Web Applications

Workshop for WebLogic provides support for building web applications using the Beehive NetUI framework.

These topics introduce you to the basic concepts behind Beehive NetUI-based web applications.

Current Release Information:

- What's New in 9.2
- Upgrading from 8.1

Useful Links:

- Tutorials
- Tips and Tricks

Other Resources:

- Online Docs
- Dev2Dev
- Discussion Forums
- Development Blogs

Topics Included in This Section

Tutorial: Accessing a Database from a Web Application

This tutorial shows you how to build a web application that communicates with a backend database.

Tutorial: Beehive NetUI / Java Server Faces Integration

This tutorial shows you how to add Java Server Faces pages to your web application and how to integrate Java Server Faces and Beehive NetUI technologies in one application.

Introduction to Beehive NetUI

This topic introduces you to the basic concepts behind Beehive NetUI, the web application framework used by Workshop for WebLogic.

The Page Flow Perspective

This topic sets out the tooling provided by Workshop for WebLogic for building web applications.

Integrating Java Server Faces into a Beehive NetUI Web Application

This topic explains how to integrate JSF and Beehive NetUI technologies in one application.

Beehive Implementation Details

This topic lists the different web applications used by Workshop for WebLogic.

Authoring Beehive NetUI JSPs

These topics introduce you to the basic techniques for creating JSP pages with Workshop for WebLogic.

Web Application Dialogs

These topics explain the web application related UI dialogs and wizards.

Tutorial: Accessing a Database from a Web Application

What This Tutorial Teaches

This tutorial teaches you how to build a web application capable of accessing a database using BEA Workshop for WebLogic Platform. It also forms a general introduction to the web application and control technologies used by Workshop for WebLogic.

> **Note:** This tutorial requests that you create a new workspace; if you already have a workspace open, this will restart the IDE. Before beginning, you might want to launch help in standalone mode to avoid an interruption the restart could cause, then locate this topic in the new browser. See Using Help in a Standalone Mode for more information.

The tutorial contains step-by-step instructions for building a simple web application for managing a customer database. As you progress through the tutorial you will learn:

- how Workshop for WebLogic leverages Beehive technologies to simplify web application development
- how to use Java Controls to encapsulate access to data resources
- how to make web applications and controls work together
- how controls can be used to access data stored in a database
- how to display complex Java objects as simple HTML tables

Tutorial Synopsis

Step 1: Create an EAR Project and a Web Application Project

The first step of this tutorial you will create two projects: an EAR project and a Web Application Project which contains a default minimal page flow. You will define a server which is running a sample database and library modules.

By the end of the first step, your application consists of the following components:

Step 2: Add a Page Flow and a Control

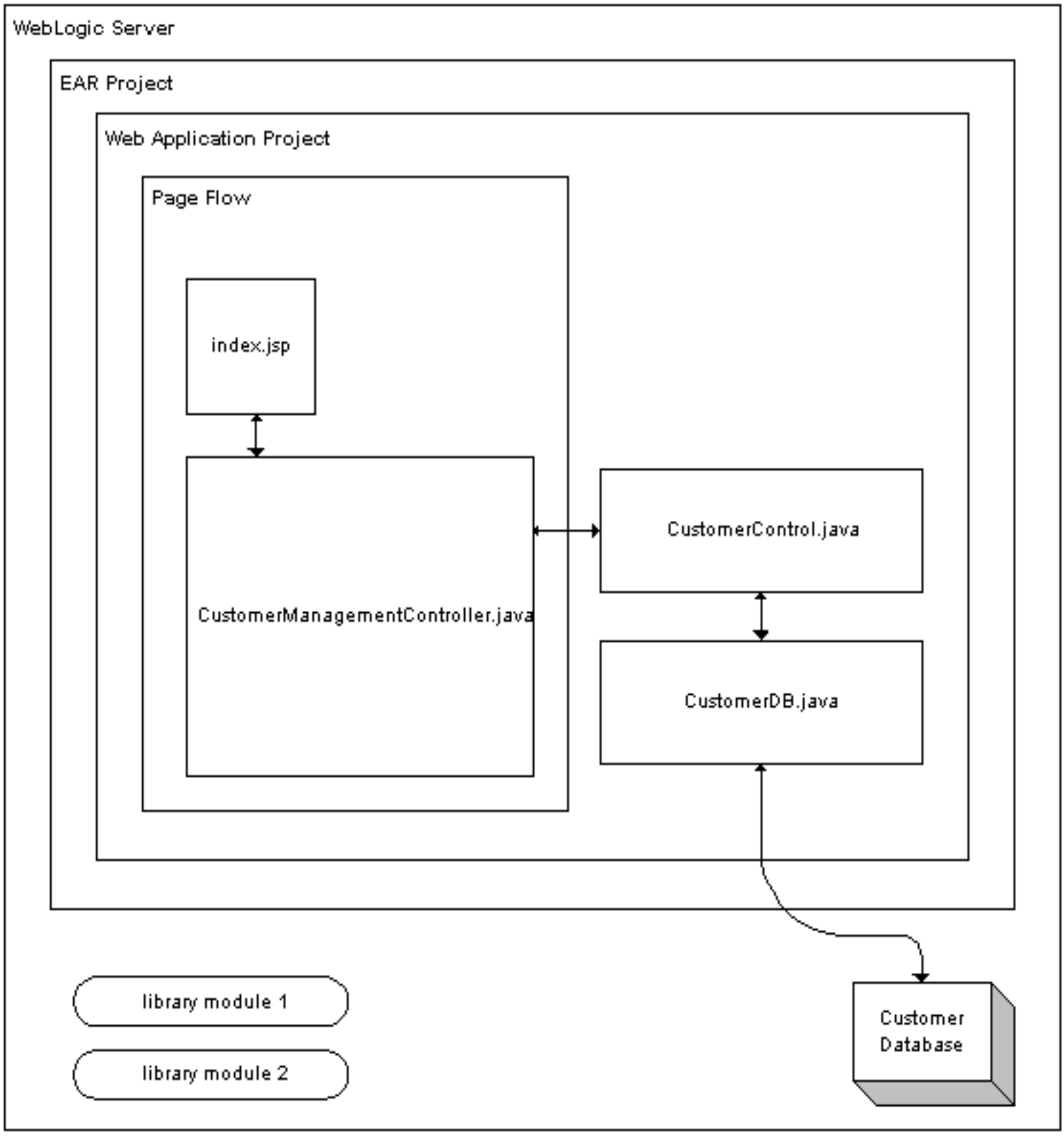
In the second step, you will add a Page Flow, and two controls to the web application project.

Page Flows are user-facing components of a web application. A Page Flow consists of any number of JSP pages and a single Java class, called a **Controller class**, that handles user actions and events inside the application.

The two controls used in this tutorial allow your application to interact with a database. The first control (CustomerControl.java) is a custom Java control. The second control (CustomerDB.java) is a database control that queries the database directly. Strictly speaking, a web application needs only one control, a database control, to access a database; but two controls are used here (a database control with a wrapper custom control) to increase the modularity of the application.

Details about this modularity are provided in step 2 of this tutorial.

At the end of step 2, your application will consists of the following components:



Step 3: Create a Data Grid

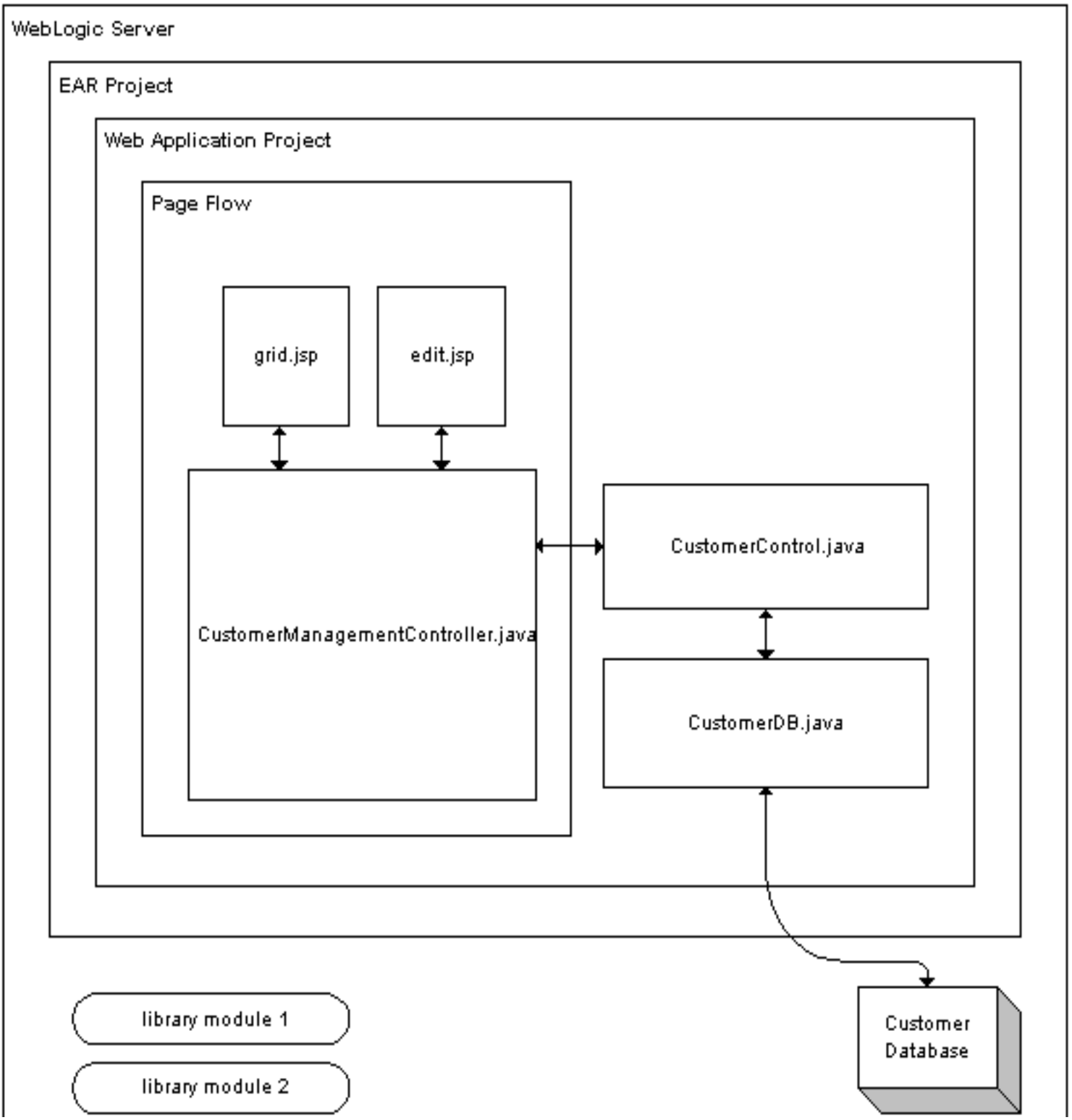
In the third step you add a data grid to a JSP page that will display the data in the database.

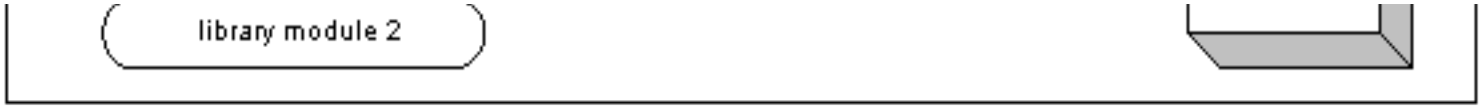
The components work together as follows: a method in the Page Flow Controller class will call the custom control, which will call the database control, which finally will query the database. The results returned by the query will then be displayed by the data grid on the JSP page.

Step 4: Create a Page to Edit Customer Data

In the last step you add an edit page to the Page Flow allowing you to edit the data in the database.

When the application is complete, it appears as follows:





Click the arrow below to navigate through the tutorial:

Step 1: Create an EAR Project and a Web Application Project

In this step you will create two projects: an EAR project and a Web Application project. These are the basic building blocks required for designing and testing a new Workshop for WebLogic web application.

An EAR project configures and stores resources for other components that belong to it, components such as web applications, EJBs, databases, etc. An EAR project has two main roles: (1) It is a composite project made up of other projects, such as web projects, EJB projects, and others. (2) Is it a resource project containing library modules and JARs which other projects utilize.

The web application project you create belongs to the EAR project.

The tasks in this step are:

- To Start Workshop for WebLogic Platform
- To Create a New Web Project and a New EAR Project
- To Import Files into the Web Project
- To Add a WebLogic Server Domain

To Start Workshop for WebLogic and Create a New Workspace

If you haven't started Workshop for WebLogic yet, use these steps to do so.

... on Microsoft Windows

If you are using a Windows operating system, follow these instructions.

