
            this.content = theFile;
        }
    }
```

The following action method shows how to transform the uploaded org.apache.struts.upload.FormFile field into a byte[] on submission. Note that the Form Bean field contentUp holds the org.apache.struts.upload.FormFile data, while the content field holds the byte[] data.

```
    /**
     * @jpf:action
     * @jpf:forward name="show" path="showImage.jsp"
     */
    public Forward upLoadFile(UploadForm form)
        throws Exception
    {
        /*
         * The following stanza loads the uploaded binary data into a byte Array.
         */
        // Make a byte Array big enough to fit the binary file.
        byte[] byteArr = new byte[filesize];
        //Create an input stream to read the uploaded file.
        ByteArrayInputStream bytein = new ByteArrayInputStream(form.contentUp.getFileData());
        // Load the input stream into the byte Array.
        bytein.read(byteArr);
        // Close the input stream.
        bytein.close();
        // Load the byte[] into the content field.
        form.setContent(byteArr);

        // In a real world example, you might do something with the binary data
        // such as transform it, or save it in a database.

        return new Forward("show",form);
    }
```

Getting Binary Data from a Database

Binary data typically is stored as a BLOB or CLOB data type in a database, but it is processed by Java code in the form of a byte Array or char Array. BLOB and CLOB data can be retrieved from the data using a database control method such as the following.

```
    /**
     * @jc:sql
     * statement::
     * SELECT CONTENT FROM WEBLOGIC.DOCUMENT_CONTENT WHERE ID = {x}
     * ::
     */
    public java.sql.Blob getPHOTO(int x)
        throws SQLException;
```

The java.sql.BLOB data can be loaded into a byte Array for further processing, or, if it is an image, for display.

```
    byte[] byteArr = blob.getBytes(1, (int) blob.length());
```

Displaying Binary Data

Once you have an image stored in a byte Array, you can display the image using a Java Servlet. The Servlet below works by reading the binary data (in the form of a byte[]) from the request object and writing the byte[] to the response object. The most important part of this Servlet is the method call response.setContentType("image/gif") which sets the MIME type to an image type.

```
/*
 * This servlet reads image data (in the form of a byte[]) from the request object.
 * It outputs the byte[] as a visible image.
 */
public class ShowImageServlet extends HttpServlet
{
    protected void service(HttpServletRequest request, HttpServletResponse response) throws Ser
    {
        byte[] rgb = (byte[]) request.getAttribute("byArr");
        if (rgb != null)
        {
            response.setContentType("image/gif");
            OutputStream stream = response.getOutputStream();
            stream.write(rgb);
        }
        else
        {
            response.setContentType("text");
            response.getWriter().write("attribute byArr not found");
        }
    }
}
```

Once you have named and mapped this Servlet in the WEB−INF/weblogic.xml file, you can call it from a variety of different sources. Assuming that the Servlet above is named ShowImageServlet.java, the following naming and mapping elements in WEB−INF/weblogic.xml allow you to access the Servlet by the URL pattern showImage.

```
    <servlet>
        <servlet−name>showImage</servlet−name>
        <servlet−class>servlets.ShowImageServlet</servlet−class>
    </servlet>

    <servlet−mapping>
        <servlet−name>showImage</servlet−name>
        <url−pattern>showImage</url−pattern>
    </servlet−mapping>
```

To display the binary image data on a JSP page, you pass the byte[] to the request object and then call the Servlet with a <jsp:forward> tag.

```
    <% request.setAttribute("byArr", rgb); %>
    <!−−
    The image data is now on the request object.
    Forward the user to the showImage servlet.
    That servlet will process and display the image data contained on the request object.
    −−>
    <jsp:forward page="/showImage" />
```

Related Topics

Handling Binary Data Sample

Using JavaScript in Page Flow and Portal Applications

The following topic explains how to access page elements, such as forms and user input elements, with JavaScript.

Limitations of Traditional Naming Techniques

In a stand alone web application, JavaScript typically accesses page elements by the id and name attributes. Page elements are given unique id or name attributes...

```
<form name="myForm" onSubmit="return validateForm()" action="myAction">
   First Name: <input type="text" id="firstName"><br>
   Last Name:  <input type="text" id="lastName"><br>
   <input type="submit" value="Submit">
</form>
```

...and then the page elements are accessed by JavaScript functions which refer to the id and name attributes...

```
<script language="JavaScript">
function validateForm()
{
    var error_message = "";
    if (document.myForm.firstName.value =="") error_message += "- your first name \n";
    if (document.myForm.lastName.value =="") error_message += "- your last name \n";
    if(error_message)
    {
        alert("Please fill in the following fields:\n" + error_message);
        return false;
    }
    return true;
}
</script>
```

But when a web application is incorporated into a Portal application, this JavaScript technique does not suffice. This is because Portal applications may contain a collage of different web applications, with no guarantee that the page elements have unique id or name attributes. If two applications in the Portal each have page elements with the same id or name attributes, there is no way for JavaScript to access the one of these page elements, since JavaScript will always return the first page element it encounters as it searches for an element with a specified name or id. For this reason, Portal applications can be made to rewrite the name and id attributes of page elements, to ensure that each page element in the Portal has a unique identifier.

Ensuring Unique Names for Each Page Element

To ensure that Page Flow page elements have unique identifiers inside a Portal, use the tagId attribute to identify the page elements. Workshop will append Portal and portlet context information to the tagId and write the result to the id and name attributes that are rendered in the browser. For example, the following Page Flow elements have been given tagId attributes.

MyJSP.jsp

```
<netui:form tagId="myForm" onSubmit="return validateForm2()" action="myAction2">
```

```
   First Name: <netui:textBox tagId="firstName" dataSource="{actionForm.firstName}"/><b
   Last Name:  <netui:textBox tagId="lastName" dataSource="{actionForm.lastName}"/><br>
   <netui:button type="submit" value="Submit"/>
</netui:form>
```

When these Page Flow elements are contained in a Portal, they are rendered in the browser as the following HTML. Notice that Workshop has rewritten the id attributes to include context information about the Portal and portlet in which the Page Flow elements appear.

```
<form id="portlet_3_1myForm" method="post" onsubmit="return validateForm()">
    First Name: <input type="text" id="portlet_3_1{actionForm.firstName}" value=""/><br
    Last Name:  <input type="text" id="portlet_3_1{actionForm.lastName}" value=""/><br>
   <input type="submit" value="Submit"/>
</form>
```

Accessing Page Elements with JavaScript

To ensure that these page elements are available to client−side JavaScript, Workshop also provides a mapping function, getNetuiTagName(tagId, tag), that maps the tagId values to the re−written id (and name) values that are rendered in the browser. The mapping is encoded into a JavaScript object written to the HTML source of the page.

```
// Build the netui_names table to map the tagId attributes
// to the real id written into the HTML
if (netui_names == null)
   var netui_names = new Object();
netui_names.myForm="portlet_3_1myForm"
netui_names.firstName="portlet_3_1{actionForm.firstName}"
netui_names.lastName="portlet_3_1{actionForm.lastName}"
```

The function getNetuiTagName(tagId, tag) reads this mapping object to retrieve the actual id or name attribute rendered in the browser.

The following JavaScript expression accesses the form element on the page ***MyJSP.jsp*** above.

```
document[getNetuiTagName("myForm",this)]
```

The function call getNetuiTagName("myForm", this) resolves to the value portlet_3_1myForm. So the JavaScript expression points at the form element on the page.

```
document["portlet_3_1myForm"]
```

To access the firstName element on ***MyJSP.jsp*** above, use the following JavaScript expression.

```
document[getNetuiTagName("myForm",this)][getNetuiTagName("firstName",this)]
```

The following JavaScript function performs simple client−side validation on ***MyJSP.jsp***.

```
<script language="JavaScript">
function validateForm()
{
    var error_message = "";
    if (document[getNetuiTagName("myForm",this)][getNetuiTagName("firstName",this)].val
    if (document[getNetuiTagName("myForm",this)][getNetuiTagName("lastName",this)].valu
    if(error_message)
```

```
            {
                alert("Please fill in the following fields:\n" + error_message);
                return false;
            }
            return true;
        }
        </script>
```

Passing the tagId Value to HTML Tags.

If you need to pass the value of the tagId attribute to an HTML tag, use the <netui:getNetuiTagName> tag. The <netui:getNetuiTagName> returns the value of a specified tagId attribute. In the following example, the HTML tag <label> is given access to the value of the tagId attribute.

```
    <netui:form action="processData">
        <netui:radioButtonGroup dataSource="{actionForm.selection}">
            <label for="<netui:getNetuiTagName tagId="radio1"/>">Display Text 1</label><netui:r
            <label for="<netui:getNetuiTagName tagId="radio1"/>">Display Text 2</label><netui:r
            <label for="<netui:getNetuiTagName tagId="radio1"/>">Display Text 3</label><netui:r
        </netui:radioButtonGroup>
        <netui:button value="Submit" />
    </netui:form>
```

Related Topics

Validating User Input

<netui:getNetuiTagName> Tag

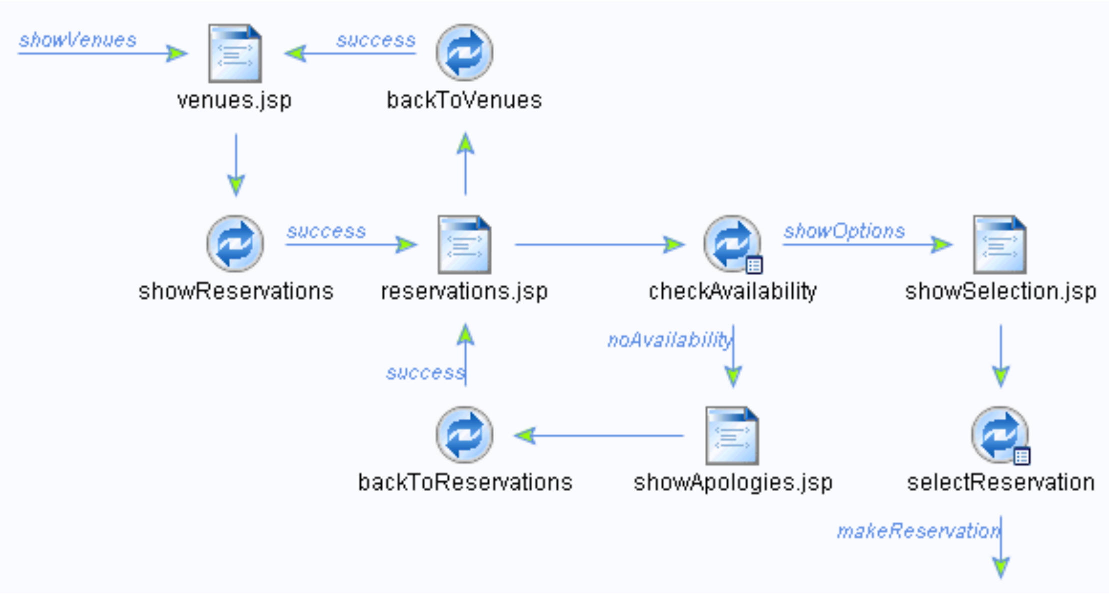
Sample Code

<netui:scriptContainer> Tag Sample

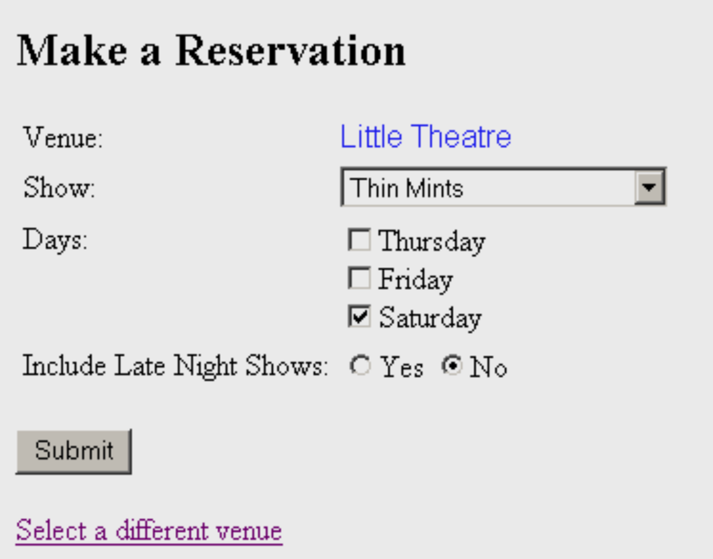
A Detailed Page Flow Example

In WebLogic Workshop it is very easy to build the basic framework of a web application's navigational control and to create JSPs that provides access to form bean properties through a form. This allows you to quickly implement the basic design of a (component of a) web application, and test the business and navigational logic of a site before enhancing JSPs and implementing minor navigational flows. This topic gives a number of pointers on how to quickly set up the basic framework of your web application. If you want to build your own page flow right now, go to Tutorial: Page Flow.

This topic uses, and partially reproduces, a sample web application that implements an online reservation system for theaters. A detail of the Flow View of the web application is shown here:



After selecting a venue on the page *venues.jsp*, you are guided to the *reservations.jsp* page showing the available shows. A detail of this page, which is still under construction, is shown here:



Make a Reservation

Venue: Little Theatre

Show: Thin Mints

Days:
☐ Thursday
☐ Friday
☑ Saturday

Include Late Night Shows: ○ Yes ● No

Select a different venue

The form displays the theater that was selected and allows the user to make various reservation selections. After clicking ***Submit***, the action method checkAvailability is invoked, which checks whether seats are available. If so, the various options are shown in showSelection.jsp. If no available seats could be found that match the user's request, the user is notified on showApologies.jsp.

Building a New Page Flow

There are various ways to go about designing a page flow from scratch. One approach is to use a top−down or forward design in which you follow the flow of events as if you were an actual user, and build new action methods and JSPs as you encounter them in the flow. Another approach, the one that is used here, is to use a data−centric approach to designing a page flow. In the data−centric approach the designer focuses on the data−to−be−exposed and the data handling logic first, and uses that to determine the JSP's user interface.

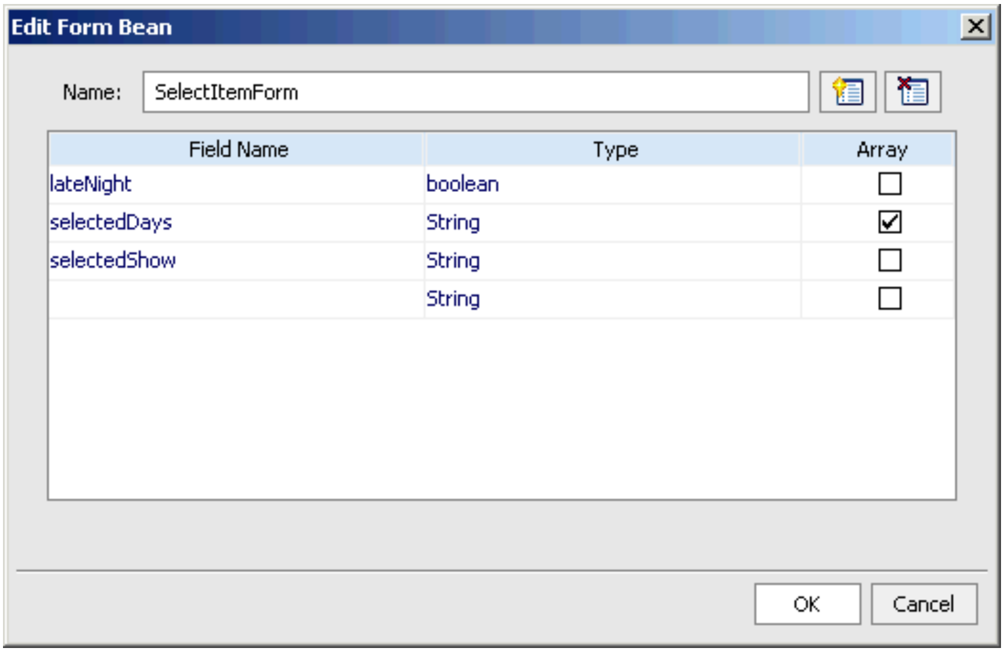
To start building the basic framework, you must have started a new page flow. If you are not sure how to do this, see How Do I: Create a Page Flow? When a page flow is open, WebLogic Workshop displays its Flow View window and a Palette window. You can choose an icon in the Palette and drag the icon onto the Flow View canvas.

Creating an Action Method that Uses a Form Bean

In a data−centric approach, we first concentrate on the data and data handling. For example, in our sample application we might decide to first create the action method checkAvailability, which uses a form bean:

- In the ***Palette*** window, select the Action icon and drag it onto the flow view. A ***New Action*** dialog will appear.
- In the ***Create New*** field, enter *checkAvailability*. Select the ***Create New*** radio button and enter a name for the form bean. Click ***OK***. The icon for the action method appears in the Flow View.
- Select the action method icon and click the Edit Form Bean icon in the ***Form Bean*** window. This is the following icon:

  If you don't see the ***Form Bean*** window, you can make it available by selecting ***Windows*****−−>*****Form Bean*** from the ***View*** menu.
- In the ***Edit Form Bean*** dialog, enter the various form bean properties. In our example, we need the following data:



Name: SelectItemForm

Field Name	Type	Array
lateNight	boolean	☐
selectedDays	String	☑
selectedShow	String	☐
	String	☐

- Click ***OK***.

In only a few steps we have created an action method and a form bean.

Creating an Input Page

To create a JSP that binds to the data in the form bean and calls the action method that we have just created, do this:

- Right−click an action method and select ***Generate Input Page***.

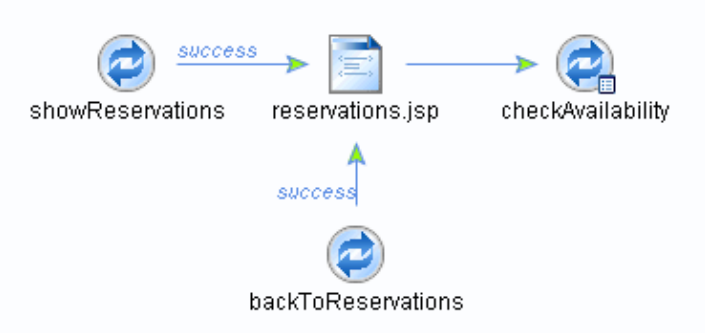
This creates a JSP with a form that calls the action method, and form fields for the various form bean properties. In our example, the form fields generated for *checkAvailability's* form bean properties are not the ones we want to use eventually, but we will worry about this later. However, we will rename the JSP to reservations.jsp.

Creating a Calling Action Method

It is similarly easy to create an action method that calls a JSP:

- Right−click a JSP and select ***Generate Calling Action***.

In our example we generate two calling action methods for reservations.jsp and we name these *showReservations* and *backToReservations*. We now have the following page flow:

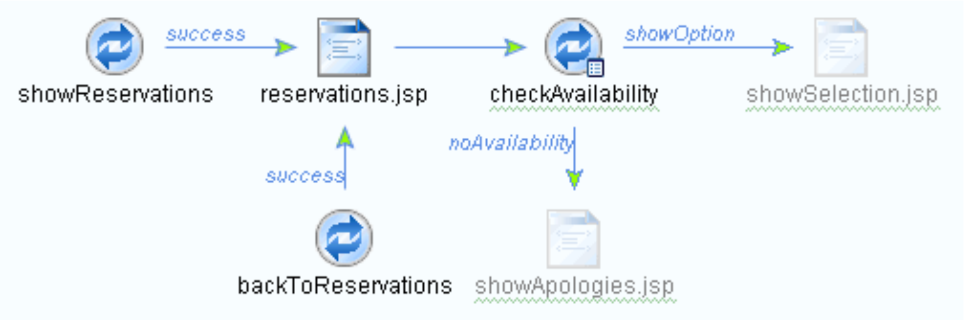


Implementing Data Handling

So far we have created a form bean, a JSP to enter values in the form, and various action methods. Now we need to determine how to process the data the user enters. In the example, we go to the checkAvailability action method in Source View and implement the handling code:

```
/**
 * @jpf:action
 * @jpf:forward name="noAvailability" path="showApologies.jsp"
 * @jpf:forward name="showOption" path="showSelection.jsp"
*/
protected Forward checkAvailability(SelectItemForm form)
{
   SelectReservationForm availableSeatsForm = new SelectReservationForm();
   if(findAvailableReservations(form, availableSeatsForm))
      return new Forward("showOptions", availableSeatsForm);

   else
      return new Forward("noAvailability");
}
```

(To create a running page flow, you will also have to implement the findAvailableReservations method and the SelectReservationsForm form bean in the JSP. We leave this exercise up to you.) If we now go back to Flow View, we see that two ghost JSP icons have appeared:



These JSP names correspond to the path names you have entered in the action method's jpf:forward tag. To turn these into real JSPs, right−click a ghost JSP icon and select ***Create***.

Running Your First Test

Now it's time to test the data handling logic you've implemented. To do so, we first need to add the begin action to our Flow View:

- Selecting the Begin Action icon in the ***Palette*** Window, drag and drop it on the Flow View, and accept the default values in the ***New Action*** dialog.
- In Flow View, draw an arrow from the begin action icon to the showReservations action icon.

Before we start testing our example, we modify the file showApologies.jsp to enable calling the action method backToReservations. To do so:

- Open the file showApologies.jsp in Design View, and drag and drop an anchor tag from the ***Palette*** window.
- Select the ***invoke an action*** radio button.
- Enter some text in the ***Text*** field.
- In the ***Action*** field, select *backToReservations*.

Now you can run the first test. You can enter values in the form fields of reservations.jsp and depending on your data handling logic, you will either see the showSelection.jsp or showApologies.jsp page. In the latter case, you can click the link on that page to go back to the page reservations.jsp.

At this point we have created a sizable piece of the basic framework of the web application. Notice that a large number of design elements have not been implemented at this point. For instance, the JSP showApologies.jsp only has an anchor at this point, but it will need explanatory text and possibly links to other JSPs, for instance to go the venues.JSP page and select a different venue.

You can now continue building the other pieces of the navigational framework, or you can further enhance the business logic and data handling in the currently available flow. The latter will be discussed next.

Enhancing a User Interface

In our example, WebLogic Workshop auto−generated the file reservations.jsp. It created a form that calls the action method checkavailability and contains form fields to access the properties of the associated form bean. This was very helpful for our first test, but it is not what the final form page should look like. For instance, WebLogic Workshop created a checkbox and textboxes to access the form bean properties, but our design incorporates different form fields. Specifically, we want the user to select a show from a dropdown list. The list of actual shows changes constantly and is stored in a database. The design also calls for enabling the user to select the days he or she can attend, and to include attendance to late night shows as a radio button option.

Creating a Dropdown List

To create a dropdown list that displays values from a database, we first need to read the values from a database and add these to one of the available data bindings context. To read the database values, we use the Java control that was created earlier for this purpose, quite possibly by another member of the development team, in our JPF file. (For more information on Java controls, see Working with Java Controls.) The list of possible shows only needs to be available in reservations.jsp, so we 've decided to add these values to the request context in the action method that calls the page, instead of in the pageFlow or globalApp:

```
/**
 * @common:control
*/
private ItemsDBControl items;
private transient Map options = null;
...

protected void initSelections()
{
   ...
   ItemSelectNode[] array = items.getItemSelectData();
   Map map = new LinkedHashMap();
   for(int i = 0; i < array.length; i++)
      map.put(new Integer(array[i].itemNumber), array[i].itemName);
   options = map;
   ...
}

/**
 * @jpf:action
 * @jpf:forward name="success" path="reservations.jsp"
*/
protected Forward showReservations() throws Exception
{
   initSelections();
   getRequest().setAttribute("allShows", options);
   return new Forward( "success");
}
```

Using the Java control, we read the possible shows from the database, put their primary key values and names in a Java Map, and add the map to the request data binding context.

In reservations.jsp, we need to replace the textbox for the show field with a netui:select tag. You can use the ***Table Navigator*** window to easily find the correct cell in the table. In Source View, this tag should be implemented as follows:

```
<netui:select dataSource="{actionForm.selectedShow}" optionsDataSource="{request.allShows}"/>
```

Notice that the dataSource attribute binds the primary key value of the show the user selects to the selectedShow property in the form bean. The optionsDataSource attribute binds to the list of shows and displays the show names as the options in the dropdown list. Instead of using a Java Map, we also could have created an ArrayList of show names. In the latter case, a select option's value and its displayed text will both be the name of a show.

Checkboxes and Radio Buttons

To implement a set of checkboxes, you use a netui:checkBoxGroup JSP tag. To implement a set of radiobuttons, you use a netui:radioButtonGroup tag. For both tags you can use the optionsDataSource attribute to read data from a data binding context, like we saw above for the dropdown list, or you can implement the various options directly in the JSP. In our example we use both approaches:

```
...
<netui:checkBoxGroup defaultValue="{pageFlow.daysOfWeek[0]}"
     dataSource="{actionForm.selectedDays}" optionsDataSource="{pageFlow.daysOfWeek}" />
...
<netui:radioButtonGroup dataSource="{actionForm.lateNight}">
   <netui:radioButtonOption value="1">Yes</netui:radioButtonOption>
   <netui:radioButtonOption value="0">No</netui:radioButtonOption>
</netui:radioButtonGroup>
...
```

The netui:checkBoxGroup JSP tag reads the various days from the ArrayList object daysOfWeek, which is stored in the pageFlow data binding context, and selects the first day in the object by default through the defaultValue attribute. The user's selection is stored in the form bean's selectedDays property.

The netui:radioButtonGroup tag uses netui:radioButtonOption tags to implement the various radio buttons. The user selection, that is, the value corresponding to the netui:radioButtonOption tag that the user selected, is stored in the form bean property lateNight referenced in the dataSource attribute of the netui:radioButtonGroup tag.

Although we have now implemented additional major pieces of functionality in the reservations.jsp page, there is still more work to be done. Minor navigational flows, for instance to cancel the reservation selection process and go somewhere else in the site, might still have to be implemented. Also, form validation of the user input is probably desirable to create a more robust system. Finally, the JSP is going to require further work to, for instance, show the selected venue and to present all this information in a visually appealing manner, aspects that might be implemented by another member of the development team. However, in relatively little time and with little effort we have created a running web application framework that can be used to test our design and allows us to make modifications to the web application's navigational and other business logic early in the software development cycle.

Related Topics

Best Practices for Page Flow Applications

Getting Started: Web Applications

Tutorial: Page Flow

Page Flow and JSP Samples

How Do I: Page Flows and JSPs

Best Practices for Page Flow Applications

This topic presents best practices for creating page flow web applications, describing tip and tricks, and ways to avoid problems.

- Location of Business Logic
- Form Validation
- Template JSPs, Contents JSPs, and Relative versus Absolute URLs
- Security
- Avoid Creating a Page Flow with the Same Name as Another Entity

- Using Read−Only Actions, Where Appropriate

Location of Business Logic

Business logic includes both the navigation control for page flows and the implementation of other business processes such as data handling. In general it is preferable to put the navigation logic in the page flow's controller (JPF) file. Data handling and other business process implementations should in principle be implemented using the other WebLogic workshop components such as built−in and custom controls, web services, and regular Java classes. These components are called by the page flow's controller file, and the results of this interaction is returned to a JSP. For more information, see Working with Java Controls and Building Web Services.

Form Validation

When you validate user input through a form on a JSP, there are two ways to accomplish this:

- Client−side validation. For instance, you add JavaScript to the onBlur or onChange method of a <netui:textBox> or to the onClick method of the <netui:button> tag. When an error is found, it can be reported to the user in a dialog.
- Server−side validation using the validate method or the Struts ValidatorPlugIn. When a form is submitted, the data is checked on the server. When an error is found, an error message can appear somewhere on the form page itself, for instance next to the affected field, or the user might be sent to a new page displaying the error messages, depending on the implementation details of the server−side validation.

Both approaches have their own strengths and weaknesses. A drawback of client−side validation is that the JavaScript is embedded directly in the JSP instead of stored separately from the user interface in a JPF or Java file. Another problem is that you have little control over the browser side. The user might actively attempt to bypass client−side controls, or use a (possibly outdated) browser with features that conflict with your validation code. In contrast, server−side validation is fully under your control.

A drawback of the server−side validation is that all the form data needs to be routed to the server and errors need to be passed back. If this is a concern, you might consider using both client−side and server−side validation. Use client−side script to catch obvious errors, and use server−side logic to do a thorough validation.

For more information about both approaches, see Validating User Input.

Template JSPs, Contents JSPs, and Relative versus Absolute URLs

In a Web project, a template page is a JSP that defines the overall look and feel of a set of pages. It does this by providing an overall layout structure, style, and design of the page, and defines placeholders for content. Note that when you use relative URIs inside a template with the <netui:base/> tag, the URI is interpreted as relative to the content JSP page that uses the template, and not as relative to the location of the template. If you use a template to ensure that a certain element is included on every web page, for instance a gif containing the company logo, you should use an absolute URL.

Recommendations for using templates:

- Templates need to use absolute URLs if the template is used to include a standard element.
- You should be careful about using JSP tags from the netui−tags−* libraries in templates. These tags may contain "actions," but those actions are relative to the content page's page flow. This means that each page flow that uses a template would have to implement the action, or the action must be defined in the GlobalApp.
- For some netui* tags where you must use them in a template, using the url= attribute instead of the action= attribute is best. For example, the <netui:anchor> tag should use the url attribute.
- If the template contains the <html> or <netui:html/> tag, you probably should also include the <netui:base/> tag. This ensures that the relative URL on the content pages are based correctly.

Recommendations for content JSPs:

- Use relative URLs when possible.
- When you are not able to use relative URLs:
  - ♦ Use request.getContextPath() to prefix absolute URLs when addressing items within the same web application.
  - ♦ Use http://hostname/appname/ to prefix absolute URLs when addressing items outside of the web application.

Security

For most web applications, security is a vital issue to protect the confidentiality and integrity of data. When designing a secure web application, you should considering the following issues:

- WebLogic Workshop comes with a security model that enables you to use secure transport protocols, and allows you to implement overall access to the web application, and fine−grained security to components of the web application, on the basis of a user's security role. For more information, see Security.
- When adding data to a data binding context, consider the scope of the data use. It is a good design practice to limit data availability to those action methods, page flows, and JSPs that require this data. This will help safequard unintended access or even modification of the data. For instance, if you only need data for a particular JSP, consider using request instead of pageFlow. Also notice that passing data as attributes on a URL is especially unsafe, as it allows for easy modification and can even enable undesired access to other data by a malintended user. For more information, see Lifetime and Scability of Data Binding Contexts.
- Consider using further restrictions to limit access. A frequently used restriction is the notion of

timeouts, where a user session is disabled when a certain amount of idle time is passed. You can for instance use client−side JavaScript to enable timeouts − consider adding this to the template page(s) you use to build your JSPs − and/or you can implement server−side timeouts, for instance by restricting the time before a session times out. For more information on session timeouts, see the session data binding section in Using Data Binding in Page Flows.

Avoid Creating a Page Flow with the Same Name as Another Entity

The Page Flow Wizard and Database Control Wizard allow you to create a new directory with the same name as an existing Java−based file. Using identical names results in packaging conflicts, which cause parse errors in .jws, .jcs, .jcx, and .java files. Avoid naming a directory and a peer Java−based file the same name. If you do this inadvertently, simply rename one of the entities.

Using Read−Only Actions, Where Appropriate

You may be able to improve the performance of your page flow webapps in WebLogic cluster environments. The @jpf:action annotation provides an optional attribute, read−only={" true | false " }. The default is read−only="false". You can designate read−only="true" to indicate your intention that this action ***will not*** update any member data in the page flow. In a cluster environment, this designation causes WebLogic Workshop to skip any attempted failover of the page flow after the action is run. This makes it unnecessary for WebLogic Workshop to communicate portions of this page flow's state data across the nodes in the cluster that pertain to the read−only actions, which may improve performance. Note however that if the action marked as "read−only" calls, or is called by, other actions that are read/write, WebLogic Workshop must replicate the state of the actions across nodes in the cluster. Therefore you should only use the read−only="true" option if you know that the action and any directly related actions will not update member data.

Related Topics

Getting Started with Page Flows

Page Flow and JSP Reference

Updating Libraries with Service Packs

After you install a BEA service pack that includes library updates, you must update the libraries in page flow ***web projects*** that you developed with an earlier version of WebLogic Workshop. For example, WebLogic Workshop 8.1 Service Pack 2 contains page flow library updates that were made since the initial 8.1 release.

To update the libraries for your page flow web projects:

1. In the WebLogic Workshop IDE, open the application you want to update.
2. In the Application pane, right−click the web project's root directory and choose ***Install > Web Project Libraries...***.
3. On the Install dialog window, the Libraries box should be checked. Also look at the selections in the Resources category. By default, the WEB−INF/validation*.* files may be checked, but the other resources are unchecked. It is important that you not allow the Install operation to overwrite resource files that you customized for your project, such as the root−level Controller.jpf and index.jsp files.
4. When you are ready, click the Install button.
5. After the updates have been made, click the Close button.
6. Repeat this process for other web projects in applications that you developed.

Configuring Page Flow Applications

The following topic explains how to prevent exhaustion of resources in your page flow because of (1) large numbers of server forwards and (2) large nesting stacks.

Limiting the Number of Server Forwards

The maximum number of server forwards allowed within a single request can be configured in the file WEB−INF/web.xml using the jpf−forward−overflow−count parameter.

```
<context−param>
    <param−name>jpf−forward−overflow−count</param−name>
    <param−value>10</param−value>
</context−param>
```

The <context−param> element should be a direct child of the <web−app> element.

If the number of server forwards exceeds the given count, an error is written to the response and no further forwarding is excuted. This is mainly to prevent infinite loops of server forwards. To reproduce the error, invoke this action in a Page Flow:

```
/**
 * @jpf:action
 * @jpf:forward name="self" path="overflow.do"
 */
public Forward overflow()
{
    return new Forward( "self" );
}
```

If the jpf−forward−overflow−count parameter is ommitted from the web.xml file, the error will be written to the response after 50 server forwards within a single request.

Limiting the Number of Nested Page Flows

The maximum depth of the Page Flow nesting stack can be configured in WEB−INF/web.xml using the jpf−nesting−overflow−count parameter.

```
<context−param>
    <param−name>jpf−nesting−overflow−count</param−name>
    <param−value>10</param−value>
</context−param>
```

The <context−param> element should be a direct child of the <web−app> element.

This parameter set the maximum size of the Page Flow nesting stack. If Page Flow are repeatedly nested until the stack exceeds the specified value, an error is written to the response object and any further nesting is not allowed. This helps prevent the nesting stack from consuming large amounts of resources.

Exception Handling and Validating User Input

The following topics explain how to handle errors that arise in a web application and how to validate data submitted by users.

Topics included in this section

Handling Exceptions in Page Flows

Explains how you can catch and handle a standard set of exception types that are defined by WebLogic Workshop for page flow applications. Also covers the global page flow, Global.app, that can be used for exception handling.

Validating User Input

Discusses client−side and server−side validation techniques to check user information entered in a form.

Related Topics

Using JavaScript in Page Flow and Portal Applications

Validating User Input

In most cases when a user enters data through a form, we want to ensure that the values entered are valid before the data is, for instance, stored in a database or used to determine the next action. Form validation is frequently done using client−side JavaScript. This approach is also possible in WebLogic Workshop. In addition, server−side validation checks the user input on the server side. The advantages of the latter approach:

- You do not have to rely on the features of the browser, over which you have no control.
- You can encapsulate the validation logic with the other business logic in the controller file.
- Error messages can be displayed right in the form next to the affected form field, instead of in popup dialogs.

Client−Side Validation

WebLogic Workshop provides a number of attributes on the <netui:form> tag and its form fields, such as onClick and onBlur, which can be used to invoke JavaScript. For example, you can have a <netui:textBox> tag in which a user enters his age:

```
<netui:textBox dataSource="{actionForm.age}" onBlur="isValidAge()"/>
```

This is an example of the JavaScript function isValidAge which is mentioned in the onBlur attribute:

```
function isValidAge() {
    if(document.forms[0].elements[1].value <= 0)
        alert('Wrong Age');
}
```

After the user has entered a value and goes to the next form field, the onBlur attribute calls the JavaScript function isValidAge, which will display a dialog when the age is not considered valid.

Notice that in the example the form and its fields are accessed by using JavaScript forms and elements arrays. If you would like to use the id attributes to access the form and its fields, you need to (1) specify the tagID attributes on the <netui:form> and its input elements. Then use the JavaScript function getNetuiTagName(tagId, this). Also, you will need to ensure that the <html> tag is replaced with <netui:html> on the JSP page. Rewriting the previous example, the code will be:

```
<netui:html>
     ...
     <netui:form tagId="myForm" >
         <netui:textBox tagId="age" dataSource="{actionForm.age}" onBlur="isValidAge();"/>
     </netui:form>
     ...
</netui:html>
```

And this would be the JavaScript function isValidAge:

```
function isValidAge() {
    if(document[getNetuiTagName("myForm",this)][getNetuiTagName("age",this)].value <= 0)
        alert('Wrong Age');
}
```

Notice that instead of using elements[1], we refer to the element's tagID using the function getNetuiTagName(tagId, this).

These approaches also work in conjunction with a submit action of the form. For <netui:button> and <netui:imageButton> tags, you can invoke the JavaScript function through the tags's onClick attribute, as is shown in the following example:

```
<netui:button value="submit" onClick="return isValidForm();" />
```

When a user clicks the button, a JavaScript function isValidForm checks the form fields and returns true if validation is successful. Or the isValidForm function displays a dialog with error messages and returns false if not all entered values are correct. (The implementation of this function is not actually shown in the sample code.)

For <netui:anchor> and <netui:imageAnchor> tags, whose implementation rely on JavaScript to submit a form, your JavaScript validation function must contain additional code to set the method and action properties of the JavaScript Form object, and call its submit method if validation is successful. The following example, which uses the same tagIDs and JavaScript function isValidForm as the previous example, demonstrates how to set the various properties and invoke the submit method when form validation is successful:

```
function submitFromAnchor() {
    document[getNetuiTagName("myForm",this)].method="POST";
    document[getNetuiTagName("myForm",this)].action="/ClientSideValidation/thankYou.do";
    if(isValidForm())
        document[getNetuiTagName("myForm",this)].submit();
}
```

As shown in the example, the form's method property must be set to *POST*. The action property must be set to the full name of the action to be called when the user clicks the anchor and submits the form. This action name corresponds to the web application name followed by the name of the action method in the JPF file, followed by a DO extension. In the example, the page flow JPF file of the web application *ClientSideValidation* contains an action method thankYou.

The same function could have been written without the use of getNetuiTagName(tagId, this), as shown in the following example:

```
function submitFromAnchor() {
    document.forms[0].method="POST";
    document.forms[0].action="/ClientSideValidation/thankYou.do";
    if(isValidForm())
         document.forms[0].submit();
}
```

However, if your application resides in a Portal application, it is recommended that you use getNetuiTagName(tagId, this) to retrieve page elements, since Portal containers may rewrite the id attributes of page elements. For details see Using JavaScript in Page Flow and Portal Applications.

Finally, the <netui:anchor> or <netui:imageAnchor> tags must call the JavaScript function in the onClick attribute and add "return false;" as is shown in the following example:

```
<netui:anchor onClick="submitFromAnchor(); return false;" action="thankYou">Submit</netui:ancho
```

If you do not add the expression "return false;", the form will always be submitted and no data will be posted to the form bean.

For more information on using client−side JavaScript see Using JavaScript in Page Flow and Portal Applications.

Server−Side Validation

WebLogic Workshop offers two ways to accomplish server−side validation, one using Java to implement the validate method in the form bean, and the other using the Struts ValidatorPlugIn to do XML−based validation. These examples are shown in the WebLogic Workshop sample application, and are described here. For the sample code, start in the following installed location:

<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/...

Java−Based Validation

To validate user input using the form bean's validate method, follow the steps outlined in this list. Note that these examples are from the following sample page flow, and associated files:

<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/basic

- Add a message−resources tag to your controller class in the JPF File, for example:

  ```
  /**
   * @jpf:controller nested="true"
   * @jpf:message-resources resources="validation.basic.Messages"
   *
   */
  public class Controller extends PageFlowController
  {
  ...
  ```

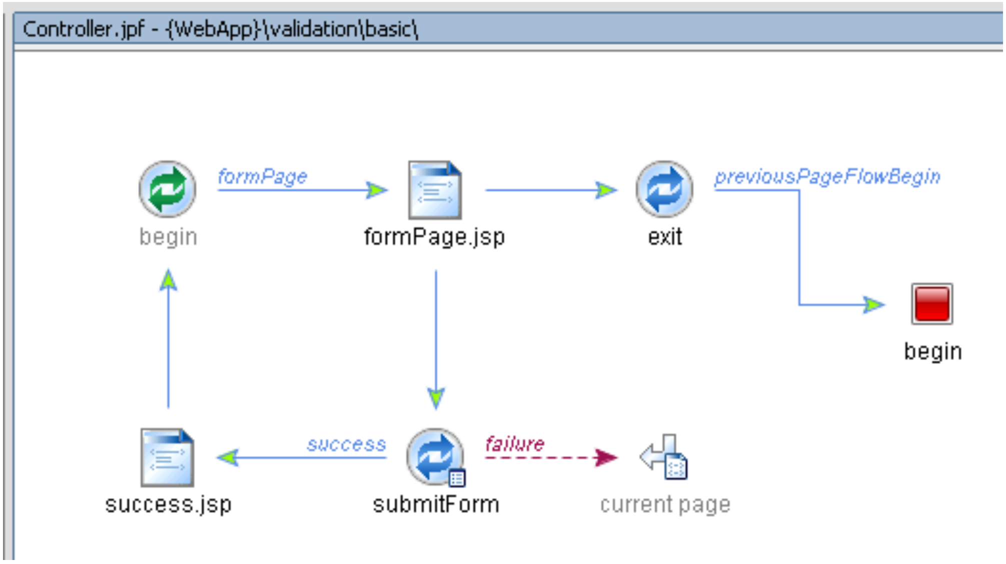
  The resources file is located under the web project's /WEB−INF/classes/... folder and has a *properties* extension. In the example above the Messages.properties file is located in /WEB−INF/classes/validation/basic. The contents of this file are described below.

  Note: In the example, notice the corresponding naming requirements in use here. If the messages resource file is <project−root>/WEB−INF/classes/validation/basic/Messages.properties, then the way to reference this file in the <project−root>/validation/basic/*.jpf page flow annotation is @jpf:message−resources resources="validation.basic.Messages".
- Add a @jpf:validation−error−forward annotation to the action that uses validation. For example:

  ```
   /**
    * @jpf:action
    * @jpf:forward name="success" path="success.jsp"
    * @jpf:validation-error-forward name="failure" return-to="currentPage"
    */
    public Forward submitForm( Form form )
    {
       return new Forward( "success" );
    }
  ```

This annotation provides a flexible mechanism to indicate which page should be loaded, or which action should be run, if a form validation error occurs as a result of running the annotated action. For details see the @jpf:validation−error−forward Annotation topic.

The following Flow View screen illustrates the relationship of the actions and pages in this sample page flow.



- Add the validate method to your form bean class. This method must have the exact signature as given in the example:

```
public ActionErrors validate( ActionMapping mapping, HttpServletRequest request )
{
    ActionErrors errs = new ActionErrors();

    int at = _email.indexOf( '@' );
    int dot = _email.lastIndexOf( '.' );

    if ( at == -1 || at == 0 || dot == -1 || at > dot )
    {
       errs.add( "email", new ActionError( "badEmail" ) );
    }

    if ( _zipCode.length() != 5 )
    {
       errs.add( "zipCode", new ActionError( "badZip", new Integer( 5 ) ) );
    }

    return errs;
}
```

In the validate method you implement the validation logic for the various form fields. If there is an error, you add a new action error. In the example, zipCode refers to a <netui:error> tag in your JSP (<netui:error value="zipCode"/>), and badZip refers to the message key in the Messages.properties file. Note that the new Integer(5) is an object that is the first replacement in the error string. For example, later in this description you will see that we have a message string: "The zip code has to be exactly {0} characters." In this case, the {0} will be replaced by the Integer(5) object. ActionErrors supports constructors with up to four additional replacement objects.

For the full example, see the page flow controller class at <WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/basic/Controller.jpf.

- In the JSP file that displays errors (typically the form page), add the <netui:error> tags to display error messages. For example:

```
    <netui:form action="submitForm">
        <table>
            <tr>
                <td><p>Enter E−mail Address:</p></td>
                <td>
                    <netui:textBox dataSource="{actionForm.email}"/>
                </td>
                <td>
                    <netui:error value="email"/>
                </td>
            </tr>

            <tr>
                <td><p>Enter 5−digit Zip Code:</p></td>
                <td>
                    <netui:textBox dataSource="{actionForm.zipCode}"/>
                </td>
                <td>
                    <netui:error value="zipCode"/>
                </td>
            </tr>
        </table>

        <netui:button>Submit</netui:button>
        <netui:button action="exit">Exit</netui:button>
    </netui:form>
```

In the example an error will be displayed next to the relevant form field. For the full example, see <WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/basic/formPage.jsp.

- In addition, or as an alternative, you can add a <netui:errors> tag to your JSP (<netui:errors/>), for instance at the bottom of your JSP. This tag will display a summary of all the errors that were found.
- The resources file contains the actual error message plus any special html formatting you might want to apply when displaying the errors in the JSP. Remember that this file is located under /WEB−INF/classes/... and has a *properties* extension. In this example, the Messages.properties file is located in /WEB−INF/classes/validation/basic.