- From the **Start** menu, click **All Programs > BEA Products > Workshop for WebLogic Platform 9.2**

...on Linux

If you are using a Linux operating system, follow these instructions.

- Run BEA_HOME/workshop92/workshop4WP/workshop4WP.sh

(If you have already started Workshop for WebLogic, select **File > Switch Workspace**.)

1. In the **Workspace Launcher** dialog, click the **Browse** button.

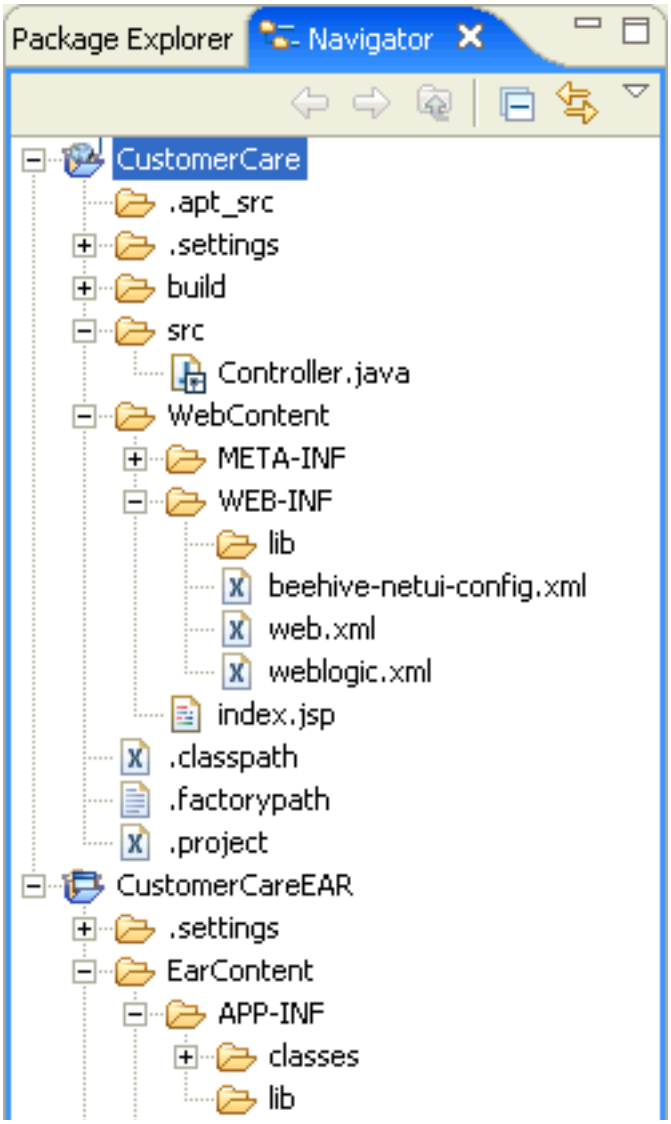
2. In the **Select Workspace Directory** dialog, navigate to a directory of your choice and click **Make New Folder**.
3. Name the new folder `WebAppTutorial`, press the **Enter** key and Click **OK**.
4. In the **Workspace Launcher** dialog, click **OK**.

To Create a New Web Project and a New EAR Project

1. From the **File** menu, select **New** > **Project**.
2. In the **New Project** dialog, on the **Select a wizard** page, select the node **Web> Dynamic Web Project**.
   Click **Next**.
3. In the **Project Name** field, enter `CustomerCare`
   Place a check mark next to **Add project to an EAR**.
   Confirm that the EAR Project Name is **CustomerCareEAR**.
4. Click **Finish**.

When you web project is first created, it is displayed in the **Package Explorer** view by default. The Package Explorer view shows a logical view of your workspace and its JAR resources.

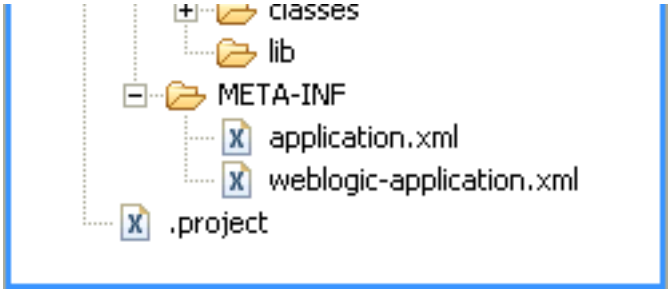
The image to the right shows your workspace in the Navigator view. To switch to the Navigator view select **Window > Show View > Navigator**. The Navigator view shows your workspace as it is saved on disk.



There are now two projects in your workspace: the web project **CustomerCare** and the EAR project **CustomerCareEAR**. The two projects *appear* as sibling projects, since they are on the same level of the directory tree. But when the projects are compiled and deployed, the EAR project CustomerCareEAR is really a container project for the web project customerCare.

The CustomerCare Web Project

- The **.settings** folder: Any directory that begins with a "." contains code generated by Workshop for WebLogic. You should not edit the files in this directory.
- The **build** folder contains .class files and other



compiled code. You should not edit the files in this directory.

- The **src** folder contains the web project's JAVA files. These files are user editable.
- The **WebContent** folder contains the web project's JSP files and other web-related resources, such as configuration files (in the **WEB-INF** folder).

The CustomerCareEAR EAR Project

- The **.settings** and **build** folders are described above.
- The **EarContent** folder contains configuration files for the EAR project.

  EarContent/APP-INF/lib: Any JARs in this directory are available to any project referenced by the EAR project.

  EarContent/META-INF/application.xml: Lists the modules referenced by the EAR, for example, the web application customerCare.

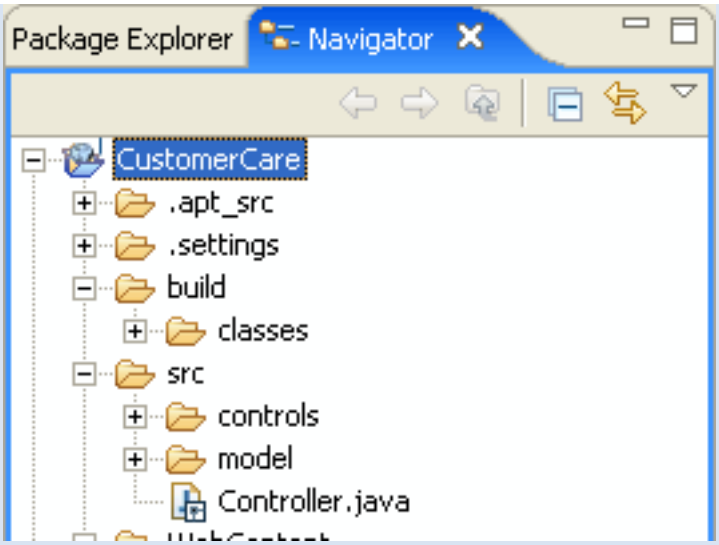
  EarContent/META-INF/weblogic-application.xml: List the library modules referenced by the EAR project. These resources can be used by any module referenced by the EAR.

To Import Files into Your Web Project

In this step you will import control files into your web project--control files that provide access to a customer database. An alternative design would locate these controls in a utility project (File > New > Project > J2EE > Utilty Project), which would make the controls available to all projects in the workspace. But for the sake of simplicity and expediency we have placed the controls directly in the web project.

1. On the **Navigator** tab, open the **CustomerCare** folder.
2. Open Windows Explorer (or your operating system's equivalent) and navigate to the directory **BEA_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples_9.2.1/tutorials/resources/webApp/**

   Watch Out! Don't open the **webService** folder!
3. Drag the folders **controls** and **model** (located at **BEA_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples_9.2.1/tutorials/resources/webApp/**) into the **Navigator** tab directly onto the folder **customerCare/src**.
4. Confirm that the following directory and file structure exists before proceeding.

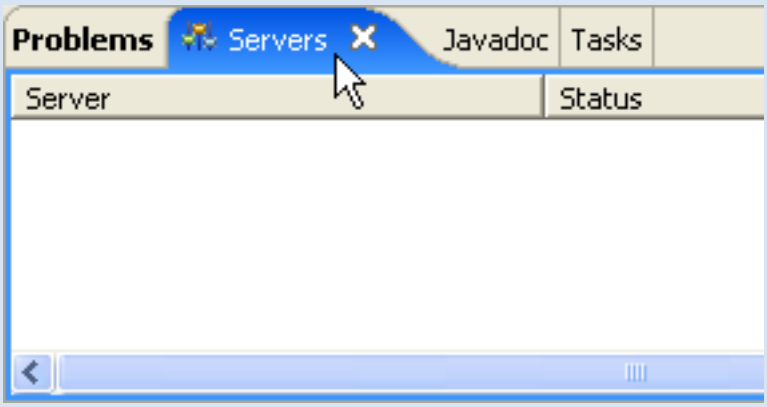


To Add a WebLogic Server Domain

In this step you will point to a server where you can deploy your application.

Note: if you have executed this tutorial before your server may already contain previous deployments of the tutorial-related projects. Before proceeding, it is recommended that you either (1) remove previous tutorial code from your server or (2) create a new server domain.

1. Confirm that you are in the Workshop perspective (**Window > Show Perspective > Workshop**).
2. Click the **Servers** tab.



3. Right-click anywhere within the **Servers** tab, and select **New > Server**.
4. In the **New Server** dialog, select **BEA Systems Inc.**> **BEA WebLogic v9.2 Server**. Click **Next**.
5. In the **Domain home** dropdown, select the location **BEA_HOME/weblogic92/samples/domains/workshop**. (Note: if you are using a newly created server domain for this tutorial, then use the **Browse** button to navigate to that server domain, for example, BEA_HOME/user_projects/domains/base_domain.) Click **Next**.

6. In the **Available projects** column, select **CustomerCareEAR**. Click the **Add** button to move the select project to the **Configured projects** column.
7. Click **Finish**.

   A new server is added to the Servers tab.

You can use the Servers tab to manage your servers and project deployments as you develop you applications.

To deploy or undeploy a project from a server, right-click the server and select **Add and Remove Projects**.

For a more properties, double-click a server.

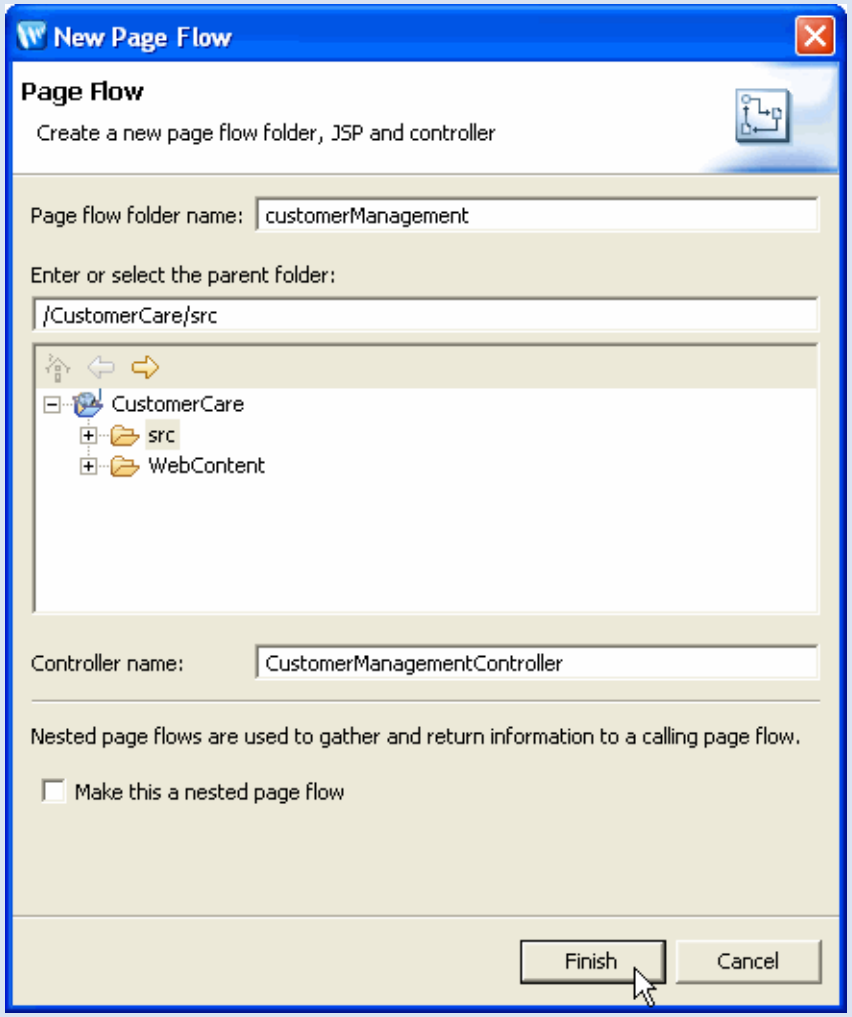
Click one of the following arrows to navigate through the tutorial:

Step 2: Add a Page Flow and a Control

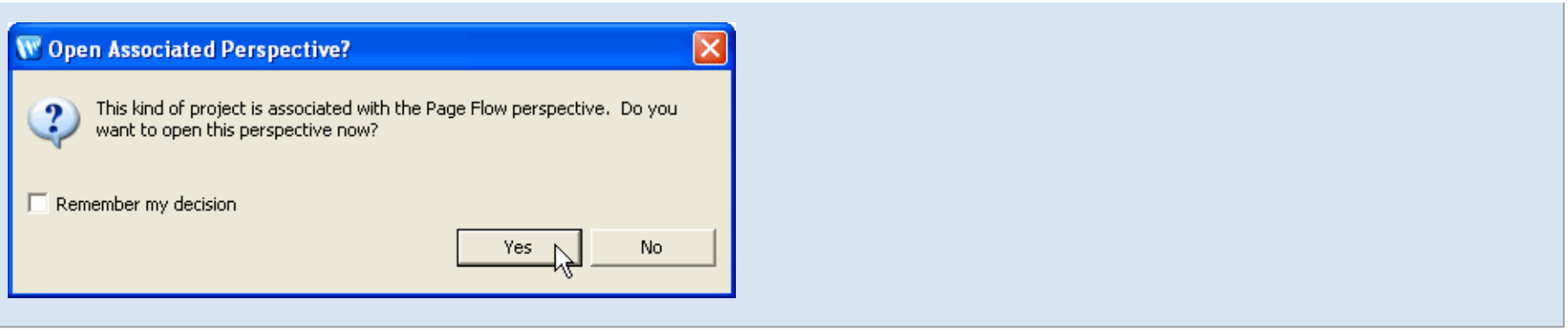
The tasks in this step are:

Create a New Page Flow

1. Right-click on the **CustomerCare** project and select **New** > **Page Flow**.
2. In the **New Page Flow** dialog, in the field **Page Flow folder name** enter `customerManagement` and click **Finish**.



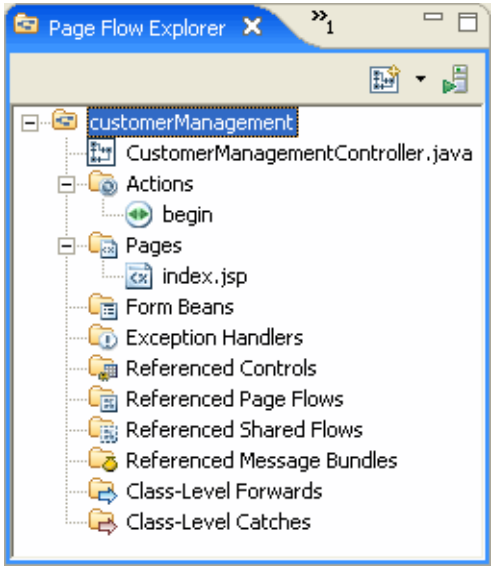
3. When asked to open the Page Flow perspective, click **Yes**.



When you add a new Page Flow, it is displayed in the **Page Flow Perspective** by default. The Page Flow Perspective gives you three different views on a particular Page Flow:

1. Page Flow Explorer
2. Page Flow Editor
3. Source Editor

Page Flow Explorer



The **Page Flow Explorer** shows a logical view of the current Page Flow, listing all of the Actions, JSP Pages, Form Beans, etc. contained in the Page Flow. The Page Flow Explorer depicts the properties in a way similar to a file tree. But it is important to note that this tree is *not* the way that the Page Flow is written to disk. (To see the actual file tree of a Page Flow as it is written to disk, switch to the **Navigator** view.)

The top-level node gives the package of the current Page Flow. In this case the package is **customerManagement**.

The first child node gives the Page Flow Controller class being viewed, in this case, **CustomerManagementController.java**.

The next node lists the Actions. In this case there is only one action: **begin**. This action is created by default with each new Page Flow.

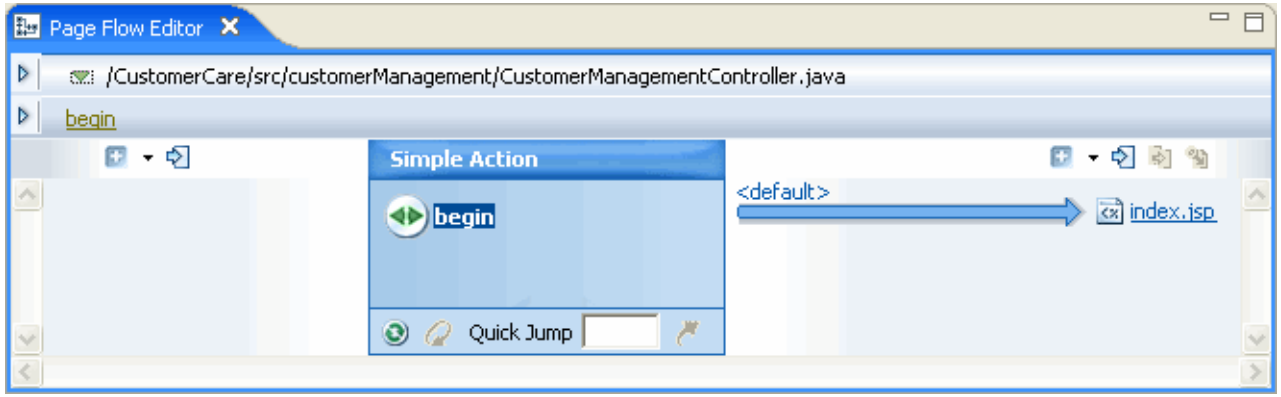
Next the JSP pages are listed. There is only one JSP page at this time: **index.jsp**.

At this time, all of the other nodes are empty, because our Page Flow is relatively undeveloped. As we proceed we'll add items to the nodes.

Page Flow Editor

The **Page Flow Editor** gives a graphical view of the current Page Flow.

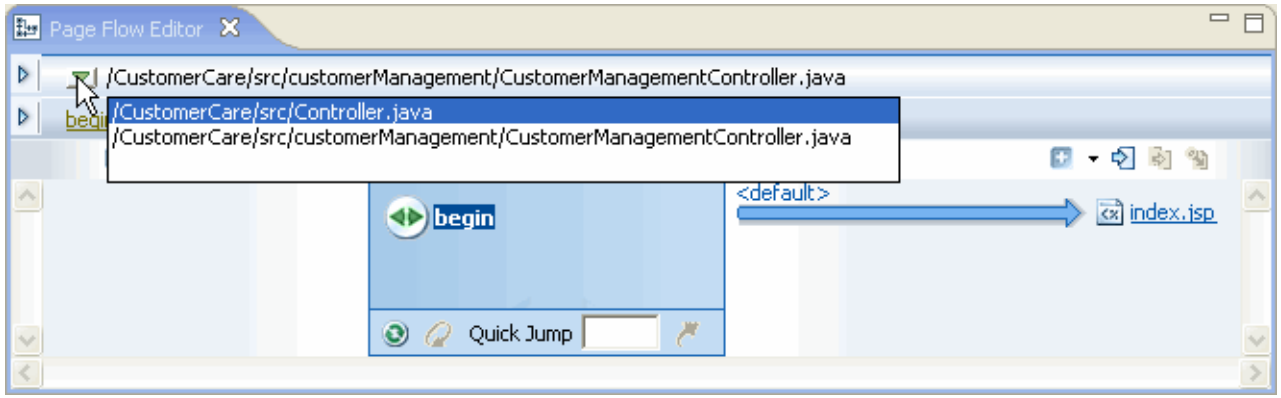
The graphical view depicts the Page Flow's actions, JSP pages, and the connections between the actions and pages. In the picture below, the begin action is shown in the **center pane**. An arrow extending from the begin action to the index.jsp page depicts the Forward that navigates users to index.jsp, whenever the begin action is called.



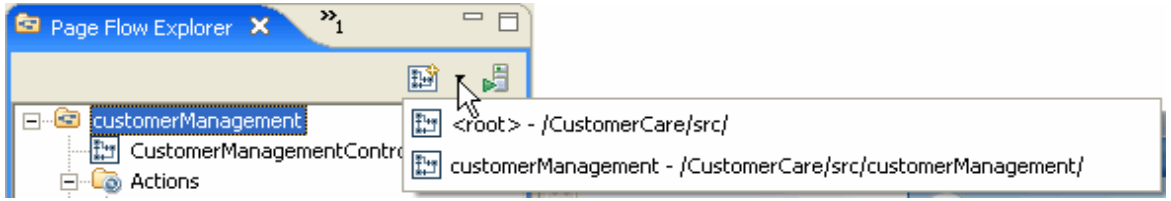
The left side of the pane is called the **upstream pane** and the right side is called the **downstream pane**. Note that the Page Flow Editor always depicts the direction of flow as starting from the left and progressing to the right.

To change the current Page Flow depicted, click the dropdown list marked by the green triangle, as shown below.

As you can see from the dropdown list shown below, there are two Page Flows in the web application: (1) Controller (a default Page Flow created with each web application) and (2) CustomerManagementController (which you will be developing for the remainder of this tutorial).



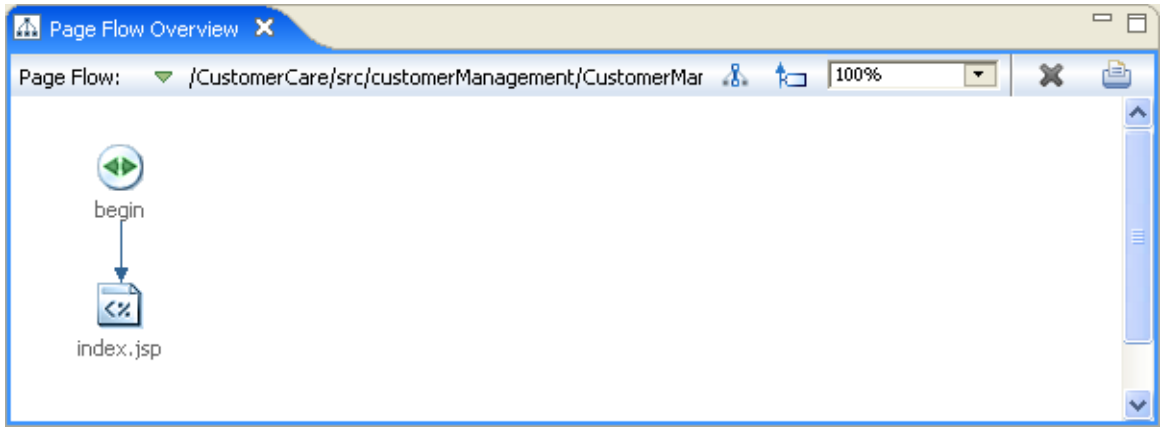
You can also access a list of available Page Flows by clicking the icon on the **Page Flow Explorer** tab. In the image below the icon is circled in red. (The icon directly to the right will pop up the new Page Flow wizard.)



Page Flow Overview

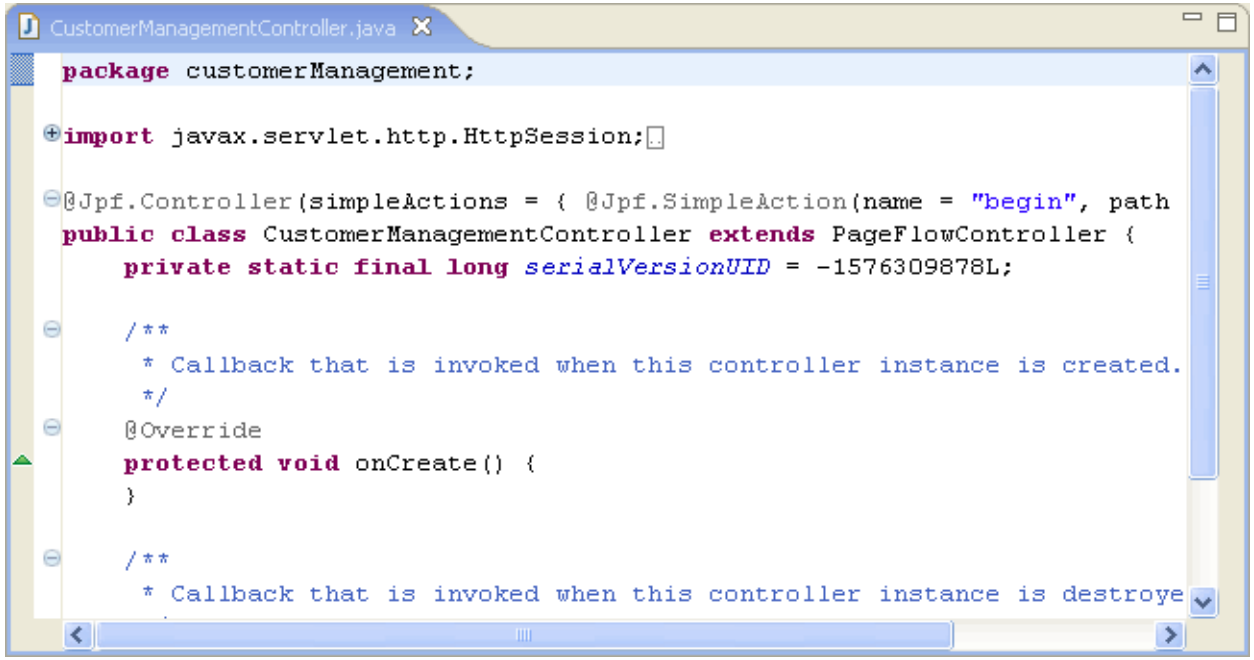
The **Page Flow Overview** gives a graphical summary of a page flow. In shows all of the actions, pages, and the relationships between them.

Double-clicking on an icon in the **Page Flow Overview** shows the associated source code in **Source View**.



Source Editor

The source editor, appearing directly underneath the Page Flow Editor view, shows the Java source for the Page Flow Controller class.



To Add a Control to the Page Flow

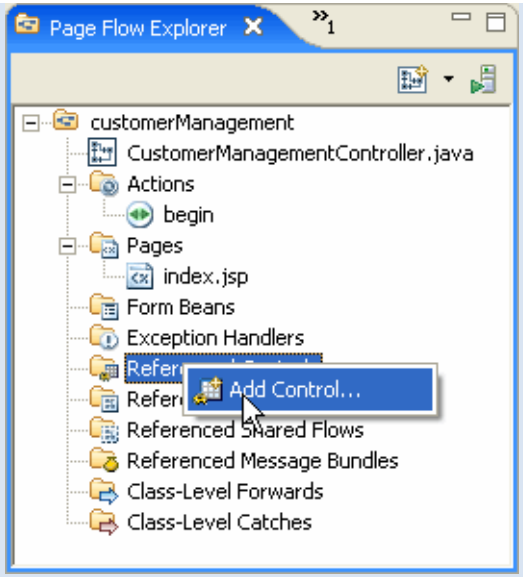
In this step, you will add a control, **CustomerControl.java**, to the Page Flow. The methods of this control (**addCustomer**, **editCustomer**, etc.) allow the Page Flow client to interact with customer data in a database. The interaction between the Page Flow client and the database consists of three classes:

1. **CustomerManagementController.java**: the client Page Flow class
2. **CustomerControl.java**: a wrapper intermediary Control class
3. **CustomerDB.java**: the Database Control class, queries the database directly

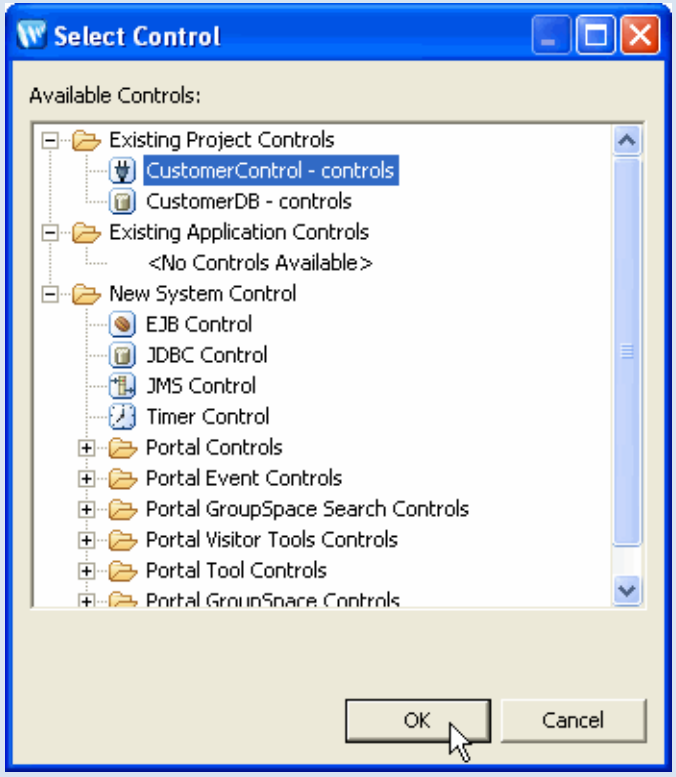
The control CustomerControl.java acts as a wrapper intermediary class between the client, CustomerManagementController.java, and the Database Control CustomerDB.java. This wrapper intermediary increases the modularity of the application, allowing the user (1) to switch to a different Database Control if necessary in the future and (2) to execute any data type recasting within the wrapper class.

In this tutorial no actual recasting occurs, but it is easy imagine a case where recasting is necessary. For example, suppose your Page Flow expects a Customer[] object but your Database Control returns an ArrayList of Customer objects. In such a situation you could use the intermediary wrapper class to load the ArrayList into a Customer[] before passing the data to the Page Flow.

1. On the **Page Flow Explorer** tab, right-click on the **Referenced Controls** node and select **Add Control**.



2. In the **Select Control** dialog, select **Existing Project Controls** > **CustomerControl - controls**. Click **OK**.



You have just added four lines of code to the Page Flow Controller class **CustomerManagementController.java**:

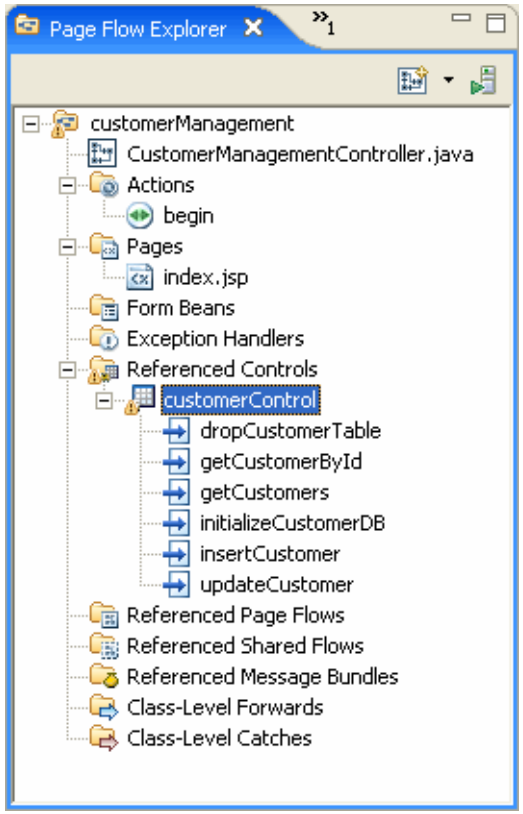
```
import org.apache.beehive.controls.api.bean.Control;
import controls.CustomerControl;

    ...

@Control
private CustomerControl customerControl;
```

These lines declare the **Customer** control on the Page Flow, allowing you to call control methods.

When you declare a control on a Page Flow class, it appears in the **Referenced Controls** node, along with a list of its available methods:



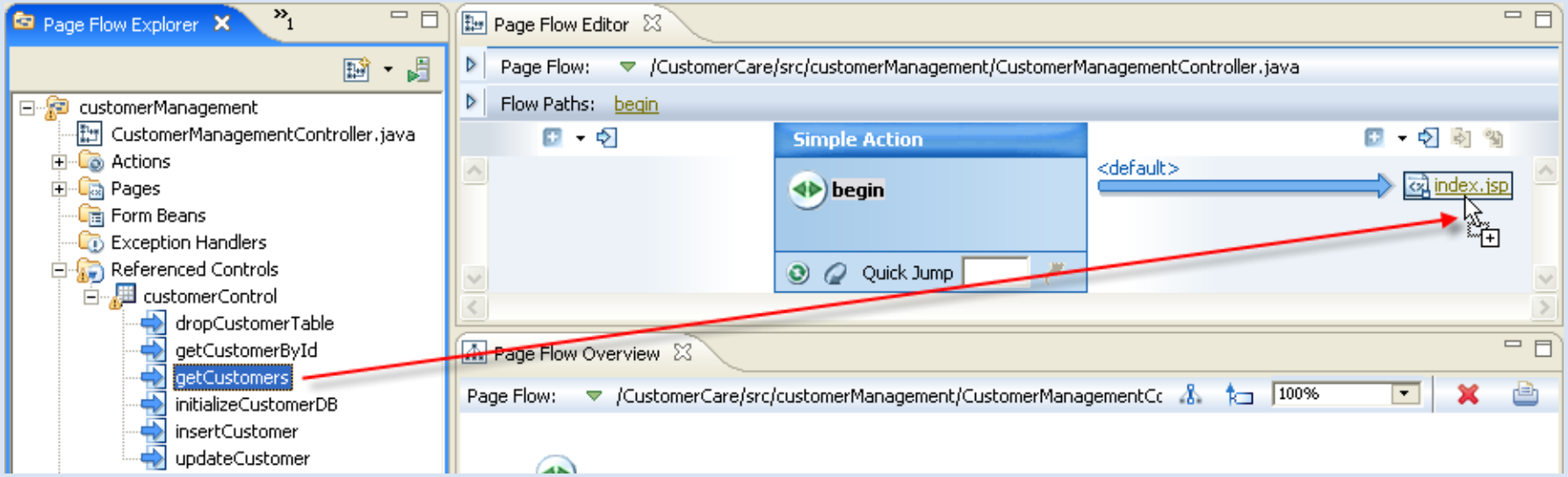
To Create an Action to Forward Data to a JSP Page

In this task you will edit the Page Flow class so that it retrieves customer data from the CustomerControl. In particular, you will add an Action (i.e., an annotated method called getCustomers()) to the the Page Flow class that calls the CustomerControl method getCustomers(), a method which returns an array of Customer objects. (In the next step you will create a JSP page that displays this array of Customer objects, rendering it as an HTML table.)

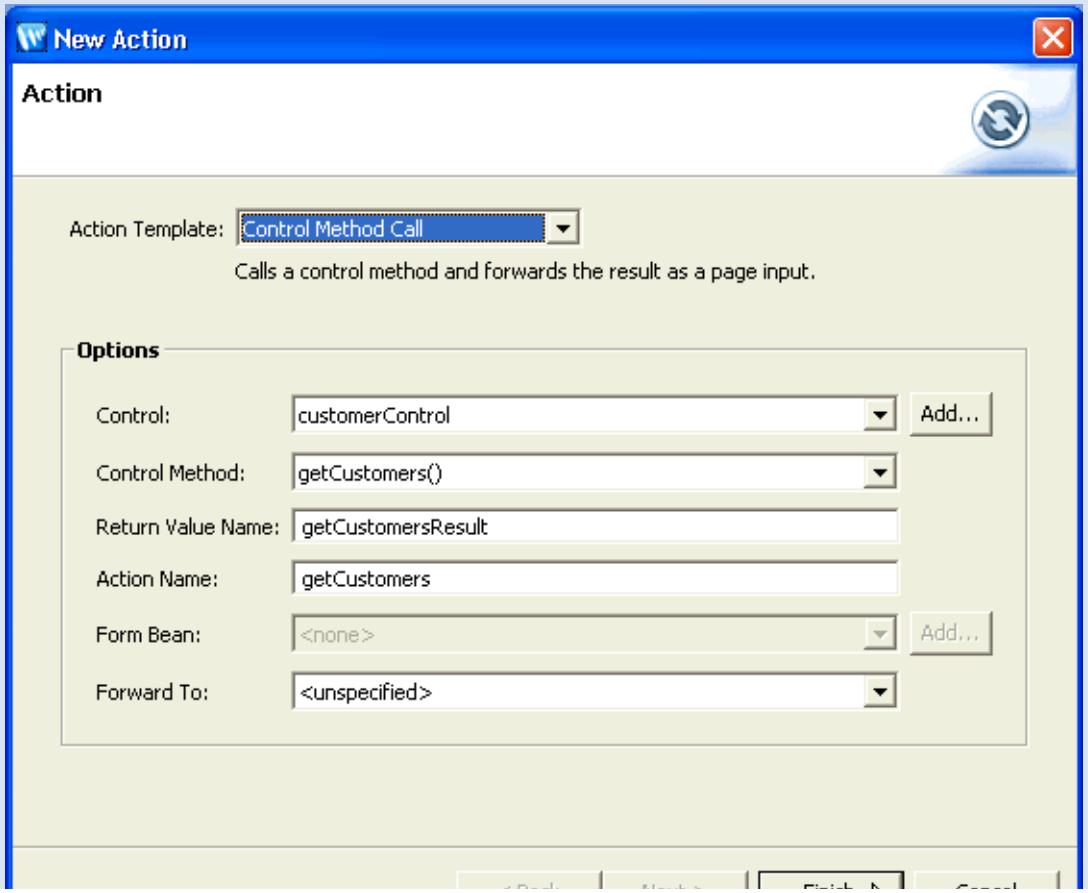
1. On the **Page Flow Explorer** tab, open the node **Referenced Controls > customerControl** and, within that node, locate the **getCustomers** method.
2. Drag the **getCustomers** method directly on top of the **index.jsp** page displayed on the right-hand side of the **Page Flow Editor** tab.