```
badEmail=Bad email address.
badZip=The zip code has to be exactly {0} characters.

errors.header=<br><hr><font color="Blue">List of errors, using the &lt;netui:error&gt; ta
errors.prefix=<li>
errors.suffix=
errors.footer=</ul></font>

error.prefix=<font color="Red">
error.suffix=</font>
```

Notice that in the example, badZip is the message key for the error message that needs to be displayed when an incorrect zipCode was entered. The other lines in the file create formatting for the <netui:error> and <netui:errors> tags, such that individual error messages are displayed in red and the error summary is displayed in blue as a list, preceded by the header "List of errors, using the

<netui:error> tag:".

XML−Based Validation

To validate user input using the Struts ValidatorPlugIn, follow the steps outlined in this list. Note that these examples are from the following sample page flow, and associated files:

<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/validator

- Add message−resources and struts−merge annotations to your controller class in the JPF File. For example:

  ```
  /**
   * @jpf:controller nested="true" struts−merge="/WEB−INF/strutsValidator−merge−config.xml
   * @jpf:message−resources resources="validation.validator.Messages"
   *
   */
  public class Controller extends PageFlowController
  {
  ...
  ```

  The @jpf:controller annotation's struts−merge attribute identifies the Struts merge file we will use. This Struts merge file will point to the Struts ValidatorPlugIn and its rules. The merge file is described later in this list.

  Also, the resources file is located under the web project's /WEB−INF/classes/... folder and has a *properties* extension. In the example above the Messages.properties file is located in /WEB−INF/classes/validation/validator. Notice the corresponding naming requirements in use here. If the messages resource file is <project−root>/WEB−INF/classes/validation/validator/Messages.properties, then the way to reference this file in the <project−root>/validation/validator/*.jpf page flow annotation is @jpf:message−resources resources="validation.validator.Messages".

- Add a @jpf:validation−error−forward annotation to the action that uses validation. For example:

  ```
  /**
   * @jpf:action
   * @jpf:validation−error−forward name="failure" return−to="currentPage"
   * @jpf:forward name="success" path="success.jsp"
   */
  public Forward submitForm( MyForm form )
  {
      return new Forward( "success" );
  }
  ```

  This annotation provides a flexible mechanism to indicate which page should be loaded, or which action should be run, if a form validation error occurs as a result of running the annotated action. For details see the @jpf:validation−error−forward Annotation topic.
- In the page flow, modify your form bean definition to extend Struts' ValidatorForm interface:

  ```
   public static class MyForm extends org.apache.struts.validator.ValidatorForm
   {
   ...
  ```

For the full example of this form bean class, see <WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/validator/Controller.jpf.

- In the JSP file that displays errors (typically the form page), add the <netui:error> tags to display error messages:

```
<netui:form action="submitForm">
    <table>
        <tr>
            <td><p>Enter E−mail address:</p></td>
            <td>
                <netui:textBox dataSource="{actionForm.email}"/>
            </td>
            <td>
                <netui:error value="email"/>
            </td>
        </tr>

        <tr>
            <td><p>Enter Age:</p></td>
            <td>
                <netui:textBox dataSource="{actionForm.age}"/>
            </td>
            <td>
            <netui:error value="age"/>
            </td>
        </tr>
    </table>

    <netui:button>Submit</netui:button>
    <netui:button action="exit">Exit</netui:button>
</netui:form>
```

In the example an error will be displayed next to the relevant form field. For the full example, see <WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/validation/validator/formPage.jsp.

- In addition, or as an alternative, you can add a <netui:errors> tag to your JSP (<netui:errors/>), for instance at the bottom of your JSP. This tag will display a listing of all the errors that were found.
- Define the struts−merge file to point to Struts' default pluggable validator definition file *validator−rules.xml* as well as the XML file containing the actual validation rules for this particular form. In our example, the merge file strutsValidator−merge−config.xml is defined as follows. You can find this merge file in:

<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/WEB−INF

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE struts−config PUBLIC "−//Apache Software Foundation//DTD Struts Configuration
  "http://jakarta.apache.org/struts/dtds/struts−config_1_1.dtd">

<struts−config>
   <form−beans/>
   <global−exceptions/>
   <global−forwards/>
   <action−mappings>
   </action−mappings>

 <plug−in className="org.apache.struts.validator.ValidatorPlugIn">
   <set−property property="pathnames"
   value="/WEB−INF/validator−rules.xml,/WEB−INF/strutsValidator−validation.xml"/>
 </plug-in>

</struts-config>
```

The default pluggable validator definitions are contained in the file validator−rules.xml, which is located in WEB−INF. The actual form validation rules should be defined in a separate file. In the example the file is called strutsValidator−validation.xml and is also located in /WEB−INF. This file is described next.

- In the file strutsValidator−validation.xml, you implement the actual form validation using the default validator definitions. For example:

```
<form−validation>
    <formset>
        <form name="myForm">
            <field property="email" depends="required">
                <arg0 key="email.displayName"/>
            </field>
            <field property="age" depends="required,integer">
                <arg0 key="age.displayName"/>
            </field>
        </form>
    </formset>
</form−validation>
```

- The resources file contains the actual error message plus any special html formatting you might want to apply when displaying the errors in the JSP file. Remember that this file is located under WEB−INF/classes/... and has a properties extension. In this example, the Messages.properties file is located in /WEB−INF/classes/validation/validator.

```
errors.header=<br><hr><font color="Blue">List of errors, using the &lt;netui:error&gt; ta
errors.prefix=<li>
errors.suffix=
errors.footer=</ul></font>

error.prefix=<font color="Red">
error.suffix=</font>

errors.required={0} is required.
errors.integer={0} must be an integer.

email.displayName=The email address
age.displayName=The age
```

Notice that in the example, email.displayName describes the name of a form field, whereas errors.required describes the error message for missing but required form field entries. If a user does not type in an e−mail address, the resulting error message will be "The email address is required". The other lines in the file create formatting for the <netui:error> and <netui:errors> tags, such that individual error messages are displayed in red and the error summary is displayed in blue as a list, preceded by the header "List of errors, using <netui:error> tag:".

Related Topics

@jpf:validation−error−forward Annotation

@jpf:message−resources Annotation

Validating User Data with Struts Sample

Merging Struts Artifacts Into Page Flows

Using JavaScript in Page Flow and Portal Applications

Handling Exceptions in Page Flows

WebLogic Workshop provides a number of standard page flow exception types that you can catch and handle in your web applications. The types represent common exception scenarios for page flows. The following table describes the exceptions:

Exception Type	***Description***
`ActionNotFoundException`	Occurs when the user tries to execute an action that does not exist on the page flow.
`EmptyNestingStackException`	Occurs when the user invokes an action in a nested page flow that is qualified with a @jpf:forward return−action="<action−name−in−calling−pageflow>" annotation, but there is no calling page flow. This can happen in iterative development mode when you have modified files and caused the web application to be redeployed, or when the session expires.
`IllegalOutputFormTypeException`	Occurs when the first output form for a Forward resolves to a @jpf:forward annotation whose return−form or return−form−type attribute demands a different form type.
`IllegalPageInputException`	Occurs when a page input has been added to a Forward that resolves to a @jpf:forward redirect="true" annotation. Page inputs may not be used on redirect forwards.
`IllegalRedirectOutputFormException`	Occurs when an output form has been added to a Forward that resolves to a @jpf:forward redirect="true" annotation. Output forms may not be used on redirect forwards.
`InfiniteReturnToActionException`	Occurs when the user invokes an action that is qualified with a @jpf:forward return−to="previousAction" annotation, but the previous action was the same as the current action (an infinite loop).
`LoginExpiredException`	Occurs when the NotLoggedInException would be thrown, and the current HttpServletRequest refers to a session that no longer exists. For more information, see Handling Sessions Expirations and Timeouts in this topic.
`NoCurrentPageFlowException`	Occurs when the user invokes an action that is qualified with a @jpf:forward annotation marked with either return−to="previousAction" or return−to="previousPage", but there is no current page flow. This happens in iterative development mode when you have modified files and caused the web application to be redeployed, or when the session expires.
`NoMatchingActionMethodException`	Occurs when the current action method does not accept the type of form passed in the Forward to the action.

	This may happen when the user returned to the calling page flow, from a nested page flow, with a specified form (@jpf:forward return−form="<form−name>" or return−form−type="<form−type>", but no action in the calling page flow accepts that form type. This exception can also occur when you simply forward to another page flow, nested or not, whose begin action (or any other specified action that was the forward target) does not accept the type of form passed in the Forward object.
`NoPreviousActionException`	Occurs when the user attempts to execute an action marked with the @jpf:forward return−to="previousAction" annotation, but there is no previously run action. For example, this could happen if the begin action returns a @jpf:forward marked with return−to="previousAction".
`NoPreviousPageException`	Occurs when the user attempts to execute an action marked with the @jpf:forward return−to="previousPage" annotation, but there is no previous page in the page flow. For example, this could happen if the begin action returns a @jpf:forward marked with return−to="currentPage".
`NotLoggedInException`	Occurs when the user attempts to execute an action marked with the login−required or roles−allowed attributes (@jpf:controller or @jpf:action annotations), but the user in not logged in. If the requested session−ID is different than the current session−ID, the LoginExpiredException will be thrown instead of the NotLoggedInException. For more information, see Handling Sessions Expirations and Timeouts in this topic.
`UnfulfilledRolesException`	Occurs when the user attempts to execute an action marked with roles−allowed, but the user is logged into an account that is not associated with any of the required roles.
`UnresolvableForwardException`	Occurs when the user returns a Forward from an action method, but there is no forward with that name on the action method, or at the class level.

Note: For information about the annotations and attributes mentioned in the previous table, see the Related Topics section below.

These exceptions extend com.bea.wlw.netui.pageflow.PageFlowException. If you do nothing to handle these exceptions (with a @jpf:catch annotation), the default error message will be written to the response.

You can handle these exceptions specifically by putting the following annotation at the class level in Global.app, or at the class level of a page flow, or on an action method in a page flow. The @jpf:catch annotation provides a declarative way to catch unhandled exceptions by forwarding to an error page.

```
* @jpf:catch type="com.bea.wlw.netui.pageflow.PageFlowException"
method="..." | path="..."
```

When you create a new "Web Project" project with WebLogic Workshop, a default Global.app file is created for you in the project's /WEB−INF/src/global directory. The Global.app file allows you to define actions that can be invoked by any page flow in the web project.

Note: In a page flow controller class that you create with the Page Flow Wizard, you may notice the following line:

```
// protected global.Global globalApp;
```

It is ***not*** required that you uncomment this line in order to have access to methods in Global.app. However, if you do uncomment it, this globalApp declaration will give the page flow easy access to public methods and variables in Global.app.

The default Global.app class definition includes the following annotations:

```
    /**
     * @jpf:catch type="Exception" method="handleException"
     * @jpf:catch type="PageFlowException" method="handlePageFlowException"
     */
     public class Global extends GlobalApp
```

The generic, "last resort" handleException method is defined for you in /WEB−INF/src/global/Global.app, as follows. It results in forwarding the user to the /error.jsp page, which is also provided for all new "Web Projects".

```
    /**
     * @jpf:exception−handler
     * @jpf:forward name="errorPage" path="/error.jsp"
     */
    protected Forward handleException( Exception ex, String actionName,
                                       String message, FormData form )
    {
        System.err.print( "[" + getRequest().getContextPath() + "] " );
        System.err.println( "Unhandled exception caught in Global.app:" );
        ex.printStackTrace();
        return new Forward( "errorPage" );
    }
```

You can modify the generated /error.jsp page to match the requirements of your web project. It contains code similar to the following:

```
<!−−Generated by WebLogic Workshop−−>
<%@ page language="java" contentType="text/html;charset=UTF−8" isErrorPage="true"%>
<%@ 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>Error</title>
  </head>
  <body>
    <p>
      An error has occurred:
    </p>
    <blockquote>
      <netui:label value="{request.errorMessage}" defaultValue="" />
      <br/>
      <netui:exceptions showMessage="true" />
    </blockquote>
  </body>
</netui:html>
<!-- Some browsers will not display this page unless the response status code is 200. -->
<% response.setStatus(200); %>
```

You can test that your /error.jsp page is working the way you intended by creating another JSP page, starting the server, and then opening that test page. For example in your web project's root directory, you could add a test page named /testError.jsp that contains the following code.

```
 <%@ 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>Test Page to Verify Error.jsp</title>
   </head>
   <body>
    <p>Using this page to verify that a deliberate exception causes
       our /error.jsp to load:</p>
    <p>
     <% if ( true ) throw new Exception( "Can you see this message?" ); %>
    </p>
   </body>
</netui:html>
```

The code shown above throws an exception and should result in the loading of your error page. With the server running, access the test page. For example:

http://localhost:7001/myApp/testError.jsp

For the page flow exceptions described in the previous table, the default Global.app provides the handlePageFlowException method:

```
    /**
     *
     * @jpf:exception-handler
     */
    public Forward handlePageFlowException( PageFlowException ex, String message,
                                            String action, FormData form )
        throws java.io.IOException
    {
        ex.sendError( getRequest(), getResponse() );
        return null;
    }
```

The handlePageFlowException handler allows page flow exceptions to write informative error pages to the response. To use the standard exception−handler for these exceptions (handleException), remove in

Global.app the handlePageFlowException method and the @jpf:catch annotation for PageFlowException.

Handling Session Expirations and Timeouts

Session expirations or timeouts can occur in your web application and should be handled. As mentioned in the prior table, if the requested session−ID is different than the current session−ID, WebLogic Workshop will issue LoginExpiredException (com.bea.wlw.netui.pageflow.LoginExpiredException) instead of the NotLoggedInException. You can handle session−expiration specifically (using a @jpf:catch annotation for LoginExpiredException). However, LoginExpiredException extends NotLoggedInException, so if you prefer you can safely catch only NotLoggedInException.

The following code shows a way to handle a session timeout in a login example.

```
/**
 * @jpf:action
 * @jpf:forward name="loginPage" path="Login.jsp" redirect="true"
 * @jpf:forward name="loginTimeoutPage" path="LoginTimeout.jsp" redirect="true"
 */
 protected Forward begin()
 {
    String requestedSessionID = getRequest().getRequestedSessionId();

    if ( requestedSessionID != null && ! requestedSessionID.equals( getSession().getId() )
    {
       return new Forward( "loginTimeoutPage" );
    }
    else
    {
       return new Forward( "loginPage" );
    }
 }
```

For information on setting session timeouts, see the "Session" section of Using Data Binding in Page Flows.

Related Topics

@jpf:catch Annotation

@jpf:exception−handler Annotation

@jpf:validation−error−forward Annotation

Working with Struts Applications

There are three approaches to using Struts components in Page Flow and Portal applications. You can

1. use your Struts components directly in Page Flow applications,
2. import simple Struts applications into new Page Flow applications, or
3. create portlets directly from your Struts applications.

The first approach uses existing Struts based components without modifying them. This allows your existing Struts components to interact with Page Flow components. You can leverage your existing JSP Pages, Action Classes, Forms Beans and flow configurations. Documentation can be found at Interoperating With Struts and Page Flows. Samples can be found at Struts Interoperation Sample.

The second approach converts simple Struts applications into Page Flow applications. This approach has several restrictions and may not be suitable complex Struts applications. But for simple Struts applications this can be a fast way to move them to the Page Flow framework. Documentation can be found at Merging Struts Artifacts Into Page Flows. Samples can be found at Struts Merge Sample.

The third approach uses the portlet wizard to create portlet definitions based on your existing Struts Modules. Point to a Struts application's configuration file and the Portlet wizard will bring the application's functionality into a Portlet. Documentation can be found at Building Struts Portlets.

Related Topics

Interoperating With Struts and Page Flows

Merging Struts Artifacts Into Page Flows

Building Struts Portlets

Samples

Struts Interoperation Sample

Struts Merge Sample

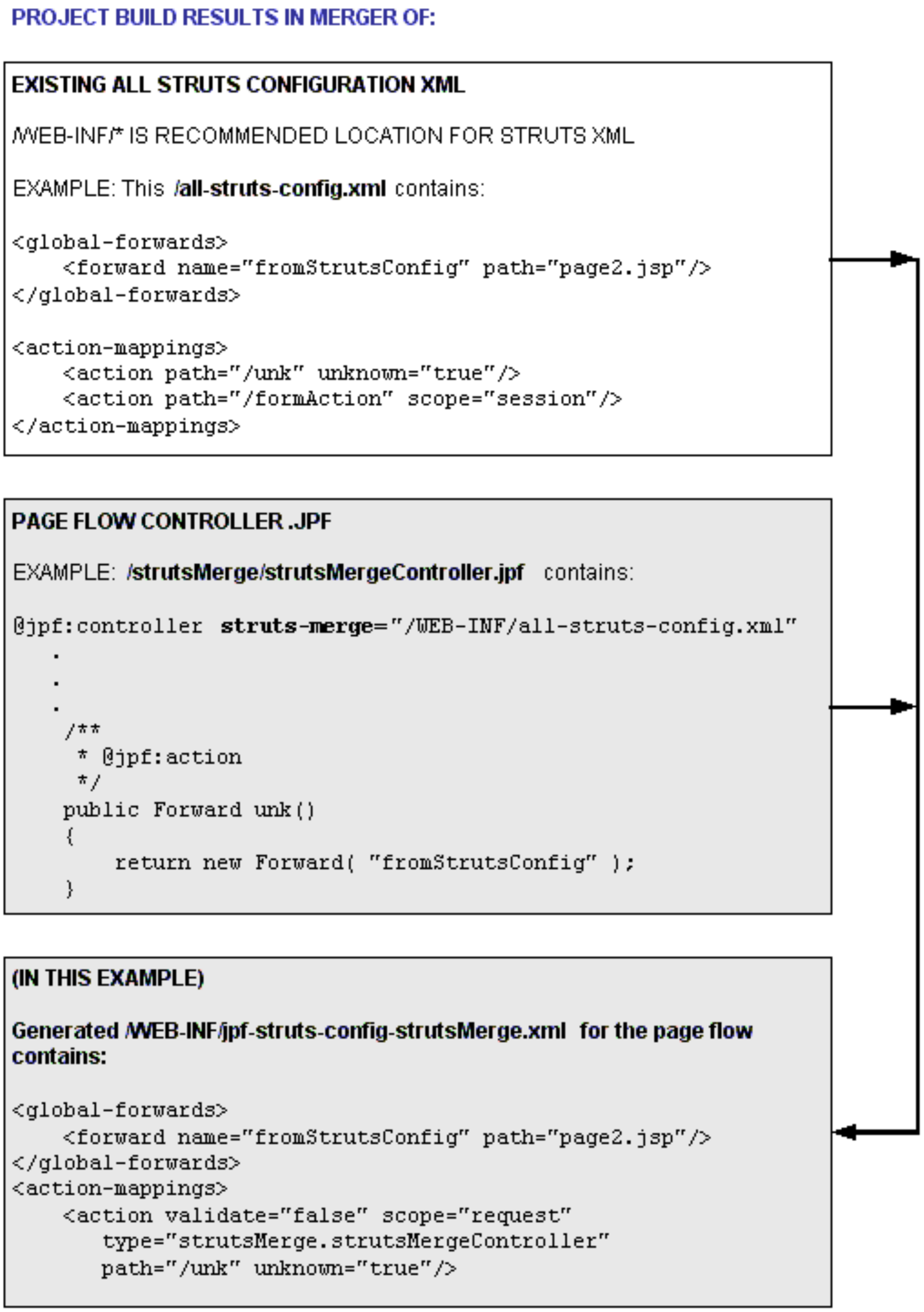
Merging Struts Artifacts Into Page Flows

WebLogic Workshop allows you to merge existing Struts artifacts, such as actions, into the configuration XML that is generated for a page flow. You can use a class−level annotation, @jpf:controller struts−merge, to enable the merger. It occurs when the web project is compiled. After the merger the Struts artifacts from the existing configuration are available for use in the page flow.

This topic focuses on the merger of a Struts configuration and a page flow. For information about enabling Struts modules and page flows to exist in the same web project, and allowing them to interact by using form beans, see Interoperating With Struts and Page Flows.

The following diagram illustrates the concepts of this feature, which is called "Struts Merge." The purpose of this feature is to enable you to override page flow defaults, or to specify settings for the page flow that are not provided by page flow annotations or their attributes. The Struts merge files should typically be small and only modify the page flow and its actions and form beans. While you could, for example, add action mappings in the Struts merge file, BEA does not recommend this practice. Struts action mappings should be declared in the page flow's .jpf file with @jpf annotations, and then (if necessary) modified with the Struts merge file.

Note: In the event of a naming conflict during the merger, artifacts in the Struts merge file will supercede any corresponding entities in the page flow.