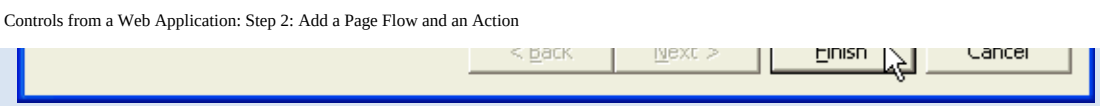
   Note: Make sure that the Page Flow **CustomerManagementController** is displayed in the **Page Flow Editor**. If any other Page Flow is displayed, select

CustomerManagementController from the dropdown list (click the green triangle to show the dropdown list).

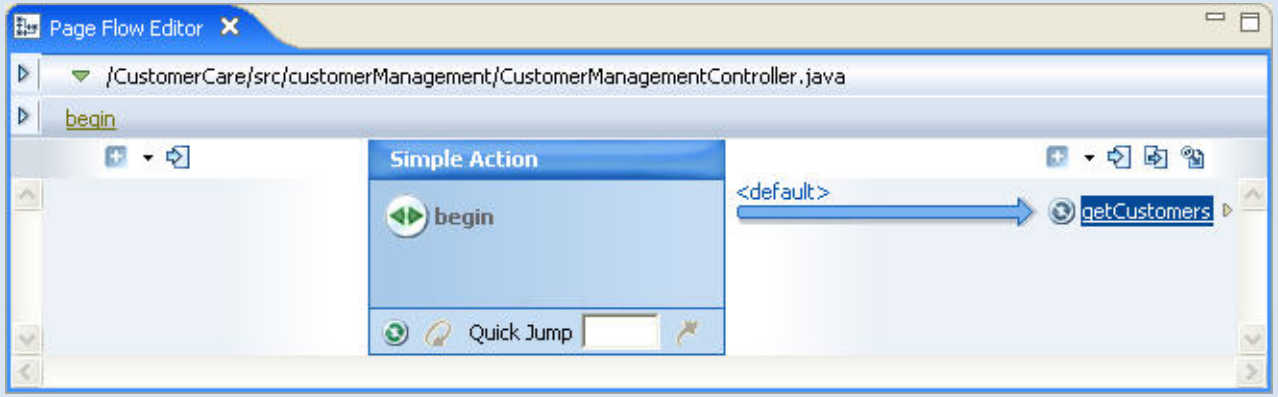


3. In the **New Action** dialog, accept the defaults and click **Finish**.





When the dialog closes, the **Page Flow Editor** should appear as follows:



You have now created a new Page Flow Action **getCustomers()** that calls the Control method **getCustomers()**. The source code of the Action looks like this:

```
    @Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "", actionOutputs = { @Jpf.ActionOutput(name = "getCustomersResult",
type = model.Customer[].class) }) })
    public Forward getCustomers() {
        Forward forward = new Forward("success");
        model.Customer[] getCustomersResult = customerControl.getCustomers();
        forward.addActionOutput("getCustomersResult", getCustomersResult);
        return forward;
    }
```

4. On the **Page Flow Explorer** tab, open the **Pages** node, right-click **index.jsp** and select **Delete**.
   In the **Confirm Remove** dialog, click **Yes**.
5. Press **Ctrl+Shift+S** to save your work.

Related Topics

None.

Click one of the following arrows to navigate through the tutorial:

Step 3: Create a Data Grid

In this step you will add a data grid to your application. A data grid is a set of JSP tags that are designed to render data as an HTML table. This is especially useful for rendering database data: the data grid renders the database fields as columns of the table and it renders the database records as rows of the table.

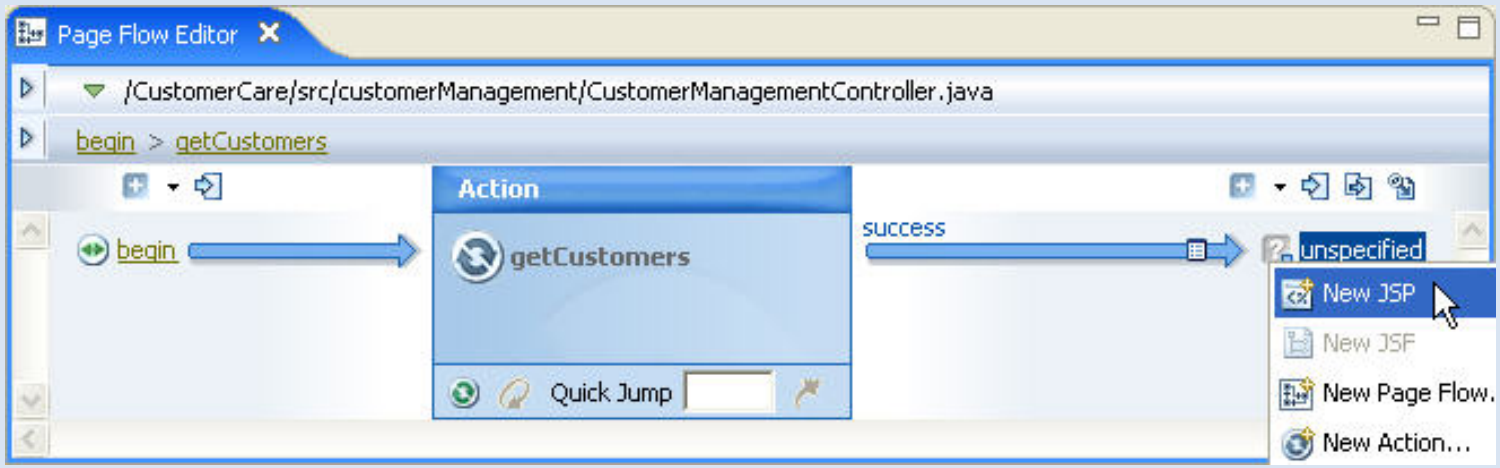
The tasks in this step are:

Create a JSP Page to Display the Customer List

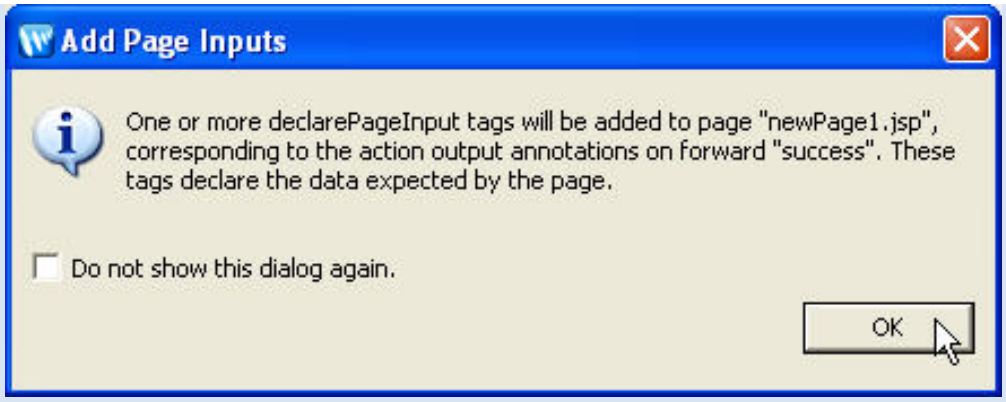
In this step you will create a new JSP page and place it within the navigation scheme of your Page Flow: when the getCustomers() action is called, the user is navigated to this JSP page.

1. On the **Page Flow Editor** tab, click on the **getCustomers** action icon to center the node.
2. Right-click on the **unspecified node** and select **New JSP Page**.

   The unspecified node means that the action getCustomers does not forward to any specified JSP page or other action. Your page flow will compile if it contains unspecified nodes, but, at runtime, if the getCustomers action is ever called, an exception will be thrown. (In terms of the source code, an unspecified node depicts an empty string in the path attribute of a Forward object: `@Jpf.Forward(name = "success", path = "" )`.



3. On the **Add Page Input** dialog, click **OK**.



Note: Data is passed from an Action to a JSP page through the `pageInput` **implicit object**. A implicit object is a location within a Page Flow where you can read and (oftentimes) write data for the purpose of passing the data around within the Page Flow.

The `pageInput` implicit object is the standard location for passing data from an Action to a JSP page.

An Action writes data to the `pageInput` implicit object by declaring an **action output**. The following action is declaring that it writes `Customer[]` data to the `pageInput.getCustomerResult` implicit object.

```
@Jpf.Action(
            ...
            actionOutputs = { @Jpf.ActionOutput(  name="getCustomersResult", type = model.Customer[].class) }
            ...
            )
public Forward getCustomers() {
```

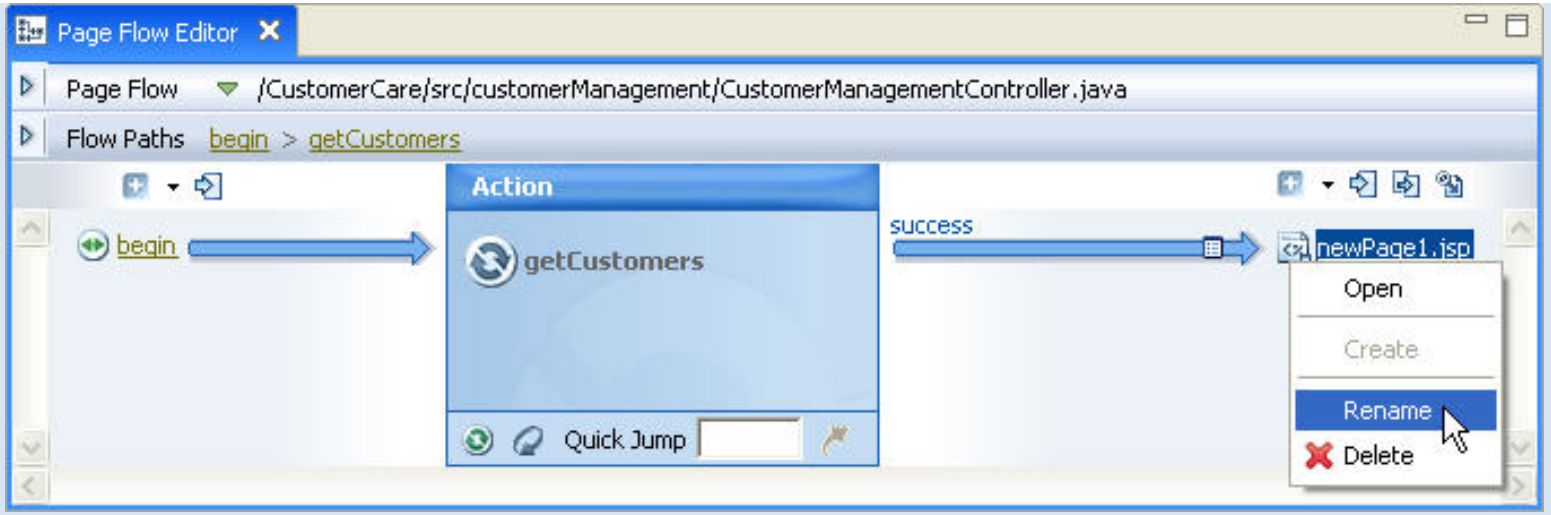
A **page input** declares the data type that a JSP page expects to *receives.* The following JSP page tag is declaring that it expects `Customer[]` data from the `pageInput.getCustomerResult` implicit object.

```
<netui-data:declarePageInput name="getCustomersResult" type="model.Customer[]" required="true" />
```

Note that if the Action passes something other than the expected data type, then a runtime exception will be thrown.

If you ever need to edit the properties of an action output/page input, right-click the arrow that passes between the Action and the JSP page and select **Edit Action Output**.

4. Right-click on the new JSP and select **Rename**.

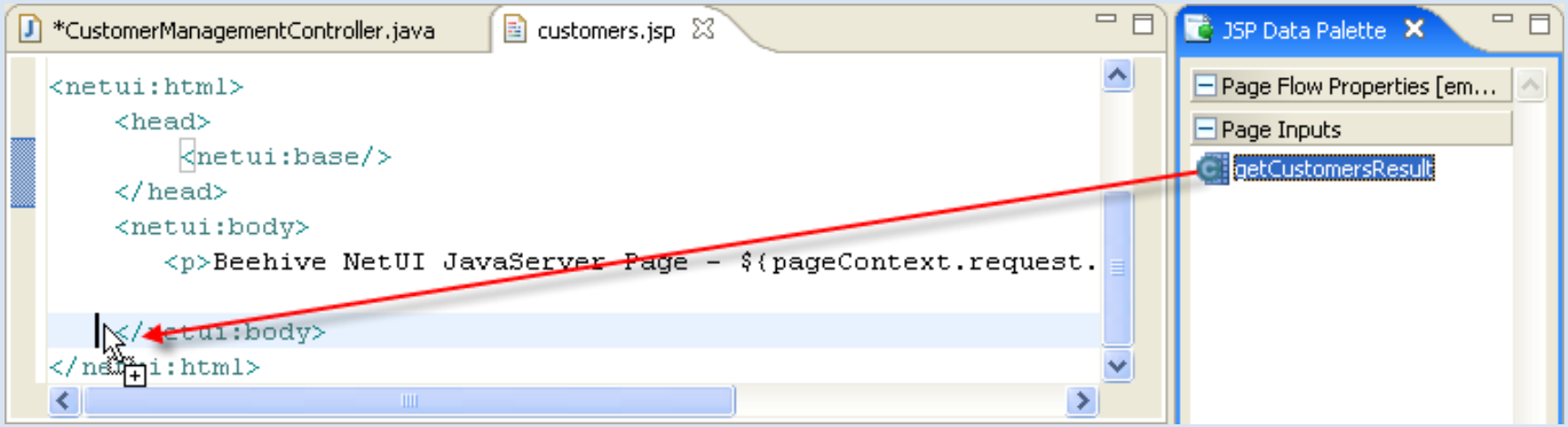


5. Rename the JSP to `customers.jsp` and press the **Enter** key.

To Create a Grid to Display the Customer List

In this task, you will add a set of JSP tags (<netui-data:dataGrid>, <netui-data:rows>, etc.) that are specially designed to render Java objects as an HTML table.

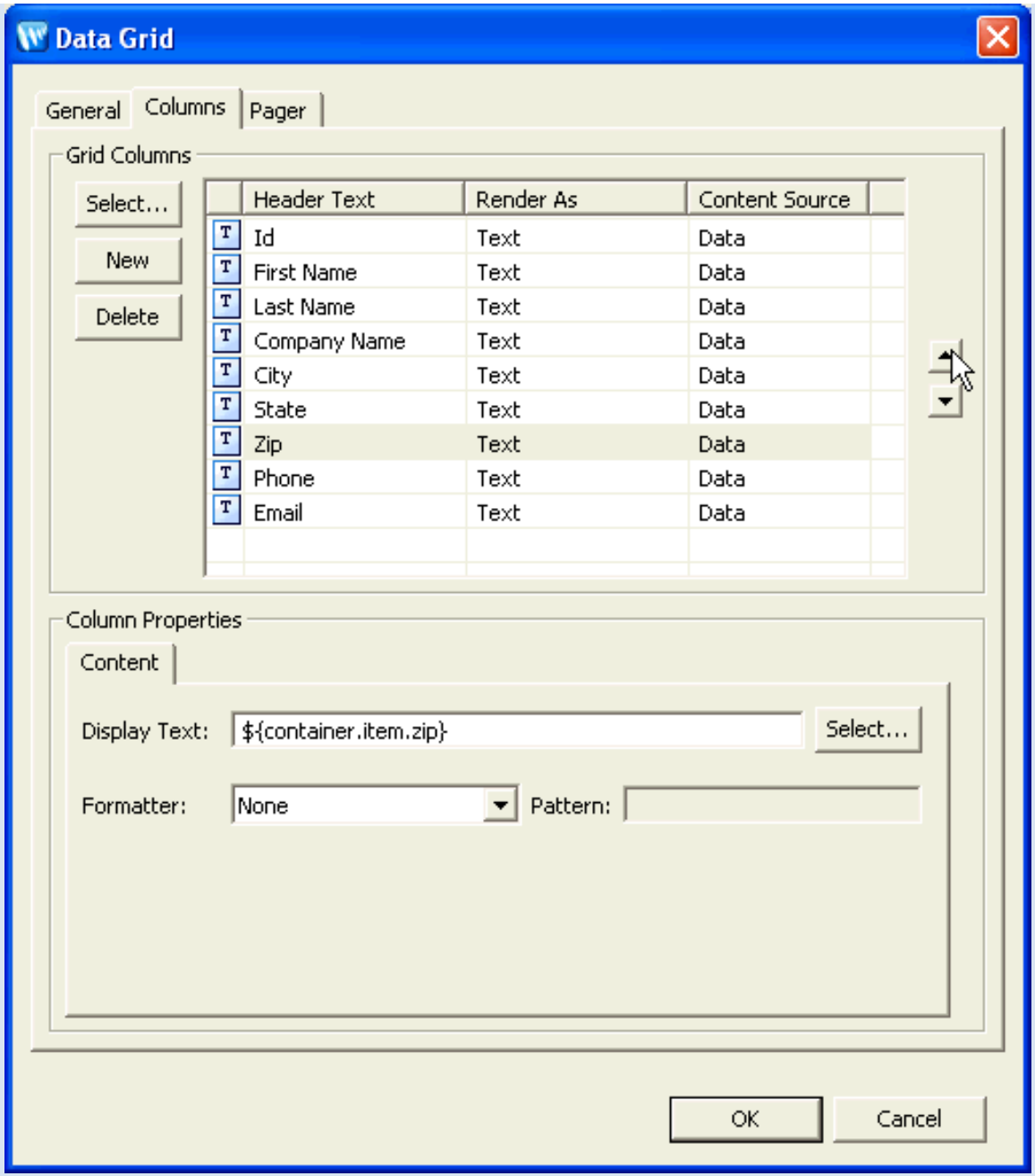
1. On the **Page Flow Editor** tab, right-click **customers.jsp** and select **Open** to open its source code.
2. On the **JSP Data Palette**, in the **Page Inputs** section, locate the **getCustomersResult** icon.
   Drag the **getCustomersResult** icon onto the source code for **customers.jsp**, dropping it directly before the **</netui:body>** tag.



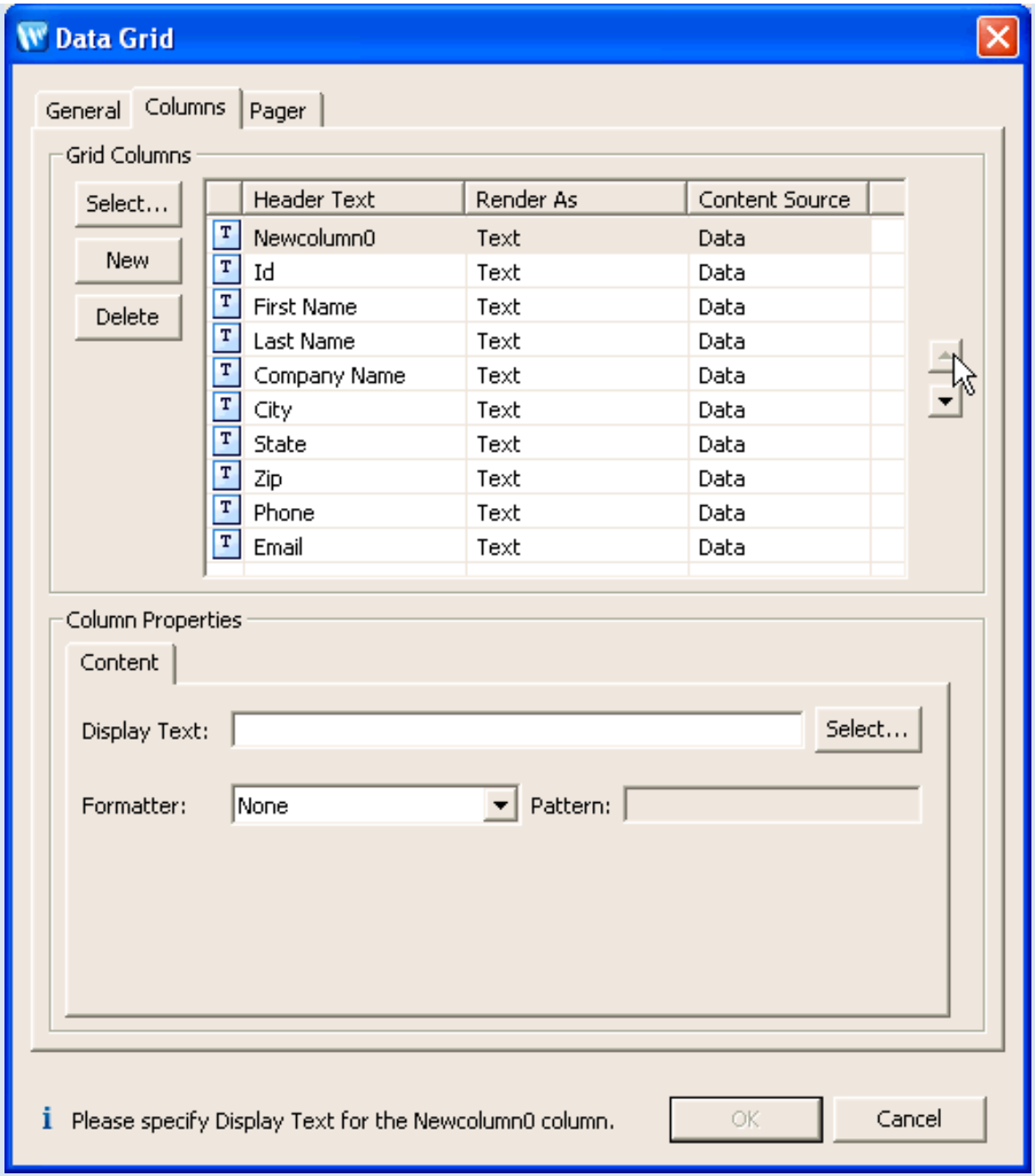
3. From the **Choose a wizard** dialog, select **Data Grid** and press **OK**.

4. On the **Data Grid** dialog, click the **Columns** tab and reorder the columns listed to match the following sequence:

   Id
   First Name
   Last Name
   Company Name
   City
   State
   Zip
   Phone
   Email