PROJECT BUILD RESULTS IN MERGER OF:

EXISTING ALL STRUTS CONFIGURATION XML

/WEB-INF/* IS RECOMMENDED LOCATION FOR STRUTS XML

EXAMPLE: This **/all-struts-config.xml** contains:

```
<global-forwards>
    <forward name="fromStrutsConfig" path="page2.jsp"/>
</global-forwards>

<action-mappings>
    <action path="/unk" unknown="true"/>
    <action path="/formAction" scope="session"/>
</action-mappings>
```

PAGE FLOW CONTROLLER .JPF

EXAMPLE: **/strutsMerge/strutsMergeController.jpf** contains:

```
@jpf:controller struts-merge="/WEB-INF/all-struts-config.xml"
   .
   .
   .
    /**
     * @jpf:action
     */
    public Forward unk()
    {
        return new Forward( "fromStrutsConfig" );
    }
```

(IN THIS EXAMPLE)

Generated /WEB-INF/jpf-struts-config-strutsMerge.xml for the page flow contains:

```
<global-forwards>
    <forward name="fromStrutsConfig" path="page2.jsp"/>
</global-forwards>
<action-mappings>
    <action validate="false" scope="request"
       type="strutsMerge.strutsMergeController"
       path="/unk" unknown="true"/>
```

Requirements of the Struts Module

Here are a few requirements of the Struts module that you will merge with a page flow:

- You must be using Struts 1.1.
- The Struts configuration XML file to be merged with the page flow must by valid xml. Compilation of the page flow's JPF file fails if the Struts merge file does not exist at the specified location, or is invalid. The merge file is parsed and validated against the DTD before it is merged. For example, not adding all required attributes in the merge file results in an error, even if the page flow generated file has all the remaining, required attributes. The merge file must be valid on its own before the merge

operation can continue.
- By convention, the Struts module's XML file should reside in the web project's /WEB−INF directory. However the Struts XML may reside in other directories, as long as the directory is part of the web project's hierarchy of directories.
- All your compiled Struts−related classes (*.class bytecode) that are in a JAR file must reside in the WEB−INF/lib directory. Your Struts *.class files, if they are not in a JAR, must go in the WEB−INF/classes directory. Placing your user−defined Struts classes or JARs in another directory, such as APP−INF/lib or <WEBLOGIC_HOME>/server/lib, will cause classloader problems.

Requirements of the Page Flow Controller

In the page flow that will use artifacts from a Struts configuration, you must specify the @jpf:controller annotation and use the struts−merge attribute. This annotation appears at the class−level for the page flow controller. For example:

```
/**
 * @jpf:controller struts−merge="/WEB−INF/all−struts−config−merge−example.xml"
 */
 public class strutsMergeController extends PageFlowController
 {
```

Then in your page flow action methods, make sure you use the exact capitalization of actions or other artifacts that will be merged from the Struts configuration.

The Merge Rules

In the generated jpf−struts−config−<pageflow>.xml file, each <action> tag from the Struts file is merged using the "path" attribute as a key. Similarly:

- Each <forward> tag from the Struts file is merged using the "name" attribute
- Each <form−bean> tag from the Struts file is merged using the "name" attribute
- Each <exception> tag from the Struts file is merged using the "type" attribute

A Merge Sample

WebLogic Workshop provides a Struts merge sample that you can run. In the IDE, choose File > Open > Application... and browse to:

`<WEBLOGIC_HOME>/samples/workshop/SamplesApp/SamplesApp.work`

With the application open in the IDE, open the WebApp project. Then look at the files in the /strutsMerge directory. Also see the all−Struts merge file in /WEB−INF/all−struts−config−merge−example.xml. It includes the following global forward and a mapping for an "unknown handler":

```
<global−forwards>
    <forward name="fromStrutsConfig" path="page2.jsp"/>
</global−forwards>

<action−mappings>
    <action path="/unk" unknown="true"/>
    .
    .
```

.

In the /strutsMerge directory, see the page1.jsp page. To demonstrate this Struts Merge feature, the page1.jsp page contains a link that refers to a deliberately undefined "foo.do" action.

```
<netui:anchor href="foo.do">This link refers to an unknown action named "foo.do"</netui:anchor
```

On the rendered page1.jsp, when you click the link that refers to the "foo" action, the "unknown handler" action in /strutsMerge/strutsMergeController.jpf is run. It is defined as follows:

```
    /**
     *
     * @jpf:action
     */
     public Forward unk()
     {
     return new Forward( "fromStrutsConfig" );
     }
```

After the project compilation, which resulted in the merger of the all−Struts and page flow configurations, we were able to use the unknown handler and global forward in the page flow at runtime. In the JPF, the unk() method forwards to the global forward ("fromStrutsConfig") that is defined in the merged file. If you look in the merged /WEB−INF/jpf−struts−config−strutsMerge.xml, it contains:

```
<global−forwards>
    <forward name="fromStrutsConfig" path="page2.jsp"/>
</global−forwards>
<action−mappings>
    <action validate="false" scope="request"
       type="strutsMerge.strutsMergeController"
       path="/unk" unknown="true"/>
```

To run this sample, start the server; then in a browser, open:

http://<host>:<port>/WebApp/Controller.jpf

By default the URL for page flow samples running on your machine is:

http://localhost:7001/WebApp/Controller.jpf

On the samples index.jsp start page, click the Go! button link in the section titled "Merging Struts Functionality with Page Flow". When you run the strutsMerge sample, notice how the global action named "fromStrutsConfig" resulted in the forward from page1.jsp to page2.jsp, via the unknown handler and the global forward.

Related Topics

For information about using a form bean to pass data between all−Struts and page flow applications, see:

Interoperating With Struts and Page Flows

Sample Code

Struts Merging Sample

Interoperating With Struts and Page Flows

Page flows in WebLogic Workshop 8.1 are based on Struts 1.1, which is a part of the Jakarta™ Project by the Apache Software Foundation™. The programming framework that is provided by WebLogic Workshop page flows adds many enhancements, including the automatic generation and synchronization of XML configuration files, plus a sophisticated graphical IDE to define, build, deploy, and maintain the web application.

WebLogic Workshop provides interchangeable support for Struts modules and page flow controller classes working together in the same web project. This feature, called "Struts Interop," extends the reach of both types of applications:

- In a page flow, you can import and use form beans that are defined by Struts 1.1 modules. This allows you to take advantage of existing Struts code quickly, without having to make modifications in your Struts action classes and form beans.
- In your Struts 1.1 action classes, you can import and use form beans that are defined by page flows.

The files that comprise your Struts modules and the page flows, if they are to interoperate, must exist in the same web project. Unless there are conflicts with the names of page flow directories, you should be able to use your existing hierarchy of directories for the files that are used by the Struts module. However, the class or JAR files must reside in specific directories, as noted in Requirements of the Struts Modules and Page Flows.

The Struts Interop feature is different from the Struts Merge feature described in Merging Struts Artifacts Into Page Flows. In the case of the Struts Merge feature, you place a @jpf:controller struts−merge="..." annotation in your page flow JPF file. This step enables an existing, all−Struts XML configuration file to be merged (at project compilation time) with the page flow's generated jpf−struts−config−<pageflow>.xml file. The purpose of the Struts Merge feature is to enable you to override page flow defaults, or to specify settings for the page flow that are not provided by page flow annotations or their attributes. The Struts merge files should typically be small and only modify the page flow and its actions and form beans. While you could, for example, add action mappings in the Struts merge file, BEA does not recommend this practice. Struts action mappings should be declared in the page flow's .jpf file with @jpf annotations, and then (if necessary) modified with the Struts merge file.

In the case of the "Struts Interop" feature, the support allows your legacy Struts modules and your new page flows to exist in the same web project, and allows them to interact by using form beans.

You can, of course, use both the Struts Merge and Struts Interop features in a page flow. For detailed information, see A Struts Interop and Struts Merge Example in this topic.

Requirements of the Struts Modules and Page Flows

Here are the requirements of Struts modules and page flows that will interoperate with each other in the same web project:

- As noted earlier, you must be using Struts 1.1.
- The files that comprise the Struts module must reside in the same web project as the page flow. By convention the Struts module should reside in the web project's WEB−INF directory. The source files for the Struts action classes and form beans may reside in <project−root>/WEB−INF/src/<my−struts−directory>/. However, this location is not a requirement.

You can continue to use your existing Struts directory structure in the web project, as long as you do not have naming conflicts (see next item) and both the Struts files and the page flow files are in the same web project.

- All your compiled Struts−related classes (*.class bytecode) that are in a JAR file must reside in the WEB−INF/lib directory. Your Struts *.class files, if they are not in a JAR, must go in the WEB−INF/classes directory. Placing your user−defined Struts classes or JARs in another directory, such as APP−INF/lib or <WEBLOGIC_HOME>/server/lib, will cause classloader problems.
- To avoid naming conflicts, the Struts module names and the page flow directory names must be unique across the web project. For example, if you have a page flow that is in a directory "/pageFlowOne", you cannot specify a Struts module of the same name in the web project's /WEB−INF/web.xml file. The following Struts module elements in web.xml would conflict with the /pageFlowOne directory name in the same web project.

```
<init-param>
   <param-name>config/pageFlowOne</param-name>
   <param-value>/someDir/struts-config-pageFlowOne.xml</param-value>
</init-param>
```

  The prior example shows an incorrect setup that would result in a naming conflict. All your page flow actions will not be found.

- Remember that any JSP pages that reside in a page flow directory always render in the page flow JPF context. If you have JSP pages that raise actions, and you only want them to render in the context of the Struts module, keep those JSP files in another directory, such as in <project−root>/strutsModule.
- If your page flow controller class and the Struts action classes will share a form bean, the form bean can be defined as either an inner class or as an external Java file. Import the form bean class in the page flow JPF and the Struts action classes that will use it. For example, if the PayrollForm form bean is defined in /WEB−INF/src/payroll/PayrollForm.java:

```
import payroll.PayrollForm;
```

  You can use page flow inner−class form beans from Struts modules. The <form−bean...> element would be defined as it is in the generated jpf−struts−config−<pageflow−name>.xml file. For example:

```
 <form-bean type="foo.fooController$MyForm" name="myForm"/>
```

- When you share a form bean, in the Struts module's XML, the value of the <form−bean name=...> attribute must match exactly the value of the name attribute generated in the page flows jpf−struts−config−<pageflow−name>.xml file. The generated name usually is the name of the form bean class, minus the package name, with a lowercase first letter. For example, form bean class xyzProject.sales.OrderForm would be named "orderForm". However if there is a naming conflict when the page flow compiler goes to generate this name, another name will be used. In this case you must look at the generated jpf−struts−config−<pageflow−name>.xml file to determine the name.
- By default, page flows scope form bean instances to the request. If you are sharing a form bean, and you are passing data from (for example) a page flow, to a Struts module, and then back to the page flow, the easiest way to share the data is to use session−scoped form beans. You can accomplish this by using the Struts Merge feature to merge in session−scoped form beans with the Struts configuration XML that will be generated for your page flow. For an example, see the next section.
- If your web project combines page flows and Struts modules, you must register each Struts module by editing the project's /WEB−INF/web.xml file. For an example, see the next section.

A Struts Interop and Struts Merge Example

The following example combines Struts Merge and Struts Interop features to demonstrate the interoperability between page flows and Struts. The strutsInteropController page flow instantiates a form bean, JpfFormBean, then passes this form to a Struts module named strutsModule. The Struts module then alters the contents of the form and passes it back to the page flow. Because page flow forms derive from Struts forms, this sharing of the form bean is possible.

By default, page flows scope form bean instances to the request. But because in this example we want to use session−scoping to share the form bean between the page flow and the struts module, strutsModule, we will cause the page flow to operate on a session−scoped form bean. We do this by using the struts−merge attribute on the @jpf:controller annotation in the strutsInteropController.jpf source file. This annotation will cause the contents of the specified Struts XML file to be merged−in with the generated jpf−struts−config−strutsInterop.xml file.

The form bean we use in this example has a single field, "field1". At each step, the action (whether page flow or Struts) will update the value of "field1" so you can see that the same instance of the form is being passed around.

To demonstrate the "Struts Merge" and "Struts Interop" features in one example, we created the following entities. All these files are part of the sample application that is installed to the following location:

<WEBLOGIC_HOME>/samples/workshop/SamplesApp/WebApp/...

The file specifications shown in the following list are relative to the web project's root directory.

- A page flow named strutsInterop:

  /strutsInterop/StrutsInteropController.jpf

  When you examine this file's source, notice how we are importing the externally defined JpfFormBean form bean. Also notice how we used a class−level annotation, @jpf:controller struts−merge="merge−jpf−struts−config.xml", to merge in session−scoped form beans with the Struts configuration XML that will be generated for your page flow. As a result of the merger, JpfAction1, JpfAction2, and JpfAction3 are available for use in the page flow. Note that by default, page flow form bean instances are scoped for the request. Because our processing will include an entity beyond the page flow's boundary, the easiest way to share it will be to scope it to the session, as we move from (1) the page flow, (2) to the Struts module, and then (3) back to the page flow.

- A form bean that is defined externally in the page flow directory, and is shared by the page flow and the Struts module:

  /strutsInterop/JpfFormBean.java

  It will be instantiated by the page flow and passed to Struts module, strutsModule, then passed back to the page flow. At each step the value of Field1 will be changed to demonstrate that the same instance of the form is being passed between page flow and Struts.

- Two Struts action classes:

/WEB−INF/src/strutsModule/Struts1.java
/WEB−INF/src/strutsModule/Struts2.java

When you examine the Struts1.java source, notice that we import the JpfFormBean that will be shared by the page flow and the Struts module. Struts1 is the first action in the Struts module and effectively causes the page flow runtime system to "bootstrap" or load the Struts module. Then the Struts1 action simply forwards to the Jsp2.jsp page, which then renders in the context of the Struts module, strutsModule.

In the source for Struts2.java, notice that we import the JpfFormBean that will be shared by the page flow and the Struts module. The Struts action method sets a new value for Field1, and then forwards back to the page flow.

- A Struts module that registers the shared JpfFormBean.java and the two Struts*.java action classes:

  /WEB−INF/struts−config−strutsModule.xml
- Several JSP pages in different directories, as follows:

  In the strutsInterop page flow:

  /strutsInterop/Jsp1.jsp
  /strutsInterop/Jsp3.jsp
  /strutsInterop/done.jsp

  For the Struts module, we created a JSP page that is not part of the page flow, but interoperates with it:

  /strutsModule/Jsp2.jsp
- We also created a Struts merge XML file that modifies three of the /strutsInterop page flow's actions, (JpfAction1, JpfAction2, JpfAction3) and sets the form bean instance to be scoped to "session:"

  /strutsInterop/merge−jpf−struts−config.xml

  The purpose of merging the Struts XML is to set the scope of the form beans to session, instead of the default request. It is worth repeating the following explanation. By default, page flows scope form bean instances to the request. Because in this example we want to share the form bean between the page flow and the Struts module strutsModule, we will scope the form bean to the session. We do this by using the struts−merge attribute on the @jpf:controller annotation in the strutsInteropController.jpf source file. This class−level annotation will cause the contents of the following XML file to be merged in with the generated jpf−struts−config−strutsInterop.xml file.
- We edited the web project's /WEB−INF/web.xml file to add an entry for the Struts module:

```
<init-param>
    <param-name>config/strutsModule</param-name>
    <param-value>/WEB-INF/struts-config-strutsModule.xml</param-value>
</init-param>
```

Next we compiled the web project that contains these entities, by selecting the WebApp project name, executing a right−mouse click, and choosing Build Project. Then we started the server and, in a browser, accessed this URL:

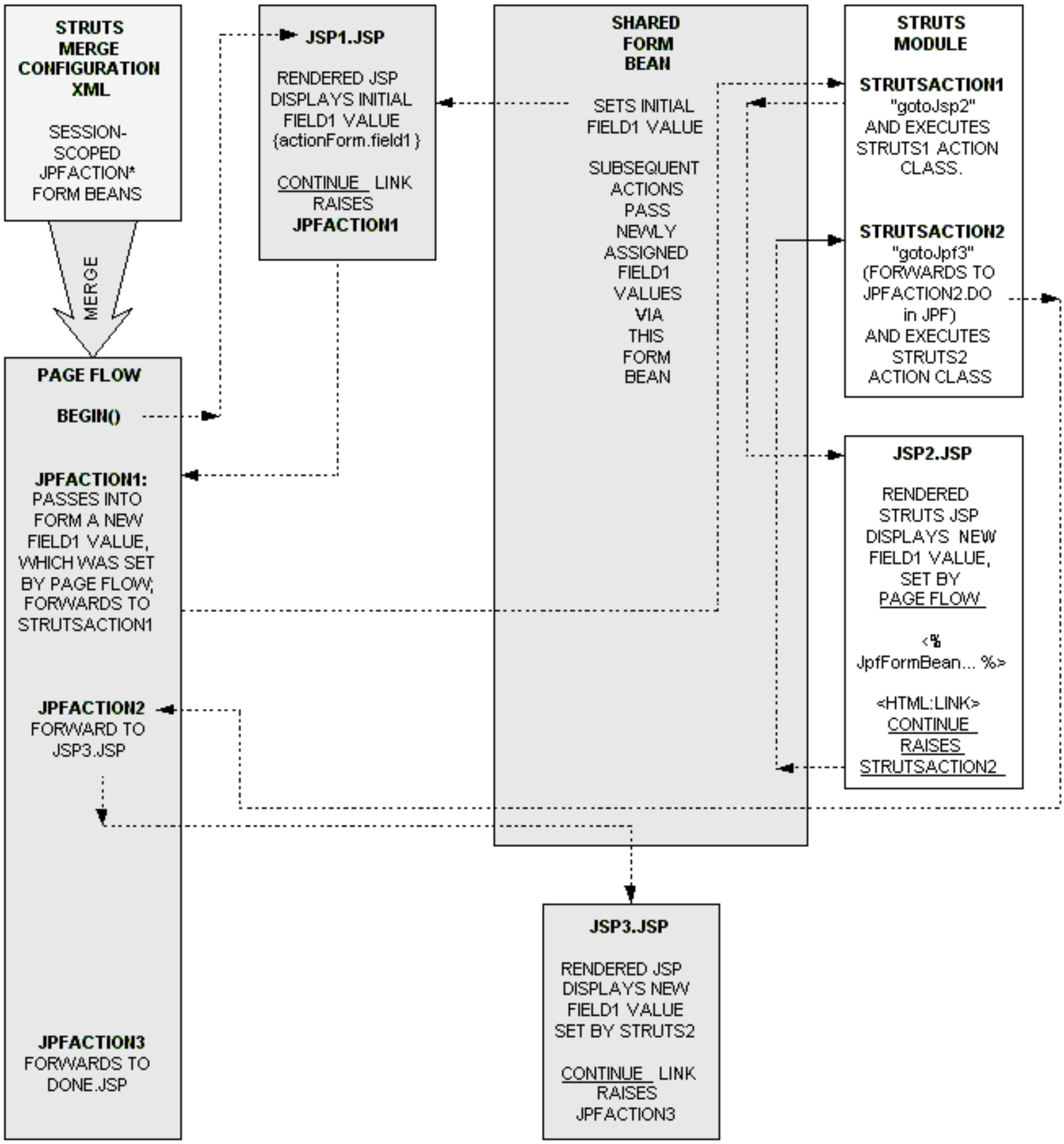
http://localhost:7001/WebApp/strutsInterop/strutsInteropController.jpf

Walking Through the Example

Now let's walk through the processing. It should be helpful to start with an accounting of the field1 values in the shared form. The following table shows the values, the order in which each value was set, which entity set each value, in which rendered JSP page the value is seen, and the page flow or Struts context.

Field1 String Value	***Set By...***	***Rendered in...***
"Form bean Field1 default value set by the form bean itself."	The shared form bean, /strutsInterop/JpfFormBean.java	/strutsInterop/Jsp1.jsp (A page flow JSP)
"Form bean Field1 value set by the page flow controller class."	The page flow controller class, /strutsInterop/strutsInteropController.jpf	/strutsModule/Jsp2.jsp (A Struts JSP)
"Form bean Field1 value set by the Struts2 class."	The Struts2 action class, /WEB−INF/src/strutsModule/Struts2.java	/strutsInterop/Jsp3.jsp (A page flow JSP)

The following diagram illustrates the processing flow. In this diagram, the shaded boxes represent entities that are part of the page flow. The unshaded boxes represent Struts entities.


STRUTS MERGE CONFIGURATION XML
SESSION-SCOPED JPFACTION* FORM BEANS
MERGE
JSP1.JSP
RENDERED JSP DISPLAYS INITIAL FIELD1 VALUE {actionForm.field1}
CONTINUE LINK RAISES JPFACTION1
SHARED FORM BEAN
SETS INITIAL FIELD1 VALUE
SUBSEQUENT ACTIONS PASS NEWLY ASSIGNED FIELD1 VALUES VIA THIS FORM BEAN
STRUTS MODULE
STRUTSACTION1 "gotoJsp2" AND EXECUTES STRUTS1 ACTION CLASS.
STRUTSACTION2 "gotoJpf3" (FORWARDS TO JPFACTION2.DO in JPF) AND EXECUTES STRUTS2 ACTION CLASS
PAGE FLOW
BEGIN()
JPFACTION1: PASSES INTO FORM A NEW FIELD1 VALUE, WHICH WAS SET BY PAGE FLOW; FORWARDS TO STRUTSACTION1
JPFACTION2 FORWARD TO JSP3.JSP
JPFACTION3 FORWARDS TO DONE.JSP
JSP2.JSP
RENDERED STRUTS JSP DISPLAYS NEW FIELD1 VALUE, SET BY PAGE FLOW
<% JpfFormBean... %>
<HTML:LINK> CONTINUE RAISES STRUTSACTION2
JSP3.JSP
RENDERED JSP DISPLAYS NEW FIELD1 VALUE SET BY STRUTS2
CONTINUE LINK RAISES JPFACTION3

Here are the steps:

1. When the web project was compiled, among the processing that occurred was the merger of the ***session−scoped*** jpfAction1, jpfAction2, and jpfAction3 form beans from the Struts /strutsInterop/merge−jpf−struts−config.xml file into the page flow's generated /WEB−INF/jpf−struts−config−strutsInterop.xml file, which includes:

```
<action−mappings>
  <action validate="false" type="strutsInterop.strutsInteropController" name="jpfFormBe
    <forward contextRelative="true" path="/strutsModule/strutsAction1.do" name="gotoSt
  </action>
```

```
    <action validate="false" type="strutsInterop.strutsInteropController" name="jpfFormBe
      <forward path="/Jsp3.jsp" name="gotoPg3"/>
    </action>
    <action validate="false" type="strutsInterop.strutsInteropController" name="jpfFormBe
      <forward contextRelative="true" path="/strutsInterop/done.jsp" name="gotoDone"/>
    </action>
```

These actions are now available for use in the page flow.