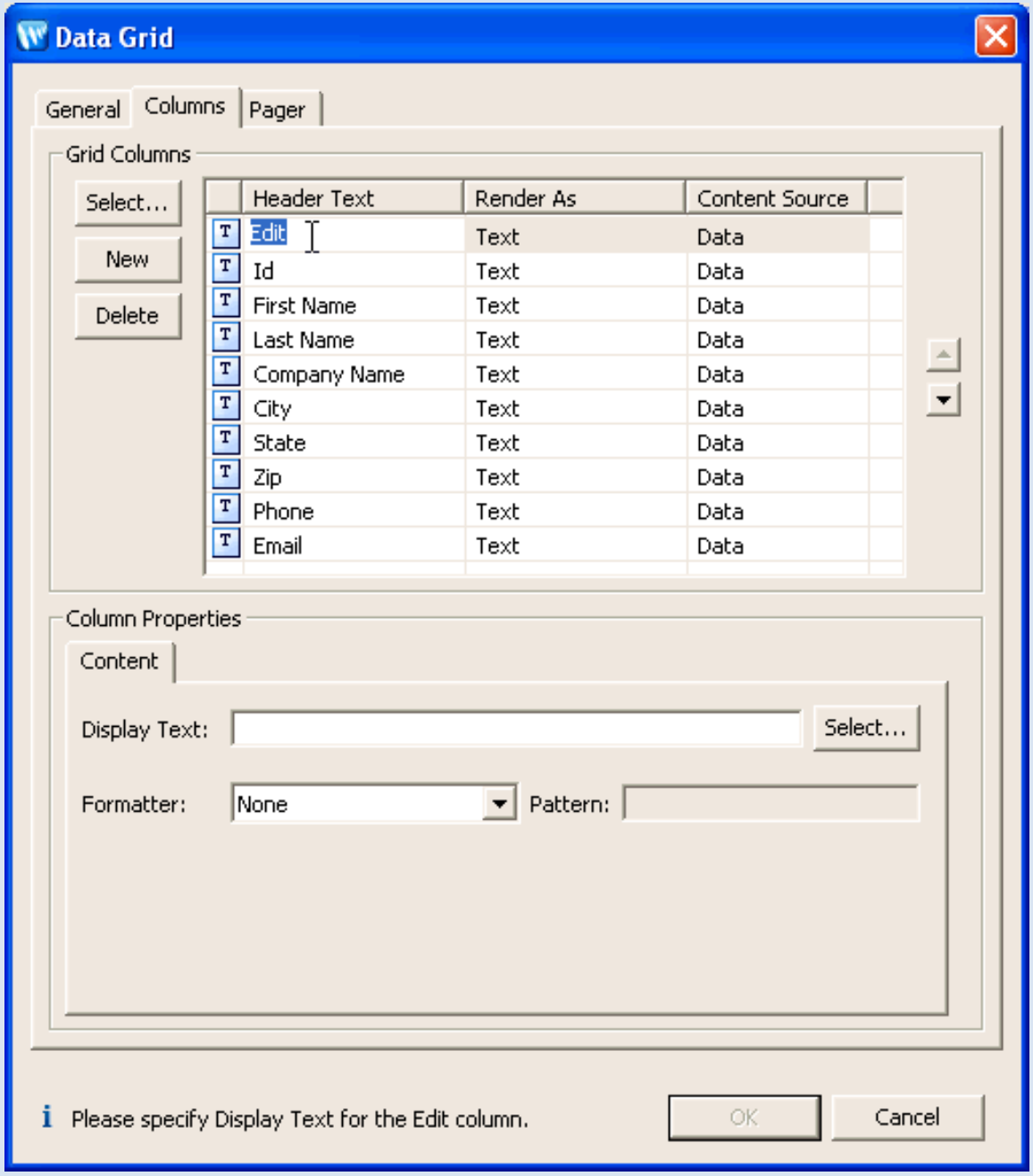


5. Click the **New** button and position the new column (named Newcolumn0 by default) at the top of the list.

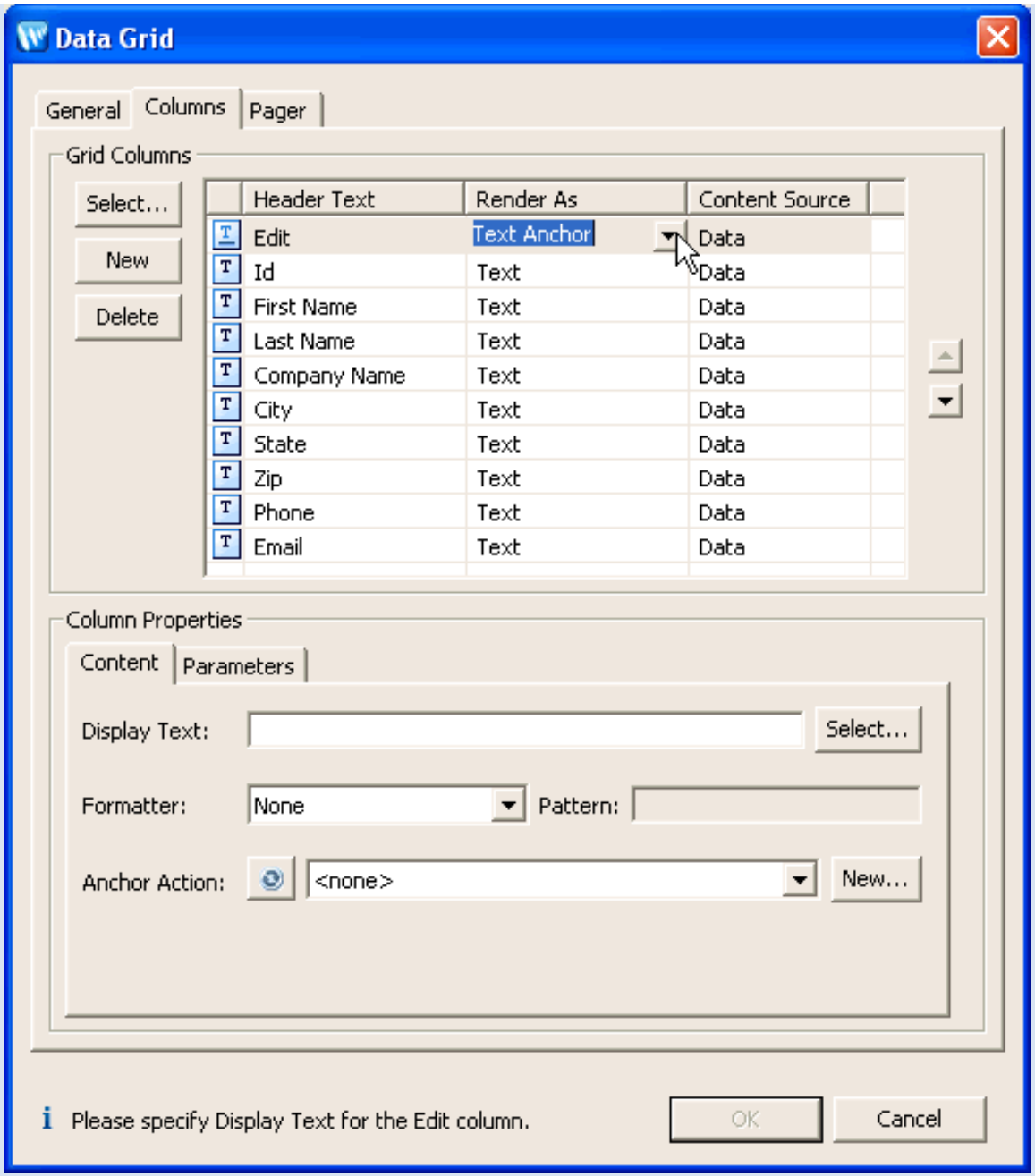


The next few tasks define an "Edit" link for each row of the table. These links take you to a editing page, where you can update the fields for a given row.

6. Change the **Header Text** of the new column from `Newcolumn0` to `Edit`. (You can change the text by clicking inside the cell you wish to edit.)



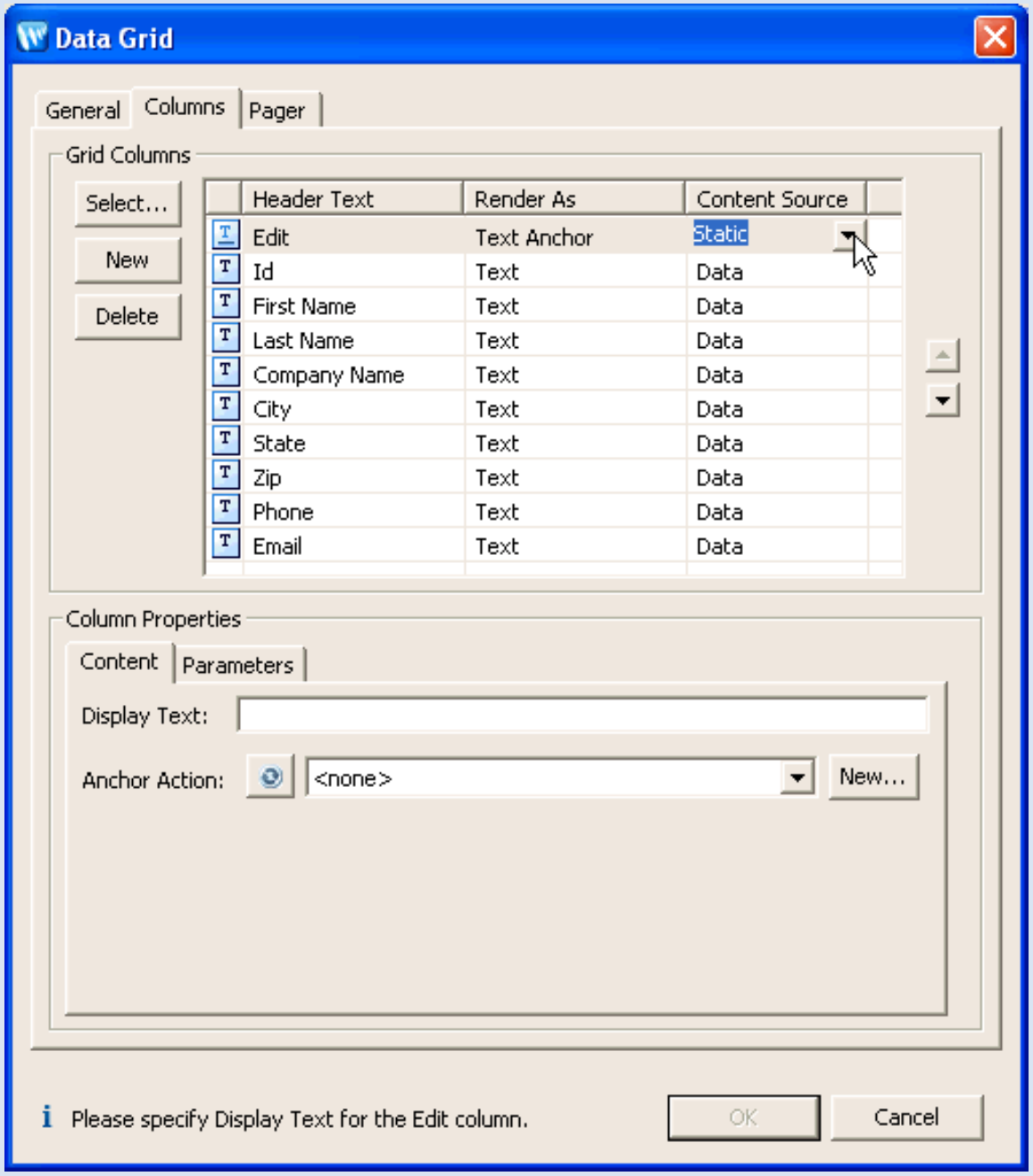
7. Set the **Render As** column to `Text Anchor`. (This makes the text into *linking* text instead of plain text.)



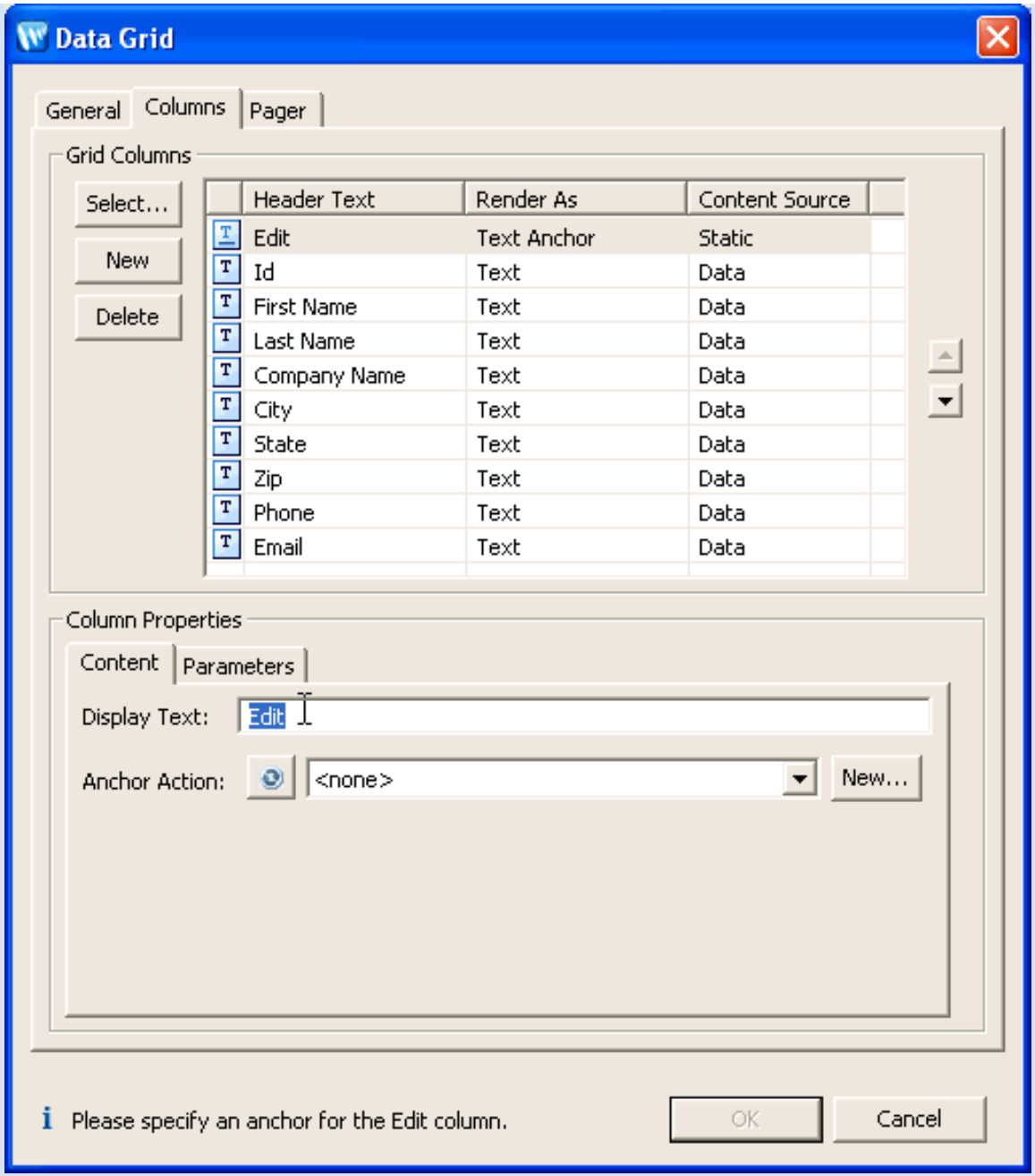
8. Set the **Content Source** column to `Static`.