2. With the server running, a browser user accesses the strutsInteropController.jpf page flow by specifying it in the URL. For test purposes, the URL might be:

   http://localhost:7001/WebApp/strutsInterop/strutsInteropController.jpf

3. When the page flow is accessed, its begin() action is run and the forward loads the Jsp1.jsp page. Also the page flow instantiates an imported form bean, JpfFormBean.
4. This extended step will focus on what happens as the Jsp1.jsp page is rendered by the server. The pre−rendered JSP includes the following:

   ```
    <netui:form action="jpfAction1">
        <netui:label value="{actionForm.field1}"/>
    </netui:form>

    <br/>
    <netui:anchor action="jpfAction1">Continue</netui:anchor>
   ```

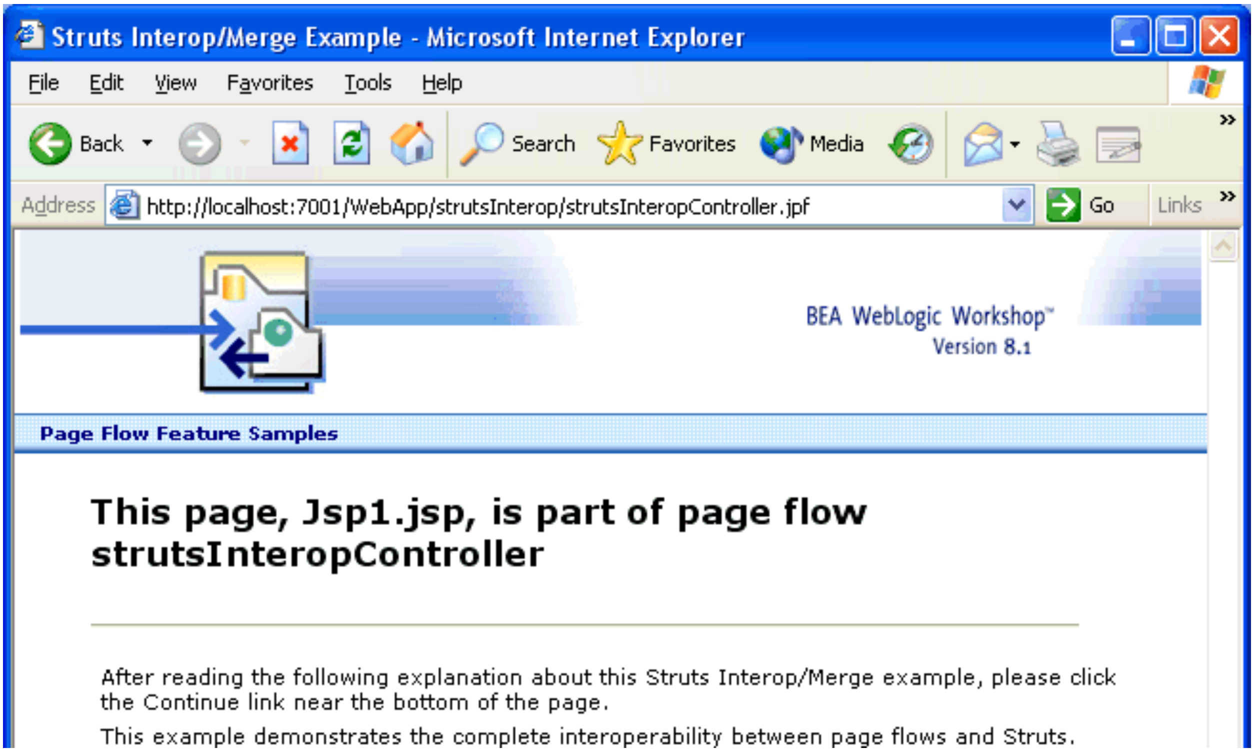
   In the page flow, jpfAction1 is defined as follows:

   ```
       /**
        * @jpf:action
        * @jpf:forward name="gotoStruts" path="/strutsModule/strutsAction1.do"
        */
        protected Forward jpfAction1(JpfFormBean inForm)
        {
            inForm.setField1(this.FORM_VALUE);
            return new Forward("gotoStruts");
        }
   ```

   In the form bean, the initial value for the actionForm context of field1 is set to a String, as follows:

   ```
   public class JpfFormBean extends com.bea.wlw.netui.pageflow.FormData
       {
       public final static String FORM_VALUE   = "Form bean Field1 default value set by the
       private             String field1       = this.FORM_VALUE;
   ```

   Thus the initial form content on the rendered Jsp1.jsp (part of the page flow) displays the String. The following sample screens show only the top and bottom portions of the rendered Jsp1.jsp page:

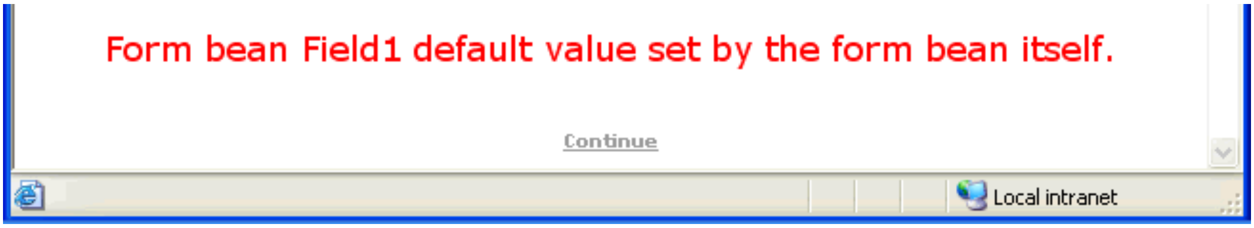

Struts Interop/Merge Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:7001/WebApp/strutsInterop/strutsInteropController.jpf
Go Links
BEA WebLogic Workshop™
Version 8.1
Page Flow Feature Samples

This page, Jsp1.jsp, is part of page flow strutsInteropController

After reading the following explanation about this Struts Interop/Merge example, please click the Continue link near the bottom of the page.

This example demonstrates the complete interoperability between page flows and Struts.

...


Form bean Field1 default value set by the form bean itself.

5. When the user clicks the Continue link on Jsp1.jsp, the jpfAction1 results in a forward to the /strutsModule/strutsAction1.do. This results in forwarding to the Struts Jsp2.jsp page and passing in content on the shared form bean. However a number of intermediate steps happen and need to be understood.

   Remember that Jsp1.jsp is managed by the page flow context. The first entity to set a value for field1 in the form was the form bean itself, as noted earlier. Again, the jpfAction1 method was defined as follows, and included the bolded line shown here:

```
/**
 * @jpf:action
 * @jpf:forward name="gotoStruts" path="/strutsModule/strutsAction1.do"
 */
 protected Forward jpfAction1(JpfFormBean inForm)
 {
     inForm.setField1(this.FORM_VALUE);
     return new Forward("gotoStruts");
 }
```

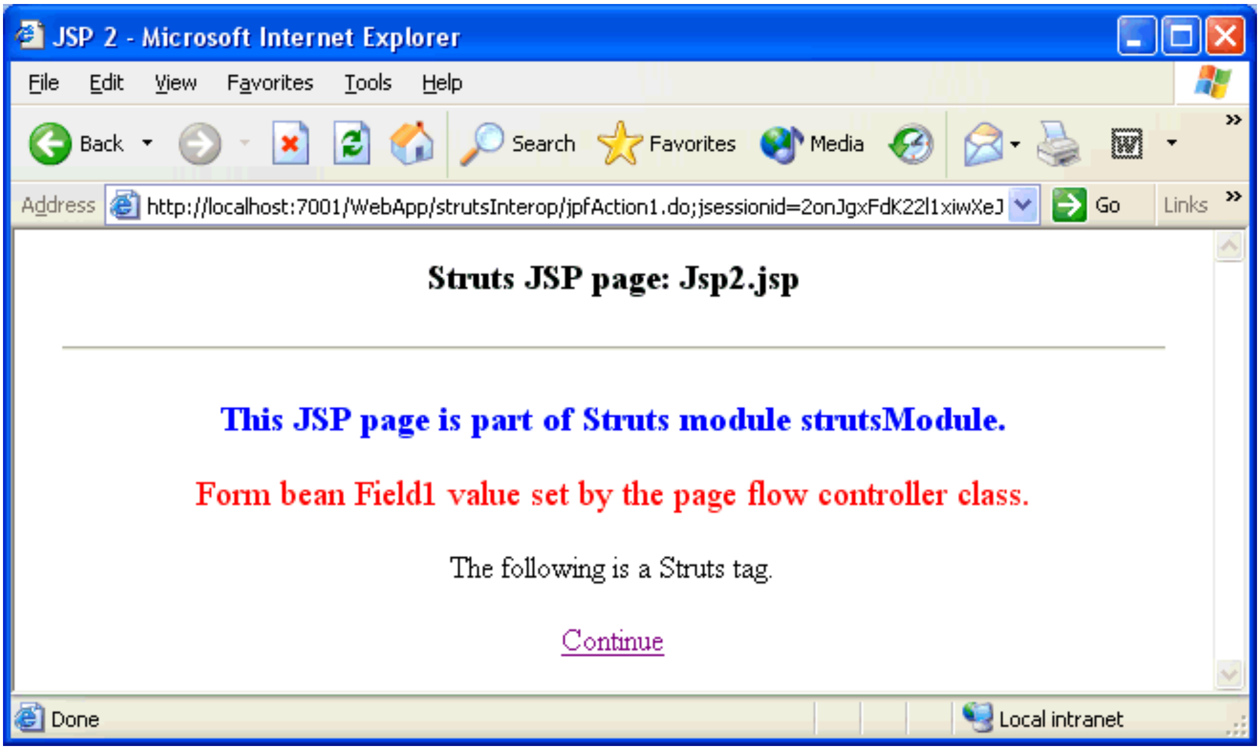
   Thus as navigation control passes from /strutsInterop/Jsp1.jsp to /strutsModule/Jsp2.jsp, the value of field1 that was set in the page flow controller class is passed, too. This value was set in the page flow with the following statement:

```
 public static final String FORM_VALUE = "Form bean Field1 value set by the page flow co
```

That string is displayed on Jsp2.jsp when it is rendered by the server. The pre−rendered JSP includes the following:

```
<%@taglib uri="http://jakarta.apache.org/struts/tags−html" prefix="html" %>
<%@ page import="org.apache.struts.action.ActionForm" %>
<%@ page import="strutsInterop.JpfFormBean" %>

...

     <%
       JpfFormBean tmpForm = (JpfFormBean) request.getSession().getAttribute("jpfFormBea
       out.write(tmpForm.getField1());
     %>

...

     <html:link action="strutsAction2">Continue</html:link>
```

Because the scope for the jpfAction1 form bean in the page flow was set to session, the scriplet you see in the Struts module's Jsp2.jsp can get the session context value for field1 from the shared form bean. That is why we see the following red text in the rendered Jsp2.jsp page:



http://localhost:7001/WebApp/strutsInterop/jpfAction1.do;jsessionid=2onJgxFdK22l1xiwXeJ

Struts JSP page: Jsp2.jsp

This JSP page is part of Struts module strutsModule.

Form bean Field1 value set by the page flow controller class.

The following is a Struts tag.

Continue

But how did we go from Jsp1.jsp in the page flow, to Jsp2.jsp in the Struts module? Let's focus now on how that happened. Remember that the page flow jpfAction1 method was annotated with this @jpf:forward:

```
    /**
```

```
* @jpf:action
* @jpf:forward name="gotoStruts" path="/strutsModule/strutsAction1.do"
*/
protected Forward jpfAction1(JpfFormBean inForm)
{
inForm.setField1(this.FORM_VALUE);
return new Forward("gotoStruts");
}
```

The web project is running with registered servlets. One of the servlets that we registered in the /WEB−INF/web.xml is:

```
<!−− Declare struts module: strutsModule −−>
<init−param>
   <param−name>config/strutsModule</param−name>
   <param−value>/WEB−INF/struts−config−strutsModule.xml</param−value>
</init−param>
```

In the Struts module, the strutsAction1 action is defined as follows:

```
 <action
     path="/strutsAction1"
     name="jpfFormBean"
     scope="session"
     validate="false"
     type="strutsModule.Struts1" >

     <!−− The forward below forwards to a JSP that is part of the Struts module −−>
     <forward name="gotoJsp2" contextRelative="true" path="/strutsModule/Jsp2.jsp" />
 </action>
```

Notice how the Struts module's action mapping for /strutsAction1 contains the following line:

```
     type="strutsModule.Struts1" >
```

That line identifies a Struts action class. In our example, the source is located in /WEB−INF/src/strutsModule/Struts1.java. When this action class runs, it returns a "gotoJsp2" forward name. Back in the Struts module, that forward name corresponds to:

```
 <forward name="gotoJsp2" contextRelative="true" path="/strutsModule/Jsp2.jsp"      />
```

So effectively, when users clicks the Continue link on the rendered Jsp1.jsp, they are forwarded to Jsp2.jsp.

Here is a summary about this important step. When the page flow jpfAction1 forwards to (path="/strutsModule/strutsAction1.do") the page flow runtime system gets this request and scans its list of Struts modules, looking for a Struts module called "strutsModule". As noted earlier, page flows are implemented as Struts modules. The page flow runtime finds the Struts module "strutsModule" because we declared it in the project's /WEB−INF/web.xml file. The page flow then bootstraps, or loads, the Struts module by processing the "struts−config−strutsModule.xml" file, which was also specified in the web.xml file. The runtime now has information about all the action mappings, forms, and so on, for this Struts module. It has the Struts module context. Next, the page flow runtime instantiates the Java class specified by the (type="strutsModule.Struts1") attribute in the /WEB−INF/struts−config−strutsModule.xml (the context) file and calls its "execute" method, passing in the form bean instance that was located in the session.

6. On the rendered Jsp2.jsp page, when the user clicks the Continue link, the action named strutsAction2 is raised in the ***Struts*** module. It is defined in the registered Struts module in /WEB−INF/struts−config−strutsModule.xml as follows:

```
<action
    path="/strutsAction2"
    name="jpfFormBean"
    scope="session"
    validate="false"
    type="strutsModule.Struts2" >

    <!-- The forward below returns to the *PAGE FLOW* -->
    <forward name="gotoJpf3" contextRelative="true" path="/strutsInterop/jpfAction2.do"
</action>
```

We are operating in the Struts module context. In /WEB−INF/src/strutsModule/Struts2.java, a new value for FORM_VALUE (field1) is set:

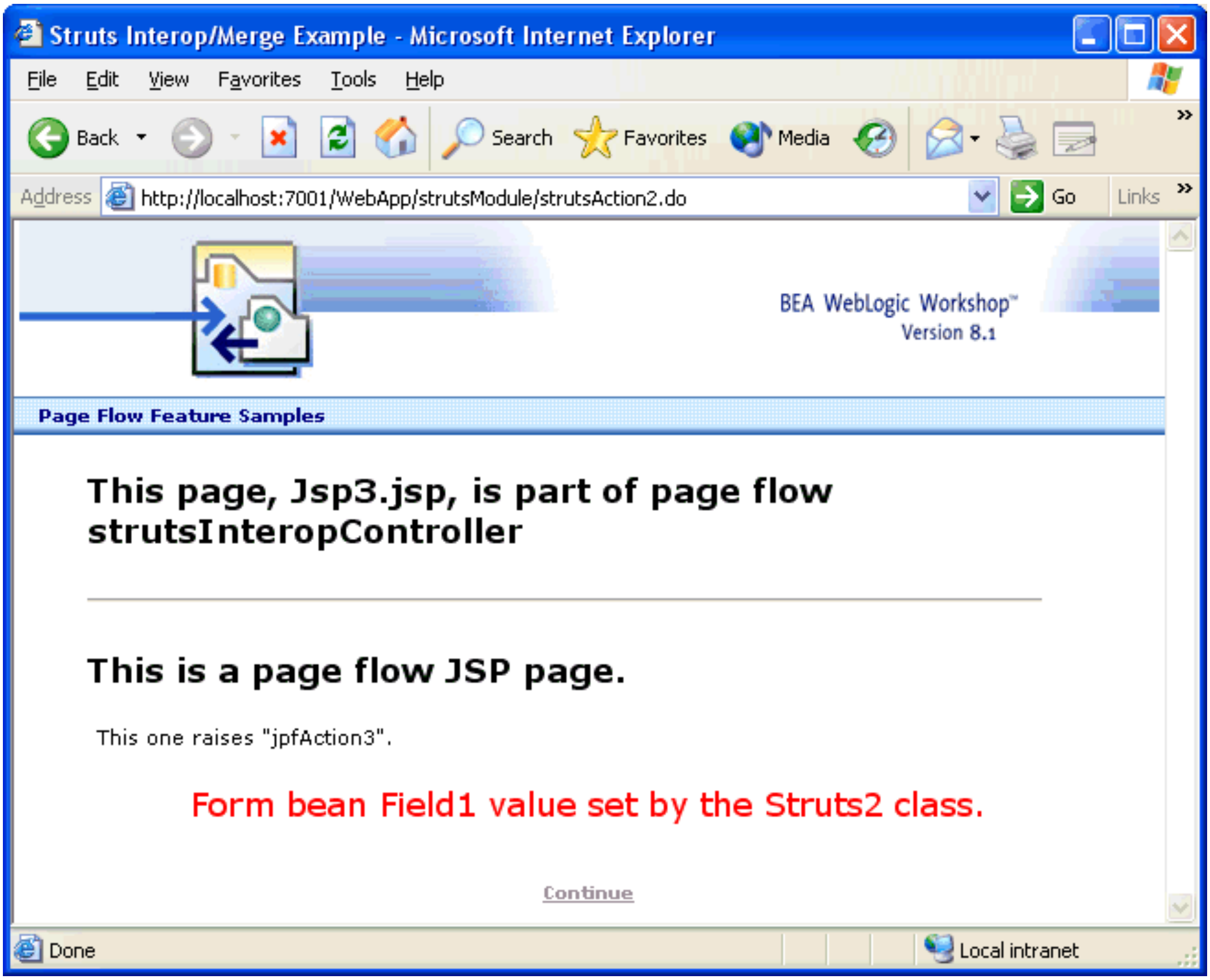
```
public final static String FORM_VALUE = "Form bean Field1 value set by the Struts2 clas
```

With that new value set and the user's click of the Continue link, the strutsAction2 action forwards to the jpfAction2.do action that is defined in the page flow. When we run that action in the page flow, it results in a forward to the page flow's Jsp3.jsp, as shown in this extract from the page flow controller class:

```
   /**
    * This action method was raised by an action in the struts module: strutsModule.
    *
    * @jpf:action
    * @jpf:forward name="gotoPg3" path="Jsp3.jsp"
    */
    protected Forward jpfAction2(JpfFormBean inForm)
    {
    return new Forward("gotoPg3");
    }
```

Note: Struts2 forwards to (path="/strutsInterop/jpfAction2.do"). The runtime looks for a Struts module "strutsInterop" and finds the page flow "strutsInterop". Again, page flows are implemented as Struts modules. However, page flows that you create do not have to be declared in the project's /WEB−INF/web.xml. The runtime automatically registers Struts modules that are generated from page flows.

Notice how the rendered Jsp3.jsp, which is part of the page flow, displays the field1 form value that was set by the Struts2 action class:



Struts Interop/Merge Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:7001/WebApp/strutsModule/strutsAction2.do
Go Links
BEA WebLogic Workshop™
Version 8.1
Page Flow Feature Samples

This page, Jsp3.jsp, is part of page flow strutsInteropController

This is a page flow JSP page.

This one raises "jpfAction3".

Form bean Field1 value set by the Struts2 class.

Continue
Done
Local intranet

The remaining processing is simply the JpfAction3 action that is raised by the Continue link on Jsp3.jsp. It forwards to the page flow's done.jsp page.

Related Topics

For information about merging the configuration XML, see:

Merging Struts Artifacts Into Page Flows

Sample Code

Struts Interoperation Sample

Page Flow and JSP Reference

These topics provide reference information for developing page flows and JSP pages.

Topics Included in This Section

Page Flow Annotations

Provides the annotations that define the behavior of the page flow controller class and its methods.

JSP Tags Reference

Provides the tags designed for use with page flow web projects.

JSP Tags Hierarchy

Shows the parent tags required for each tag in the JSP tag library.

Summary of IDE Windows and Wizards for Page Flows

Describes the IDE components that assist your work as you create page flows and JSP files.

Summary of Flow View Icons

Describes the icons used in the Flow View of a JPF controller file.

Related Topics

Guide to Building Page Flows

Page Flow Annotations

The page flow annotations used in web applications define the behavior of the page flow controller class and its methods.

@jpf:action Annotation

Designates an action method. Optionally, this annotation may set login and J2EE role requirements. Without a @jpf:action annotation, an action method will not be recognized by the page flow runtime as an action method. This annotation may be used at the method−level only.

@jpf:catch Annotation

Provides information used by the page flow runtime to catch exceptions and possibly route them to error handlers. This annotation may be used on action methods and the page flow class, and in the global page flow that resides in WEB−INF/src/global/Global.app.

@jpf:controller Annotation

Indicates whether the page flow is nested; whether users must be logged in; whether logged−in users must be associated with existing J2EE roles; and whether a Struts XML configuration file should be merged into the page flow. This annotation may be used at the page flow class level only.

@jpf:exception−handler Annotation

Designates an error handler method. Without a @jpf:exception−handler annotation, the page flow runtime will not recognize an error handler method.

@jpf:forward Annotation

Describes a location to which the page flow runtime should go to next. The location may be a JSP, a page flow, or an action method. This annotation may be used on action methods and the page flow class, and in the global page flow that resides in WEB−INF/src/global/Global.app.

@jpf:message−resources Annotation

Specifies a message bundle, allowing you to implement validation in the page flow's form beans. This annotation may be used at the page flow class level only.

@jpf:validation−error−forward Annotation

Indicates which page should be loaded, or which action should be run, if a form validation error occurs as a result of running the annotated action. This annotation may be used at the method level only.

@jpf:view−properties Annotation

Describes layout information for your page flow's graphical Flow View. Do not edit this data.

@common:control Annotation

Indicates that a member variable in a page flow references a WebLogic Workshop control, such as a control or a database control.

Related Topics

Guide to Building Page Flows

JSP Tags Reference

WebLogic Workshop provides JSP tags in the following tag libraries. These JSP tags are designed for use with page flow web projects.

<netui:...> Tags

<netui−data:...> Tags

<netui−template:...> Tags

<netui:...> Tags

Tag	***Summary***
<netui:anchor>	Generates a URL−encoded hyperlink to a specified URI.
<netui:attribute>	Adds an attribute and value to the parent tag.
<netui:base>	Provides the base for every URL on this page.
<netui:bindingUpdateErrors>	A development−time aide that displays messages for data binding update errors that occurred when a form was posted. The messages are displayed on the command window. By default, this tag is disabled when the server is running in Production mode.
<netui:button>	Create a button on your JSP page. Place this tag in a <netui:form> ... </netui:form> tag set.
<netui:checkBox>	Generates a checkbox that binds to a form bean property or databound expression.
<netui:checkBoxGroup>	Groups a collection of CheckBoxOptions, and handles data binding of their values.
<netui:checkBoxOption>	A checkbox whose state is determined by its enclosing CheckBoxGroup.
<netui:content>	Used to display text or the result of an expression to the page.
<netui:error>	Renders an error message with a given error key value.
<netui:errors>	Used to report multiple validation errors.
<netui:exceptions>	Displays formatted exception messages.
<netui:fileUpload>	Uploads a file from the client to the server.
<netui:form>	Represents an input form, associated with a bean whose properties correspond to the various fields of the form.
<netui:formatDate>	A formatter used to format dates.
<netui:formatNumber>	A formatter used to format numbers.

<netui:formatString>	A formatter used to format strings.
<netui:getNetuiTagName>	Returns the value of a specified tagId attribute.
<netui:hidden>	Generates a hidden tag with a given value.
<netui:html>	Generates the html element and performs error handling within its body.
<netui:image>	Places an image file type on your page.
<netui:imageAnchor>	Combines the functionality of the netui:image and netui:anchor tags.
<netui:imageButton>	Combines the functionality of the netui:image and netui:button tags.
<netui:label>	Places formatted or dynamically generated text on the page.
<netui:node>	Instantiates a TreeNode object that will get added to the parent tag (either a Tree or another Node).
<netui:parameter>	Writes a URL parameter to a URL on its parent tag.
<netui:parameterMap>	Provides a read−only data binding expression that points to a map of parameters. Each entry in the map provides a URL parameter that will be added to the parent tag's URL.
<netui:radioButtonGroup>	Defines a group of netui:radioButtonOption elements.
<netui:radioButtonOption>	A radio button whose state is determined by its enclosing netui:RadioButtonGroup.
<netui:rewriteName>	Allows the URL Rewriter to rewrite the name attribute before it is output into the HTML stream
<netui:rewriteURL>	Allows the URL Rewriter to rewrite the url attribute before it is output into the HTML stream
<netui:scriptContainer>	Each scriptContainer will collect all the JavaScript defined for other tags that are embedded with the <netui:scriptContainer> ... </netui:scriptContainer> tag set, and output all the JavaScript in a single <script>. Also each scriptContainer provides optional scoping of its tagID, which allows you to set the scope of the JavaScript's methods.
<netui:select>	Defines a multiple−choice menu or drop−down list within a netui:form element.
<netui:selectOption>	An option whose state is determined by its enclosing netui:selectOption.
<netui:textArea>	Renders a databound TextArea with the given attributes.
<netui:textBox>	Renders a databound TextBox with the given attributes.
<netui:tree>	Renders a tree control represented by a set of TreeNode objects.

<netui−data:...> Tags

Tag	***Summary***

<netui−data:anchorColumn>	Renders an HTML anchor into each data cell in the column.
<netui−data:basicColumn>	Renders data from the data set into the page.
<netui−data:callControl>	Used to call a method on a control.
<netui−data:callMethod>	Used to call a method on an object.
<netui−data:callPageFlow>	Used to call a method on the current page flow controller.
<netui−data:cellRepeater>	A repeating, databound tag that renders its body into each cell of a table of the specified dimensions.
<netui−data:choice>	When nested in a netui−data:repeaterItem tag, allows its body to be rendered conditionally.
<netui−data:choiceMethod>	Used to choose a particular netui−data:choice tag whose body should be rendered for the current data item in the Repeater.
<netui−data:columns>	Provides a container for the columns that will render the header, data, and footer for each column in a netui−data:grid.
<netui−data:declareBundle>	Declare a resource bundle that is available in the bundle data binding context.
<netui−data:declareControl>	Declare a control that is stored in the pageContext attribute map.
<netui−data:declarePageInput>	Lets you specify what data this JSP page expects an action to forward. In the JSP Designer, you can later drag elements of the specified data onto the page's canvas for display.
<netui−data:expressionColumn>	A column that can use data binding expressions in addition to formatters to format the value of a data cell.
<netui−data:getData>	Evaluates an expression and places the result in the JSP's Page Context. Can be used to evaluate objects from forms, page flows, and other objects that can be databound. You can then write a scriplet to access the data by using the getAttribute method of javax.servlet.jsp.PageContext.
<netui−data:grid>	The containing tag of a tag set that renders regular data with the ability to page, sort, and filter the data set.
<netui−data:gridStyle>	Allows parameterization of the style components of the HTML table that a Grid renders.
<netui−data:imageColumn>	In a grid, renders an image into a column.
<netui−data:message>	Allows you to format messages according to any sequence you want, using one or more values from arguments defined in <netui−data:messageArg> tag(s). The results are available to the page context.
<netui−data:messageArg>	Allows you to set values that are used as arguments to the <netui−data:message> tag. The formatted message results are available to the page context.
<netui−data:methodParameter>	Used to add an argument to a method that will be called on some object.

<netui−data:pad>	Affects the number of items that are rendered in a Repeater.
<netui−data:pager>	Allows the grid to render a subset of data on each page and renders the navigation links for moving between grid pages.
<netui−data:repeater>	A markup−generic tag that repeats over a data set, and renders the data onto the page.
<netui−data:repeaterFooter>	Used to render the footer of a Repeater.
<netui−data:repeaterHeader>	Used to render the header of a Repeater.
<netui−data:repeaterItem>	Used to render each item in the data set.

<netui−template:...> Tags

Tag	***Summary***
<netui−template:attribute>	Use the netui−template:attribute element by (1) placing it in a tempate file and (2) setting its value with the netui−template:setAttribute tag. (The netui−template:setAttribute tag is placed on the page that invokes the template file.
<netui−template:includeSection>	Include a netui−template:includeSection element in a template file to mark out content to be used in another JSP page.
<netui−template:section>	Used to mark out content to replace a netui−template:includeSection within a template file.
<netui−template:setAttribute>	Used to set the value of an netui−template:attribute element in a template file.
<netui−template:template>	Used to associate a JSP page with a particular template file.
<netui−template:visible>	Used to turns on or off display of the body content based upon the visible state of the tag.

Related Topics

Guide to Building Page Flows

Using Data Binding in Page Flows

Presenting Complex Data Sets in JSPs

JSP Tags Hierarchy

The table below shows the required parent tags of the Page Flow JSP tag library (<netui:...>, <netui−data:...>, and <netui−template:...>).

<netui:...> Tags

Tag	*Required Parent Tag*
<netui:anchor>	none
<netui:base>	none
<netui:bindingUpdateErrors>	none
<netui:button>	none
<netui:checkBox>	none
<netui:checkBoxGroup>	none
<netui:checkBoxOption>	<netui:checkBoxGroup>
<netui:content>	none
<netui:error>	none
<netui:errors>	none
<netui:exceptions>	none
<netui:fileUpload>	none
<netui:form>	none
<netui:formatDate>	<netui:content>, <netui:label>
<netui:formatNumber>	<netui:content>, <netui:label>
<netui:formatString>	<netui:content>, <netui:label>
<netui:hidden>	none
<netui:html>	none
<netui:image>	none
<netui:imageAnchor>	none
<netui:imageButton>	none
<netui:label>	none
<netui:node>	<netui:tree>
<netui:parameter>	<netui:anchor>, <netui:form>
<netui:parameterMap>	<netui:anchor>, <netui:form>

<netui:radioButtonGroup>	none
<netui:radioButtonOption>	<netui:radioButtonGroup>
<netui:rewriteName>	none
<netui:rewriteURL>	none
<netui:scriptContainer>	none
<netui:select>	none
<netui:selectOption>	<netui:select>
<netui:textArea>	none
<netui:textBox>	none
<netui:tree>	none

<netui−data:...> Tags

Tag	***Required Parent Tag***
<netui−data:anchorColumn>	<netui−data:columns>
<netui−data:basicColumn>	<netui−data:columns>
<netui−data:callControl>	none
<netui−data:callMethod>	none
<netui−data:callPageFlow>	none
<netui−data:cellRepeater>	none
<netui−data:choice>	<netui−data:repeaterItem>
<netui−data:choiceMethod>	<netui−data:repeaterItem>
<netui−data:columns>	<netui−data:grid>
<netui−data:declareBundle>	none
<netui−data:declareControl>	none
<netui−data:declarePageInput>	none
<netui−data:expressionColumn>	<netui−data:columns>
<netui−data:getData>	none
<netui−data:grid>	none
<netui−data:gridStyle>	<netui−data:grid>

<netui−data:imageColumn>	<netui−data:columns>
<netui−data:message>	none
<netui−data:messageArg>	<netui−data:message>
<netui−data:methodParameter>	<netui−data:callControl>, <netui−data:callMethod>, <netui−data:callPageFlow>, <netui−data:choiceMethod>
<netui−data:pad>	<netui−data:repeater>
<netui−data:pager>	<netui−data:grid>
<netui−data:repeater>	none
<netui−data:repeaterFooter>	<netui−data:repeater>
<netui−data:repeaterHeader>	<netui−data:repeater>
<netui−data:repeaterItem>	<netui−data:repeater>

<netui−template:...> Tags

Tag	***Required Parent Tag***
<netui−template:attribute>	none
<netui−template:includeSection>	none
<netui−template:section>	<netui−template:template>
<netui−template:setAttribute>	<netui−template:template>
<netui−template:template>	none
<netui−template:visible>	none

Related Topics

JSP Tags Reference

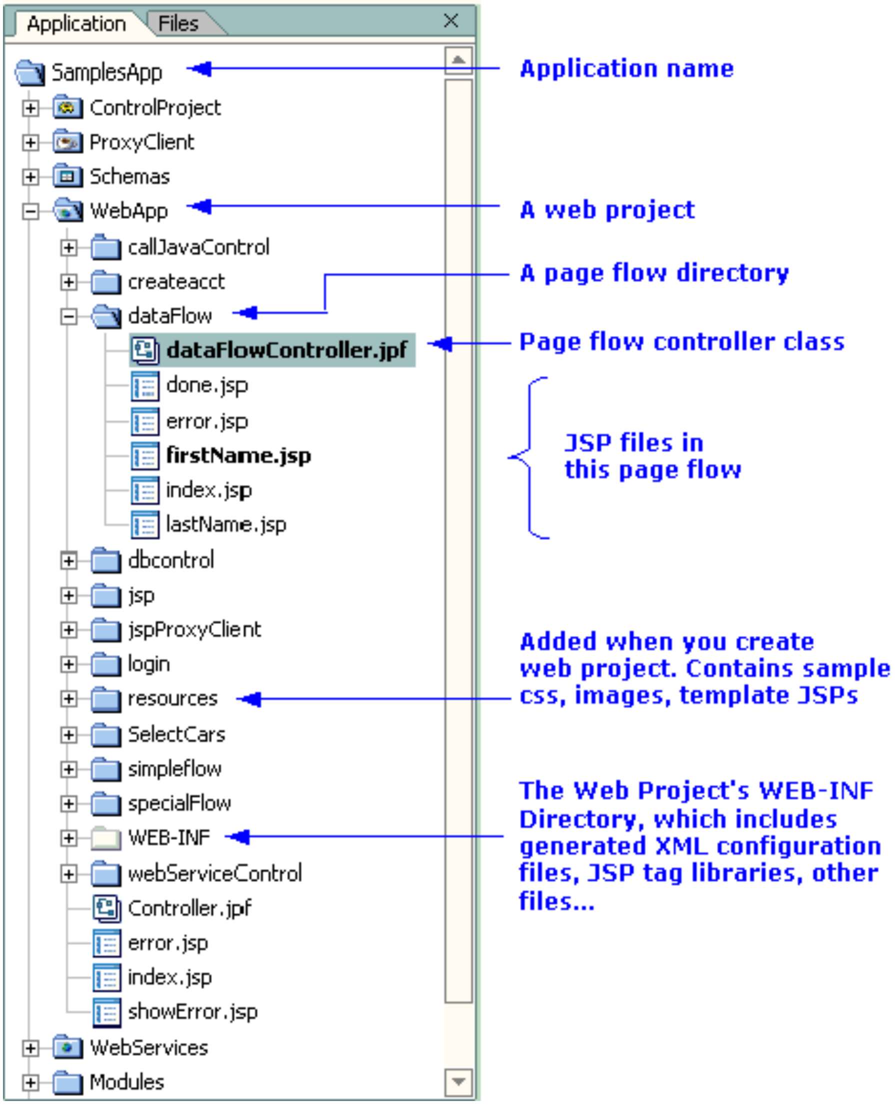
Summary of IDE Windows and Wizards for Page Flows

This topic describes the various windows and wizards that are available to assist your work as you create page flows and JavaServer Pages. The IDE components discussed here are:

- Application Window
- Palette Window for Page Flows
- Data Palette Window for Page Flows
- Document Structure Window for Page Flows
- Property Editor Window When Action View is Open and Form Bean Selected
- Palette Window for JSP Tags
- Property Editor Window for JSP Tags
- Table Navigator Window for JSP Files
- Page Flow Wizard
- New Action Window
- JSP Tag Wizards, Including the Form Wizard

Application Window

The application window shows the hierarchy of directories for the open application. The following sample screen shows the SamplesApp application window, with comments that identify how directories correspond to web application, project, and page flow components.



This application is installed in:

`<WEBLOGIC_HOME>\samples\workshop\SamplesApp`

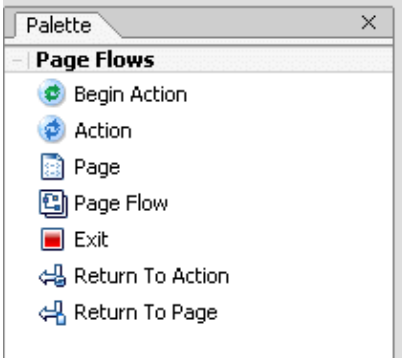
To open an application, use ***File*****−−>*****Open*****−−>*****Application*** in the IDE.

Note: each application uses a *.work file that is located in the application's root directory. If you use ***File*****−−>*****Open*****−−>*****File*** to open the application's *.work* file, the IDE will assume that you want to edit the file instead of opening the application.

Palette Window for Page Flows

Before you can open a page flow, the application it is part of must be open in the IDE. When the application is open, choose the web project directory that contains the page flow. Then open the page flow's *.jpf file name, which by default is <page flow name>Controller.jpf.

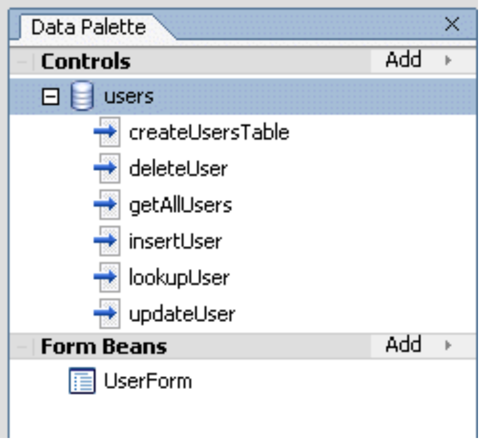
When a page flow's *Controller.jpf file is open, you can use its Palette window:



To add an action, a new JSP, or a reference to another existing page flow to the current page flow, you can select an icon an icon from this palette with your cursor, hold down the mouse button, and move (or "drag") the icon onto the page flow's Flow View canvas. When you drag an action icon onto the Flow View, WebLogic Workshop presents a New Action dialog window. For an introduction to that window, see New Action Window. For more information about the Flow View canvas, see Summary of Flow View Icons.

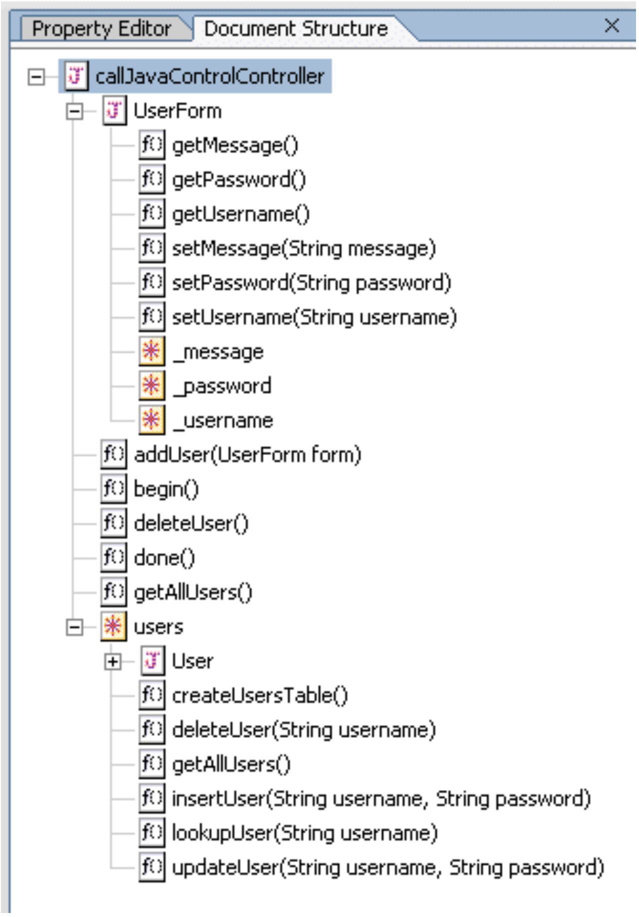
Data Palette Window for Page Flows

When a page flow is open, its Data Palette window identifies any Controls or Form Beans that are defined. You can also click the Add button to add items to the current page flow.



Document Structure Window for Page Flows

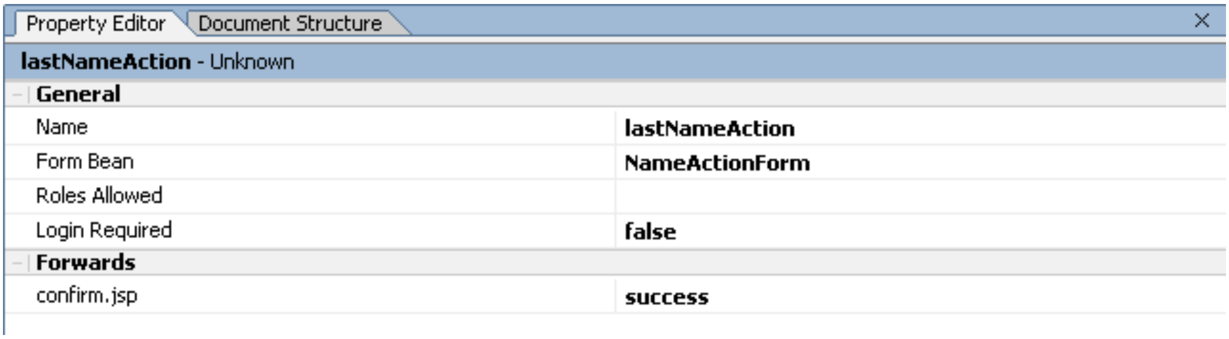
When a page flow is open, its Document Structure window identifies the Java class, methods and signatures, any inner classes, and so on. This window provides a helpful visual summary of the Java entities that make up your page flow class.



- callJavaControlController
  - UserForm
    - getMessage()
    - getPassword()
    - getUsername()
    - setMessage(String message)
    - setPassword(String password)
    - setUsername(String username)
    - _message
    - _password
    - _username
  - addUser(UserForm form)
  - begin()
  - deleteUser()
  - done()
  - getAllUsers()
  - users
    - User
    - createUsersTable()
    - deleteUser(String username)
    - getAllUsers()
    - insertUser(String username, String password)
    - lookupUser(String username)
    - updateUser(String username, String password)

Property Editor Window When Action View is Open and Form Bean Selected

When a page flow is open and an action method that uses a form bean is selected in the Action View, the Property Editor window identifies the action name, the form bean it uses, and related information such as the Forward object's name and the web component (such as a JSP file) that is forwarded to when this action runs.

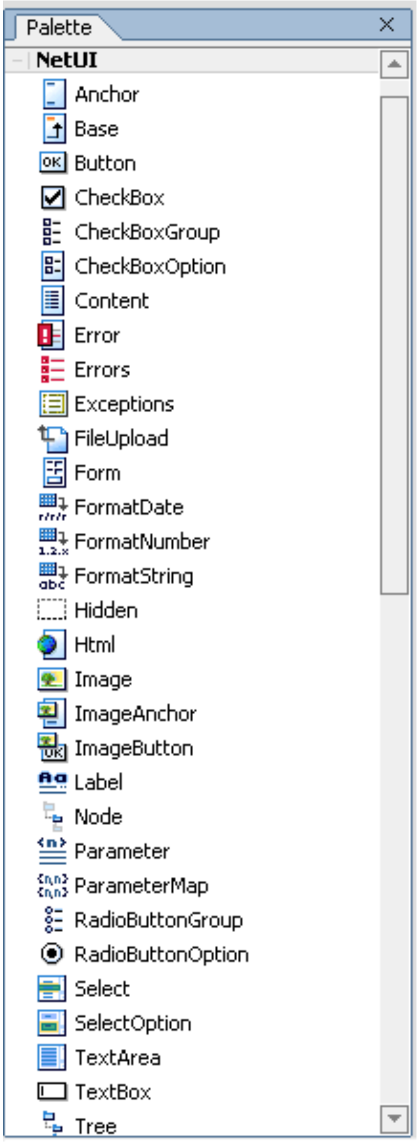


lastNameAction - Unknown

General	
Name	**lastNameAction**
Form Bean	**NameActionForm**
Roles Allowed	
Login Required	**false**
Forwards	
confirm.jsp	**success**

Palette Window for JSP Tags

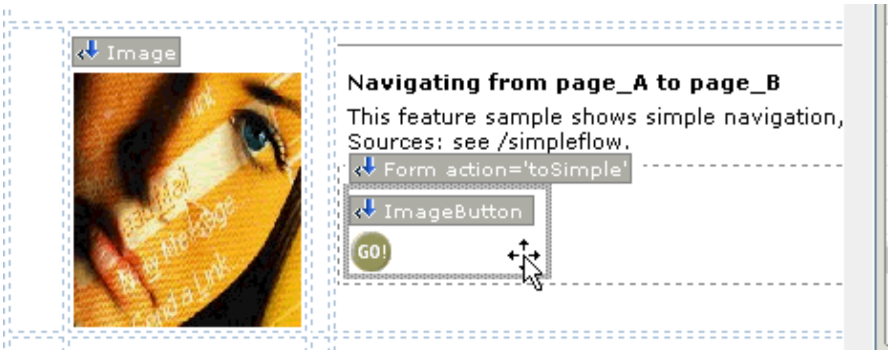
When a JSP file is open, its Palette window shows the special JSP tags that are provided by WebLogic Workshop tag libraries. These JSP tags are designed to save you coding time by implementing commonly used functions, such as user interface controls that can be associated with actions and dynamic data. You can "drag and drop" these tags from the palette to the JSP Design View canvas. In some cases, a dialog window will appear that allows you to assign actions to tags that work with form beans.



The preceding diagram shows a portion of the tags provided. The categories of tags are Netui, Netui Databinding, Netui Templates, and HTML. For more information about these tags, see Designing User Interfaces in JSPs, Using Form Beans to Encapsulate Data, and Presenting Complex Data Sets in JSPs.

Property Editor Window for JSP Tags

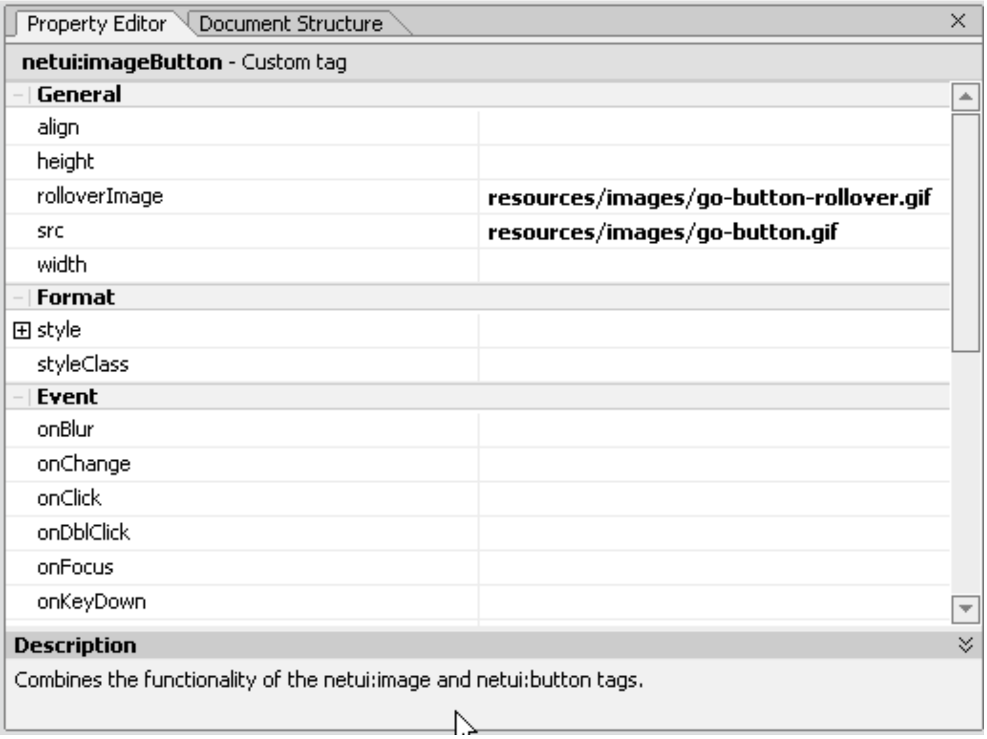
When a JSP file is open in Design View, you can select an item on the page to view, set, or modify its properties in the Property Editor window. For example, if on the JSP Design View canvas we selected the small GO! button:



Navigating from page_A to page_B

This feature sample shows simple navigation,
Sources: see /simpleflow.

Then the Property Editor would contain values, such as the following example:



netui:imageButton - Custom tag

General	
align	
height	
rolloverImage	**resources/images/go-button-rollover.gif**
src	**resources/images/go-button.gif**
width	
Format	
⊞ style	
styleClass	
Event	
onBlur	
onChange	
onClick	
onDblClick	
onFocus	
onKeyDown	

Description

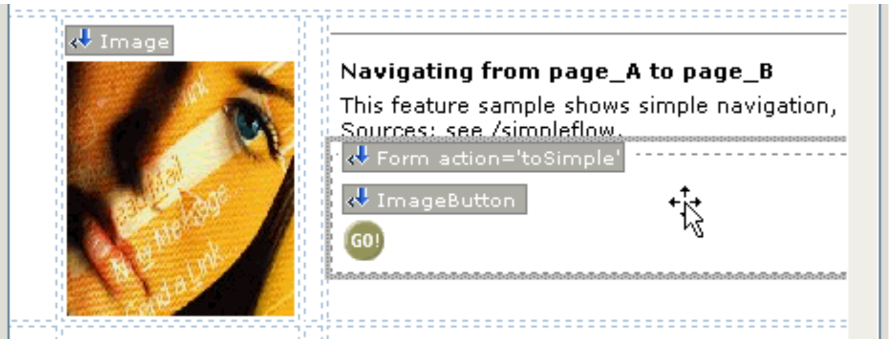
Combines the functionality of the netui:image and netui:button tags.

Notice how the Property Editor has identified the netui:imageButton tag, and two other properties: the URL of the images that is presented first, and the URL of the rollover image. (This tag automatically generates a rollover script for you; you do not have to create the script.)

Note: when you use the JSP Property Editor, make sure you select the precise item on the JSP canvas for which you want to see properties. Notice how in this example the GO! button is selected, but no action is identified in the Property Editor. That is because the action name is identified in a netui:form that contains this netui:imageButton tag. For example:

```
<netui:form action="toSimple">
  <netui:imageButton
     rolloverImage="resources/images/go-button-rollover.gif"
     url="resources/images/go-button.gif" />
</netui:form>
```

To view the Property Editor values for the form in this example, you must click the netui:form area just outside of the GO! button area. For example:

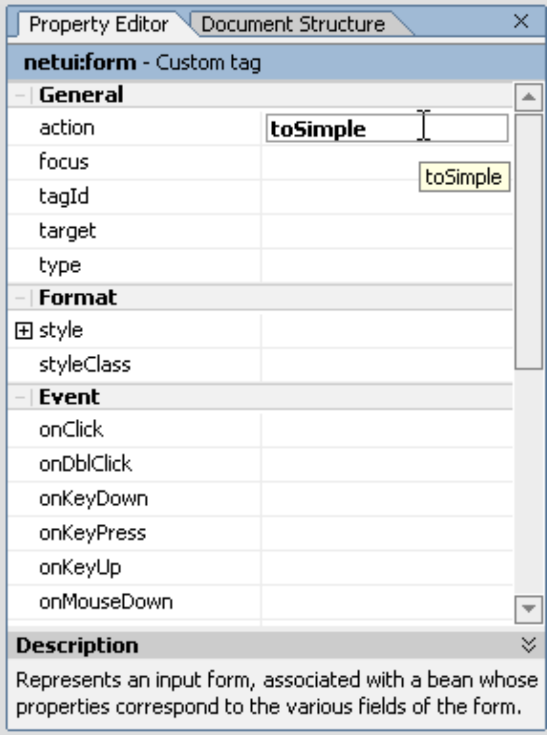


Navigating from page_A to page_B

This feature sample shows simple navigation,

Sources: see /simpleflow.

Now the Property Editor presents the netui:form properties, which includes the action name:



netui:form - Custom tag

General	
action	**toSimple**
focus	
tagId	
target	
type	
Format	
⊞ style	
styleClass	
Event	
onClick	
onDblClick	
onKeyDown	
onKeyPress	
onKeyUp	
onMouseDown	

Description

Represents an input form, associated with a bean whose properties correspond to the various fields of the form.

You can find the installed location of a sample JSP that we referenced in this example here:

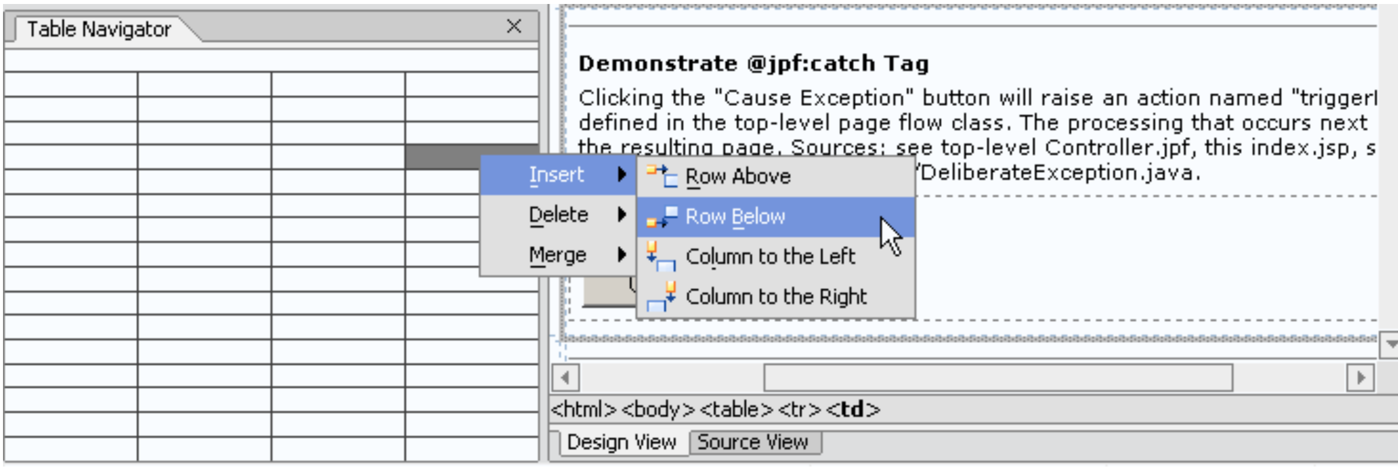
`<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\index.jsp`

Table Navigator Window for JSP Files

Formatting HTML and JSP files is easier if you use tables to create areas on the page. When you have a JSP file open in Design View, you can use the Table Navigator window to see where a selected item is located in a table. You can also use the Table Navigator to add, merge, or delete table cells and rows.

You can make your selection on an item on the JSP canvas, or a cell in the Table Navigator window. In the following sample screen, the fourth row and fourth column of a table were selected from the navigator tool.

Then we used right−mouse click, and displayed the Insert menu.



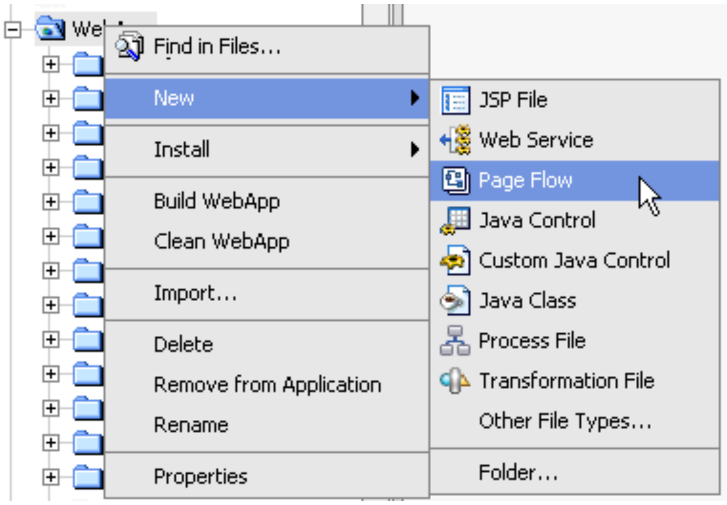
Demonstrate @jpf:catch Tag

Clicking the "Cause Exception" button will raise an action named "triggerI
defined in the top-level page flow class. The processing that occurs next
the resulting page. Sources: see top-level Controller.jpf, this index.jsp, s
/DeliberateException.java.

Page Flow Wizard

To make your job easier, WebLogic Workshop provides a Page Flow Wizard that creates the initial page flow controller class, two or more methods (depending on the type of page flow), and one or more default JSP files.

Before you start the Page Flow Wizard, make sure you have open the appropriate application open and the correct web project selected in the Application window. Then from the top menu in WebLogic Workshop, choose ***File*−−>*New*−−>*Page Flow***. Or right−mouse click and select the Page Flow option from the menu. For example:



For information about the Page Flow Wizard screens, see How Do I: Create a Page Flow?

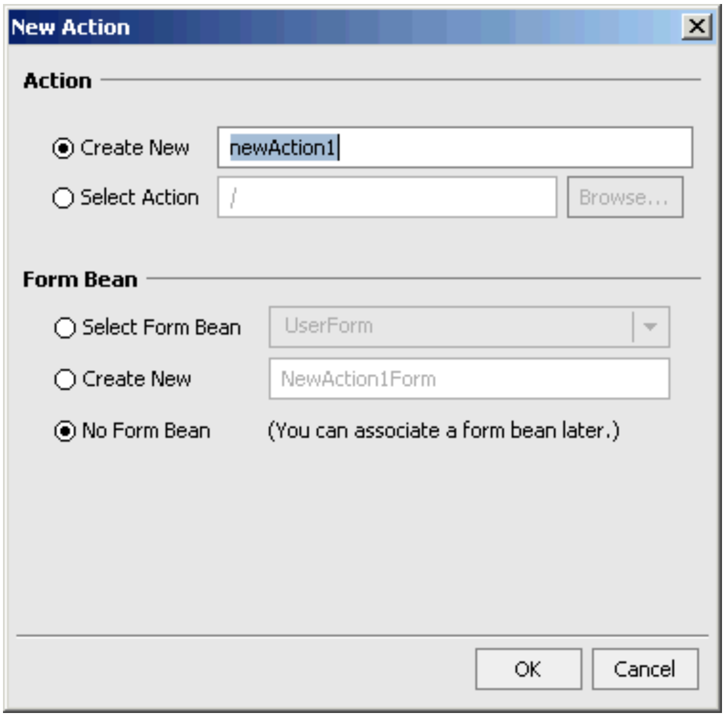
The page flow types that are available with the Page Flow Wizard are:

Basic page flow	If you choose the "Basic page flow" option, the Page Flow Wizard creates a simple page flow that does not use any controls. You can add them later, if needed. For details about starting the Page Flow Wizard, see How Do I: Create a Page Flow?

Page flow from a Java control	If you choose the "Page flow from a Java control" option, you can select one of the existing controls. Then choose Next. On the Select Actions screen, select one or more action methods that are defined by the control. Then choose ***Create***. For details about creating new custom Java controls, see the topic Building Custom Java Controls. Another option on the "Select Page Flow Type" screen is to select the "New RowSet Control" option, to start the server and define a new control.

New Action Window

When a page flow is open, you can drag an action icon from the page flow's palette to the Flow View canvas. WebLogic Workshop presents a New Action window. For example:



● Create New newAction1

○ Select Action /

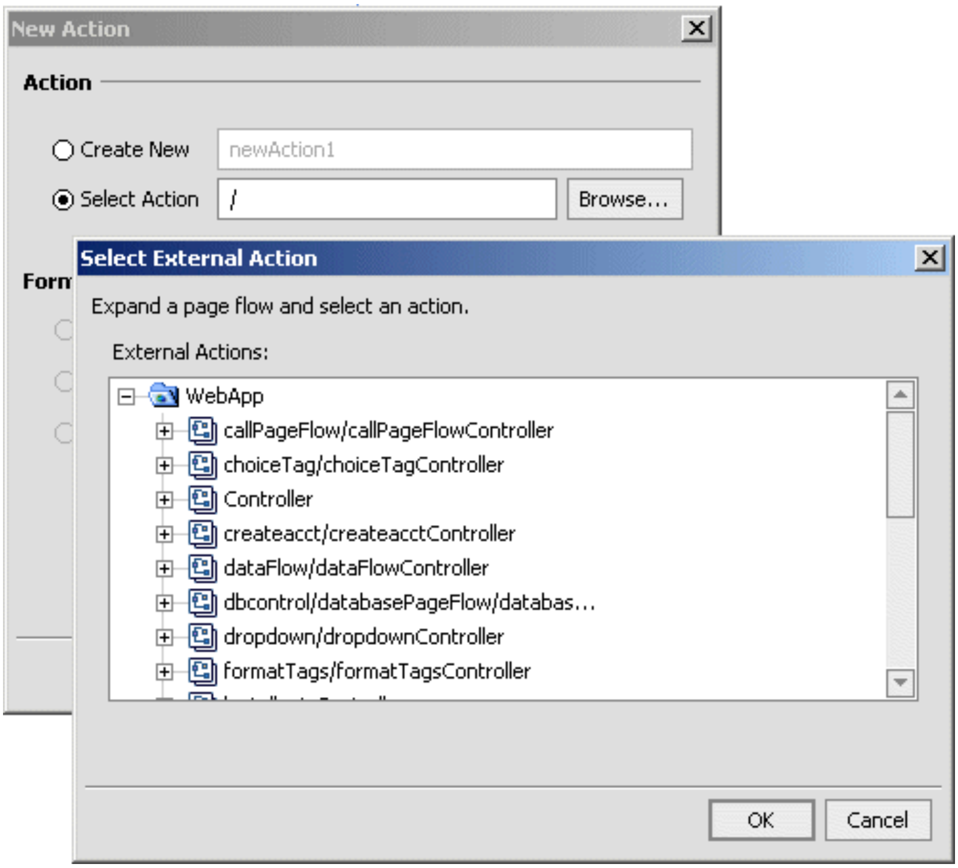
○ Select Form Bean UserForm

○ Create New NewAction1Form

● No Form Bean (You can associate a form bean later.)

You can define a new action, which will define the action method in your page flow class. You can also define a form bean that will be associated with this action. The form bean can be defined an inner class within the page flow class, or you can choose the No Form Bean option. Of course you can add a form bean later, if needed.

With this dialog, you can also browse to find actions, form beans, or both that you have defined previously in this web project. Actions defined in other page flow classes are called external actions. For example, if you choose ***Select Action*** and click ***Browse...***, WebLogic Workshop displays a ***Select External Action*** dialog:

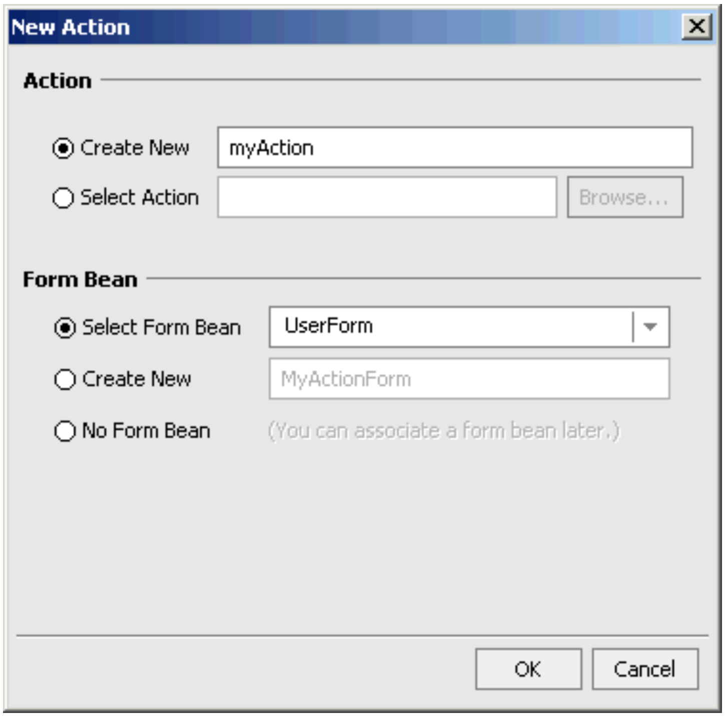


New Action
Action
○ Create New
newAction1
● Select Action
/
Browse...
Forn
Select External Action
Expand a page flow and select an action.
External Actions:
WebApp
callPageFlow/callPageFlowController
choiceTag/choiceTagController
Controller
createacct/createacctController
dataFlow/dataFlowController
dbcontrol/databasePageFlow/databas...
dropdown/dropdownController
formatTags/formatTagsController
OK
Cancel

An additional option with this New Action dialog window is to add an action to the current page flow, and associate this action with an existing form bean in this page flow. For example, in the following sample page flow:

```
<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\callJavaControl\callJavaCont
```

In this page flow, if you drag the Action icon to the callJavaControlController's Flow View canvas, WebLogic Workshop provides a dialog similar to the following. We have already changed the action name to myAction and selected the UserForm.



New Action

Action

● Create New: myAction

○ Select Action

Form Bean

● Select Form Bean: UserForm

○ Create New: MyActionForm

○ No Form Bean (You can associate a form bean later.)

On the Flow View canvas, initially the disconnected icon is as follows:

Initially in the page flow class, WebLogic Workshop adds the following method and annotation:

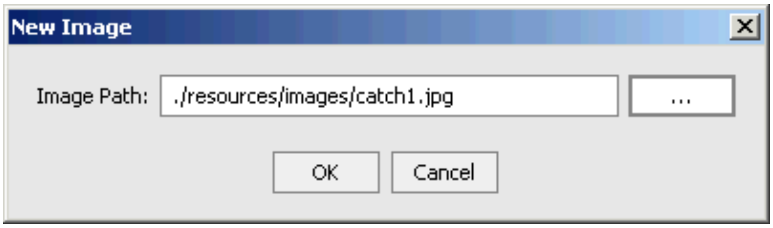
```
/**
 * @jpf:action
 */
 protected Forward myAction(UserForm form)
 {
 return new Forward("success");
 }
```

You can add a tag such as @jpf:forward in the annotations, and conditional logic in the method to indicate what happens when this action is invoked. For more information about the tags that comprise the page flow annotations, see Page Flow Annotations.

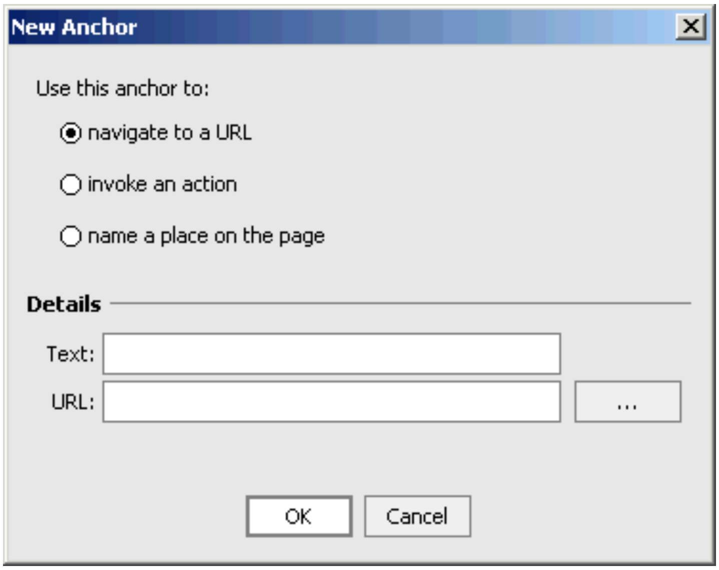
JSP Tag Wizards, Including the Form Wizard

When you drag and drop many of the JSP tags provided by WebLogic Workshop onto the JSP Design View canvas, a dialog window helps you define tag attributes. You can change these attributes later, if necessary.

Some of the dialogs are simple, such as when you drag a netui:image tag from the JSP palette to the canvas:



Others present several options, such as the one for the netui:anchor tag drop:



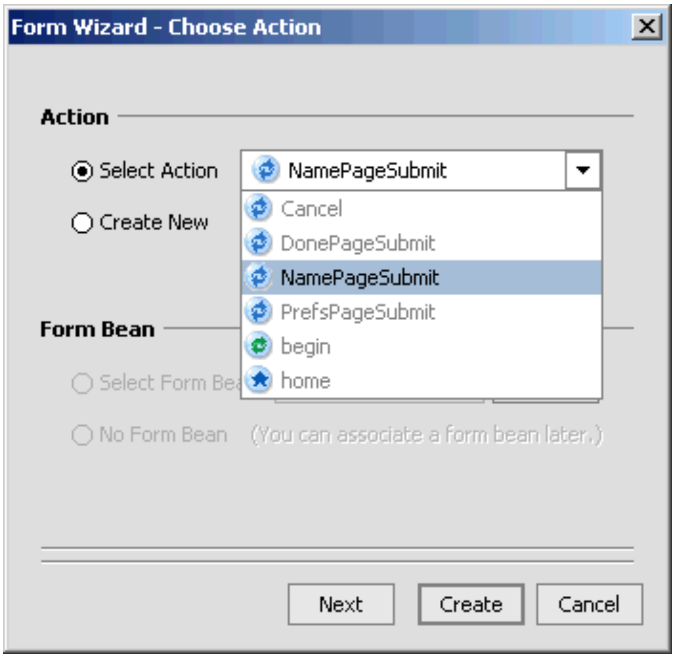
Existing actions that are defined in the page flow class are shown on the drop wizards for those tags that take actions.

You can also use the drop wizards (for JSP tags that have an action attribute) to define new actions that are not yet present in the page flow class. The ***New Action*** dialog is invoked, and after completing that dialog, you are returned to the JSP tag drop wizard. After clicking ***OK***, there is a pending addition on the page flow's Flow View. For example, if on the netui:anchor drop wizard we had clicked ***New...***, and created an action named toPageC, the new action is added to the page flow class, but appears in lighter colors, indicating that it has not yet been added to the page flow class. You can right−mouse click on the "ghosted" action icon and select ***Create*** from the menu.

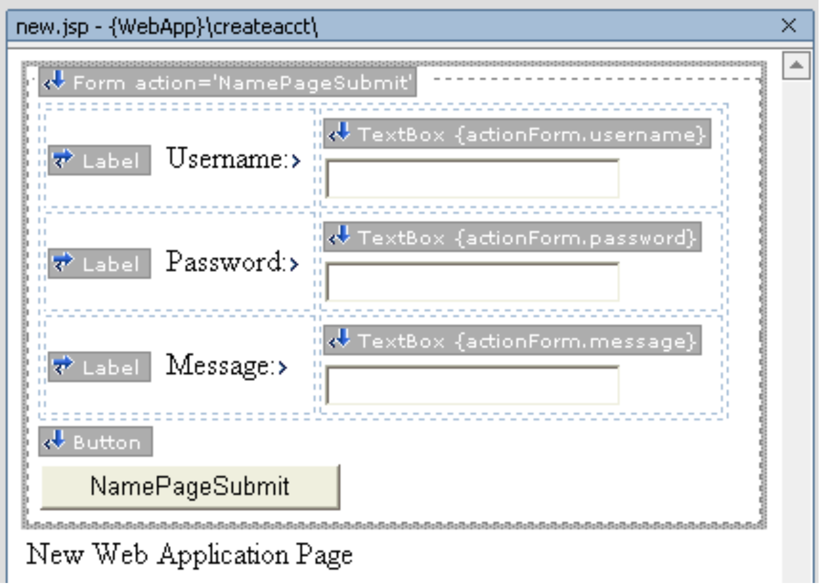
Dragging a netui:form tag from the JSP palette to a JSP Design View canvas invokes the Form Wizard. For example, on a new JSP file in the following sample page flow:

```
<WEBLOGIC_HOME>\samples\workshop\SamplesApp\WebApp\createacct\
```

We dragged the netui:form tag to the (just created) new.jsp file's Design View canvas. On the Form Wizard, we selected the existing NamePageSubmit form bean.



After clicking Create, WebLogic Workshop creates the form for us. The following shows a portion of the JSP Design View canvas:


new.jsp - {WebApp}\createacct\
Form action='NamePageSubmit'
Label
Username:
TextBox {actionForm.username}
Label
Password:
TextBox {actionForm.password}
Label
Message:
TextBox {actionForm.message}
Button
NamePageSubmit
New Web Application Page

For more information about using the Form Wizard, see Tutorial: Your First Page Flow Application.

Related Topics

Guide to Building Page Flows

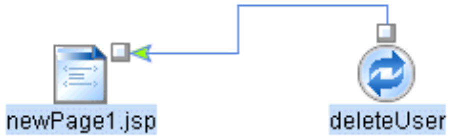
Page Flow and JSP Reference

Summary of Flow View Icons

You can use the page flow's Palette icons and the Flow View to create a graphical representation of a page flow class. The drawing can depict the following:

- The relationship between many components that make up the page flow
- The action methods defined in the page flow
- The actions that are raised in forms on the JavaServer Pages (JSPs)
- The JSP and JPF page flows that are invoked when action methods execute

Drag icons from the Palette, and drop them on the Flow View. To draw a connecting arrow from an Action icon to a JSP Page icon, hover your cursor near the edge of the first icon. When the connecting square boxes appear, move the cursor to the target icon. When the arrow appears, the drawing is similar to the following example:



An arrow from an Action to another entity such as a JSP page, another JPF page flow, or another action method indicates that the result of the source action will load the target.

An arrow from a JSP page icon to an action icon indicates that the action is raised on the JSP page.

If the attempted arrow−drawing operation is invalid, the Flow View editor displays a small circle with a line drawn through it. For example, it does not make sense to connect a "return to action" icon to a JSP file.

Note: An external JSP file is one that resides in a directory outside of the current page flow's directory. On the Flow View canvas, an external JSP file can only be the target of an action forward; that is, the arrow is pointing from the action method's icon to the external JSP file's icon. The external JSP does not actively participate in the current page flow, as a local JSP does. An external JSP does not raise actions outside of its local controller.

For more information about each Flow View icon, see the Related Topics section.

Print and Zoom Options

With the Flow View open, you can use the File > Print... menu options to print a copy of the diagram. You can use the percentage−based Zoom option in the lower−right corner of the Flow View to adjust what size the items in the diagram appear in the window, and in the printed output.

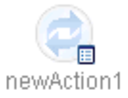
Adding and Removing JSPs from the Flow View Canvas

When you drag a JSP icon from the page flow's Palette to its Flow View canvas, a new JSP file is created in the page flow's directory. If you delete that JSP icon from the Flow View, the JSP file on the file system is deleted. (In the WebLogic Workshop IDE, you are first prompted to confirm the deletion.) Similarly, if you create JSP files in the page flow's directory outside the IDE, those JSP files will show up automatically when you open the page flow in the IDE.

Think of the page flow canvas as a representation of the page flow directory. This notion enforces the concept that the page flow is the combination of the single *.jpf Java controller class file and all the JSP files that reside in the same directory. In the IDE Flow View, you can add, modify, and delete the files that comprise the page flow in its directory.

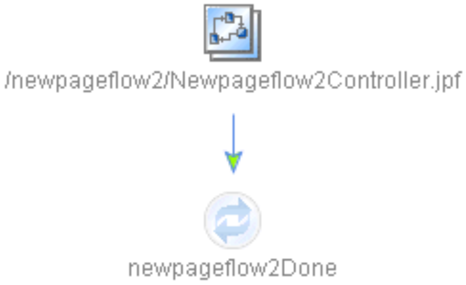
Ghost Icons and Subsequent Steps

In some cases, a page flow's Flow View canvas will display a "ghost" version of an icon, where the icon is shown in lighter colors. For example:



The purpose of a ghost icon is to show you visually that some entity (such as an action, or a page) is referenced in a JPF or JSP file, but either has not been created yet, or has subsequently been deleted in one of the files.

For example, a done() return action that is now referenced on an external nested page flow is not yet implemented in the current page flow. Flow View shows the icon for the external nested page flow, and the ghosted icon for the return action that it now expects to find in the current page flow:



In the example above, to create the expected newpageflow2Done action in the current page flow you can right−mouse click on the ghost icon and select Create.

Here is another example of a scenario that would result in a ghost action icon. An action was raised at one point by a JSP page, and in the page flow controller class you subsequently deleted the action method in the JPF source. The action is represented graphically by a ghost icon on the Flow View because it is still being raised by a JSP page. Note that you cannot delete the ghost icon from the Flow View canvas because, in this case, the JSP page is the source of record on what it is trying to raise. In effect, the ghost icon is a visual reminder that you have an obsolete reference that requires corrective action on your part. In order to delete the ghost action from the Flow View, you would need to go to the JSP page source and delete the NetUI tag that raises this action.

Related Topics

Action Icon

Begin Action Icon

Exit Icon

Page Flow Icon

Page Icon

Return To Action Icon

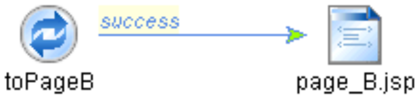
Return To Page Icon

Guide to Building Page Flows

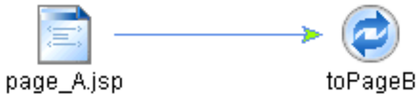
Action Icon

The Action icon represents an action method that you define in the page flow controller class. An action method can perform control logic such as forwarding the user to a new JSP or another page flow. On the Flow View canvas, the direction of the connecting arrow is significant:

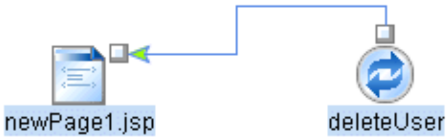
An arrow from an Action to a JSP page or JPF page flow indicates that the result of the action will load the target. For example:



An arrow from a JSP page icon to an action icon indicates that the action is raised on the JSP page. For example:



To draw a connecting arrow from one icon to a related icon (such as from an Action icon to a JSP Page icon), hover your cursor near the edge of the first icon. When the connecting square boxes appear, move the cursor to the target icon. When the arrow appears, the drawing is similar to the following example:



Note: In some cases, the Flow View will display a "ghost" icon, where the icon is shown in a lighter color. For information about ghost icons, see the Ghost Icons and Subsequest Steps section of the Summary of Flow View Icons topic.

Related Topics

Begin Action Icon

Exit Icon

JavaServer Page (JSP) Icon

Page Flow Icon

Return to Action Icon

Return to Page Icon

Guide to Building Page Flows

Begin Action Icon

The Begin Action icon represents this page flow's begin() action method:

It is added automatically for you by the Page Flow Wizard. All page flows must have a begin(). It defines what page is loaded first when the page flow is started. By default, the Page Flow Wizard sets this first page as index.jsp, and creates the file in the page flow's directory. You can modify the begin() method to match the starting functionality required by your page flow.

Related Topics

Action Icon

Exit Icon

JavaServer Page (JSP) Icon

Page Flow Icon

Return to Action Icon

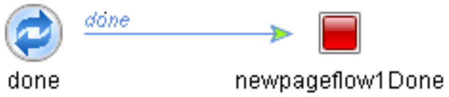
Return to Page Icon

Guide to Building Page Flows

Page Flow and JSP Reference

Exit Icon

The red Exit icon represents an action that will be raised by the current ***nested*** page flow on the nesting page flow, where control will return when this nested page flow is finished. For example:



By default the Page Flow Wizard creates a done() method for nested page flows, as shown in the following code from a page flow *.jpf file:

```
/**
 * @jpf:action
 * @jpf:forward name="done" return−action="newpageflow1Done"
 */
public Forward done()
{
    return new Forward( "done" );
}
```

Typically the "done" action is raised on a JSP in this page flow. For example:

```
<netui:form action="done">
    <netui:button type="submit" value="Done" />
</netui:form>
```

In the code example notice that the attribute "return−action" on the @jpf:forward annotation. This annotation means that when this done() method in the JPF is executed, the page flow runtime:

- Returns to the page flow that called this nested page flow.
- In the calling (or "nesting") page flow, the action specified by the return−action="" attribute is run.

After you have used the Page Flow Wizard to generate a nested page flow, or after dropping an Exit icon onto the Flow View canvas of a nested page flow, you can change the value of the return−action attribute to an actual action method that exists in the nesting page flow.

Related Topics

Action Icon

Begin Action Icon

JavaServer Page (JSP) Icon

Page Flow Icon

Return to Action Icon

Return to Page Icon

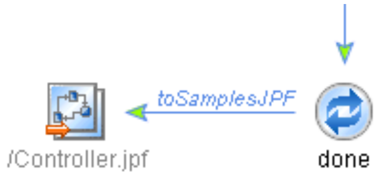
Guide to Building Page Flows

Page Flow and JSP Reference

Page Flow Icon

The Page Flow icon represents another page flow in this web project that will be loaded when an action method runs.

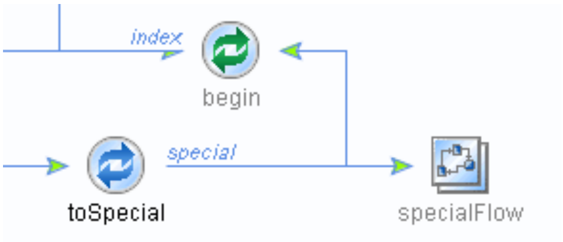
In the following page flow, a done() action method includes an annotation with a path to a /Controller.jpf file that resides in the web application's root directory:



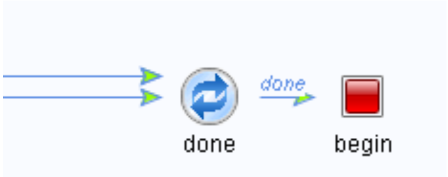
The generated code is as follows:

```
/**
 * @jpf:action
 * @jpf:forward name="toSamplesJPF" path="/Controller.jpf"
 *
public Forward done()
{
    return new Forward( "toSamplesJPF" );
}
```

In some cases the page flow icon will appear without the orange arrow like this:



In this case the page flow represented by the icon returns to the parent flow by calling an action method of the parent page flow, and not by calling the parent JPF file directly. This is graphically represented in the child page flow like this:



The generated code in the child page flow is as follows:

```
/**
 * @jpf:action
 * @jpf:forward name="done" return−action="begin"
```

```
*/
public Forward done()
{
   return new Forward( "done" );
}
```

Notice that return−action is used instead of path in the jpf:forward tag. In the example above the begin method of the parent page flow is called.

Related Topics

Action Icon

Begin Action Icon

Exit Icon

JavaServer Page (JSP) Icon

Return to Action Icon

Return to Page Icon

Guide to Building Page Flows

Page Flow and JSP Reference

JavaServer Page (JSP) Icon

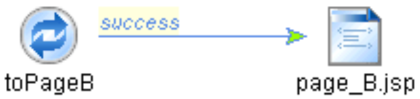
The JSP, or "Page" icon represents:

- A JSP that resides in this (local) page flow directory, and therefore is an active participant in this page flow
- Or an external JSP that is defined in another page flow, but is referenced by the local page flow.

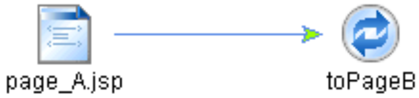
Icons for local and external JSPs can appear on the Flow View icon.

Note: An external JSP file is one that resides in a directory outside of the current page flow's directory. On the Flow View canvas, an external JSP file can only be the target of an action forward; that is, the arrow is pointing from the action method's icon to the external JSP file's icon. The external JSP does not actively participate in the current page flow, as a local JSP could. An external JSP does not raise actions outside of its local controller.

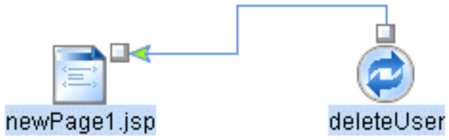
The direction of the arrow is significant. An arrow from an Action to a JSP page indicates that the result of the action will load the JSP. For example:



An arrow from a JSP page icon to an action icon indicates that the action is raised on the JSP page. For example:



To draw a connecting arrow from one icon to a related icon (such as from an Action icon to a JSP Page icon), hover your cursor near the edge of the first icon. When the connecting square boxes appear, move the cursor to the target icon. When the arrow appears, the drawing is similar to the following example:



Note: In some cases, the Flow View will display a "ghost" icon, where the icon is shown in a lighter color. For information about ghost icons, see the Ghost Icons and Subsequest Steps section of the Summary of Flow View Icons topic.

You can find more information about the Palette icons, and the Flow View, Action View, and Source View for page flows by reading the following topics.

Related Topics

Action Icon

Begin Action Icon

Exit Icon

Page Flow Icon

Return to Action Icon

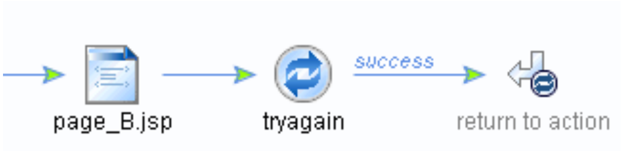
Return to Page Icon

Guide to Building Page Flows

Page Flow and JSP Reference

Return To Action Icon

The Return to Action icon shows that an action method reruns the last action:



For example, if a user enters information on a JSP such as an airport destination, and clicks the submit button triggering an action method that will interpret this information, you can display a matching airport code on the next page with the option to continue, or allow the user to change the airport code and try again. In the last case the previous action can be rerun to interpret the user information.

The corresponding code for the action method in the JPF controller file is:

```
 /**
  * @jpf:action
  * @jpf:forward name="again" return-to="previousAction"
  */
  public Forward tryagain()
  {
     return new Forward( "again" );
  }
```

Related Topics

Action Icon

Begin Action Icon

Exit Icon

JavaServer Page (JSP) Icon

Page Flow Icon

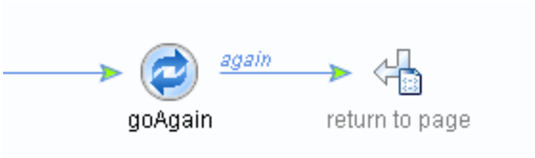
Return to Page Icon

Guide to Building Page Flows

Page Flow and JSP Reference

Return to Page Icon

The Return to Page icon shows that an action method loads the previous page:



For example, if Page1.jsp calls Page2.jsp, and Page2.jsp calls a return−to−page action, navigation returns to Page1.jsp.

The corresponding code for the action method in the JPF controller file is:

```
/**
 * @jpf:action
 * @jpf:forward name="again" return-to="previousPage"
 */
 public Forward goAgain()
 {
    return new Forward( "again" );
 }
```

Related Topics

Action Icon

Begin Action Icon

Exit Icon

JavaServer Page (JSP) Icon

Page Flow Icon

Return to Action Icon

Guide to Building Page Flows

Page Flow and JSP Reference

Changes to Page Flow Features in WebLogic Workshop Releases

This topic outlines changes made to the page flow features in WebLogic Workshop releases. For more details, see the latest WebLogic Workshop Release Notes on the BEA E−docs web site.

Changes from WebLogic Workshop 8.1 to 8.1 SP2

This section summarizes the changes in the WebLogic Workshop 8.1 Service Pack 2 (SP2) release. The prior release was WebLogic Workshop 8.1.

- The supported Struts version changed from Struts 1.1 Release Candidate 1 (RC1), to the released version, Struts 1.1. For more information, see Interoperating with Struts and Page Flows.
- In @jpf:forward annotations, the return−to="action" attribute was renamed as return−to="previousAction", to clarify the purpose of this attribute. After installing SP2, when you access an existing page flow in the IDE, WebLogic Workshop prompts you about the deprecated return−to="action" attributes. If you select the Yes button, the update is made for you automatically.
- In @jpf:forward annotations, the return−to="page" attribute was renamed as return−to="previousPage", to clarify the purpose of this attribute. After installing SP2, when you access each existing page flow in the IDE, WebLogic Workshop prompts you to about the deprecated return−to="page" attributes. If you select the Yes button, the update is made for you automatically.
- In @jpf:forward annotations, SP2 added support for the return−to="currentPage" attribute. For information about the @jpf:forward annotation, see @jpf:forward Annotation.
- SP2 supports a more flexible forward validation process. Instead of using an @jpf:action annotation with an attribute such as validation−error−page="formPage.jsp", SP2 provides a new annotation: @jpf:validation−error−forward. For information, see @jpf:validation−error−forward Annotation.
- After installing SP2, WebLogic Workshop changes any "return−to−page: IDs" and "return−to−action: IDs" entries in the @jpf:view−properties XML to "return−to: IDs". This change happens automatically the first time you open an existing page flow controller class file in the IDE.
- In a web project that uses page flows, WebLogic Workshop prompts you to move each page flow's WEB−INF/jpf−struts−config−*.xml file to a new, recommended location: WEB−INF/.pageflow−struts−generated/jpf−struts−config−*.xml

  This recommended change was designed to help you understand that these jpf−struts−config−*.xml XML files are generated files, and therefore do not need to be maintained in a source file management system along with your application's source files. However, you have the option of entering No in response to the prompt. You may want to delay the migration, for example, if currently you do maintain these generated XML files in a source file management system.
- Several new NetUI tags were added in SP2:
  - <netui−data:declarePageInput> Tag
  - <netui:rewriteName> Tag
  - <netui:rewriteURL> Tag
  - <netui:scriptContainer> Tag

Changes from WebLogic Workshop 8.1 Beta to 8.1

This section summarizes the changes in the WebLogic Workshop 8.1 release. The prior release was the WebLogic Workshop 8.1 Beta.

- The .<pageflow>.lyt file evolved into a @jpf:view−properties annotation that described the page flow's Flow View layout structure
- The @jpf:nestable was replaced by the @jpf:controller nested="true" annotation on the page flow's main class
- @jpf:exceptionHandler −> @jpf:exception−handler (on methods)
- loginRequired (attribute on @jpf:action) −> login−required
- validationErrorPage (attribute on @jpf:action) −> validation−error−page
- returnAction (attribute on @jpf:forward) −> return−action
- returnTo (attribute on @jpf:forward) −> return−to
- returnFormType (attribute on @jpf:forward) −> return−form−type
- returnForm (attribute on @jpf:forward) −> return−form
- messageKey (attribute on @jpf:catch) −> message−key

Related Topics

Guide to Building Page Flows

Page Flow and JSP Reference