   By setting this dropdown to `Static` you are signaling your intent to display the same content in the column for each row, for example, a static image. When you set it to `Data` you are signaling your intent to display dynamic content in the column, typically some display text based on the data in the row, for example, the ID of the row.

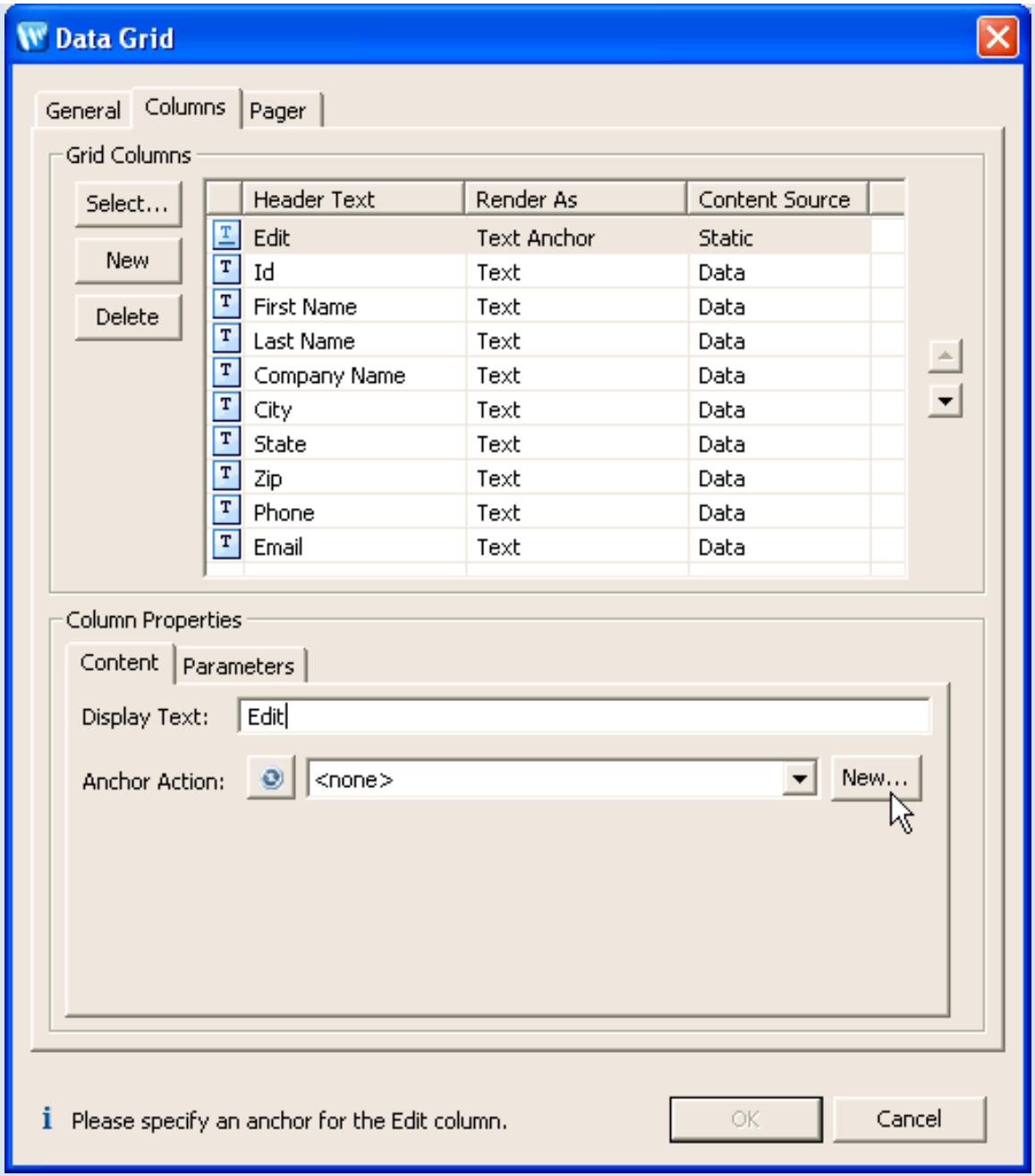
Notice that when you set the dropdown to `Data`, fields appear in the lower part of the wizard to help you format the dynamic display text. If you set the dropdown to `Static`, those fields disappear.



9. In the **Display Text** field, enter `Edit`.



10. Click the **New** button (next to the **Anchor Action** field).

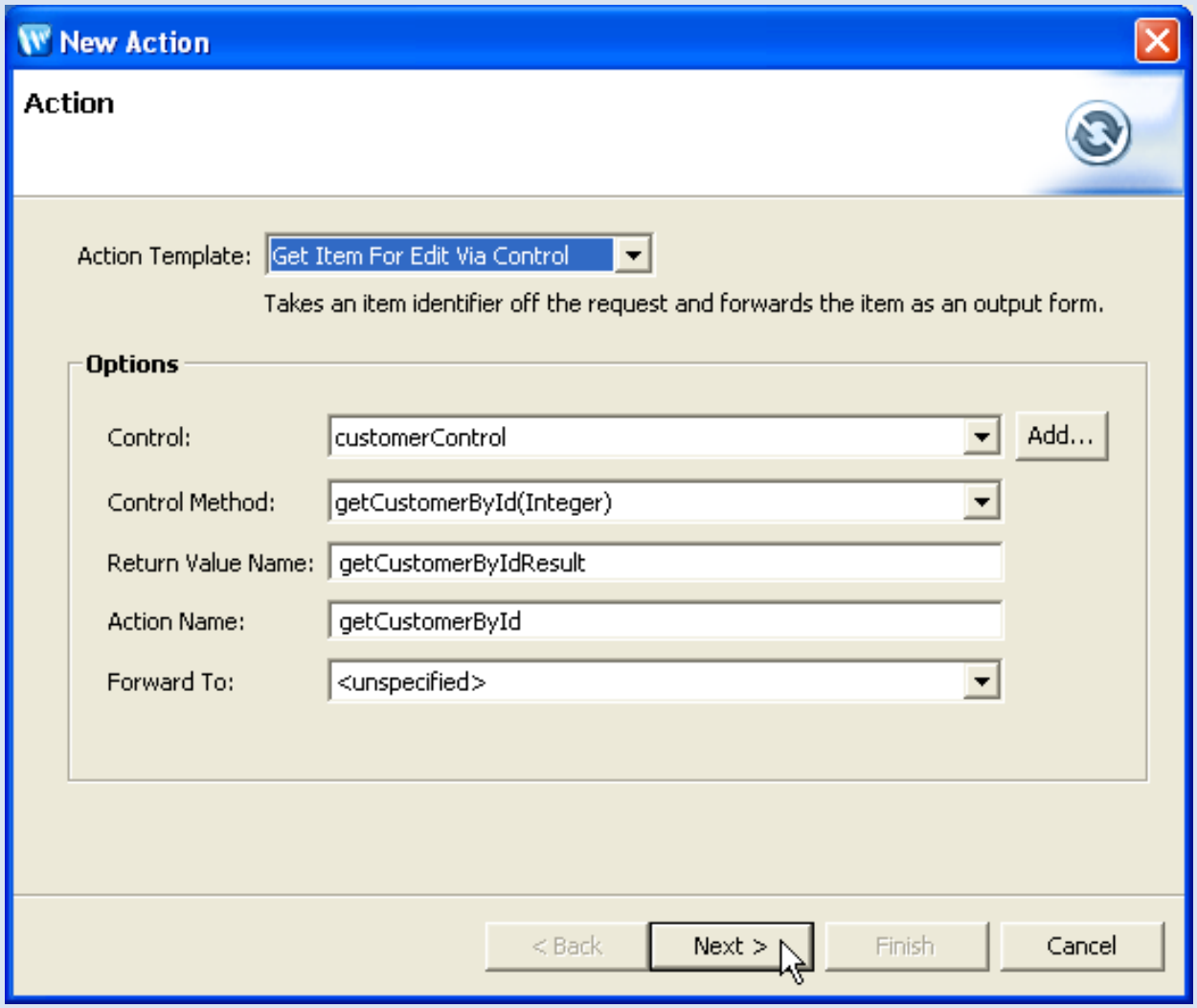


11. On the **New Action** dialog, from the **Action Template** dropdown list, choose **Get Item for Edit Via Control**.

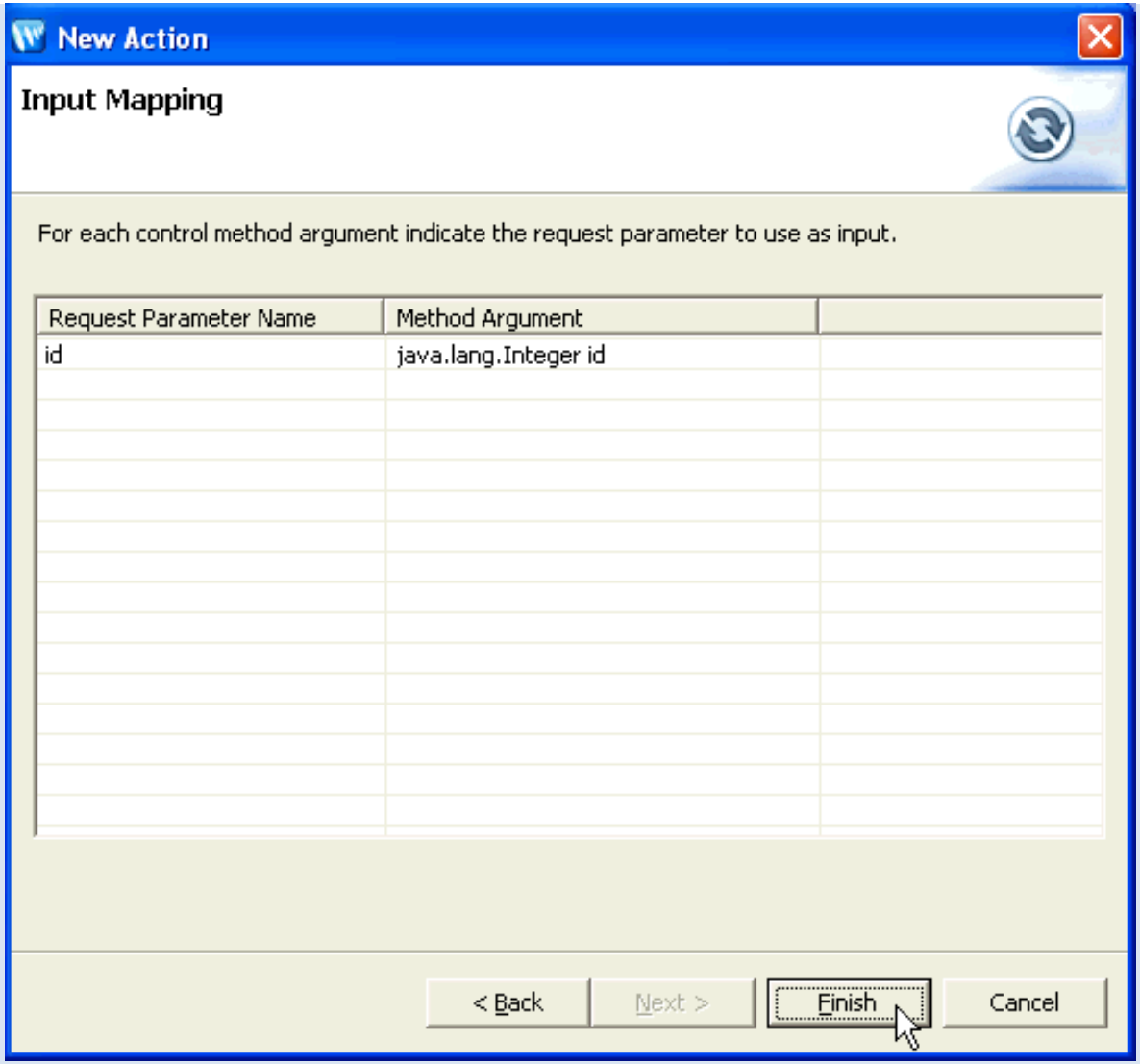
    This New Action wizard helps construct different actions for typical scenarios. Note the different options available for creating new actions. Choosing 'Simple' helps you set up a simple navigational action. Choosing 'Control Method Call' helps you set up a control-calling action.

    From the **Control Method** dropdown list, confirm that the method **getCustomerById(Integer)** is selected.

Click **Next**.

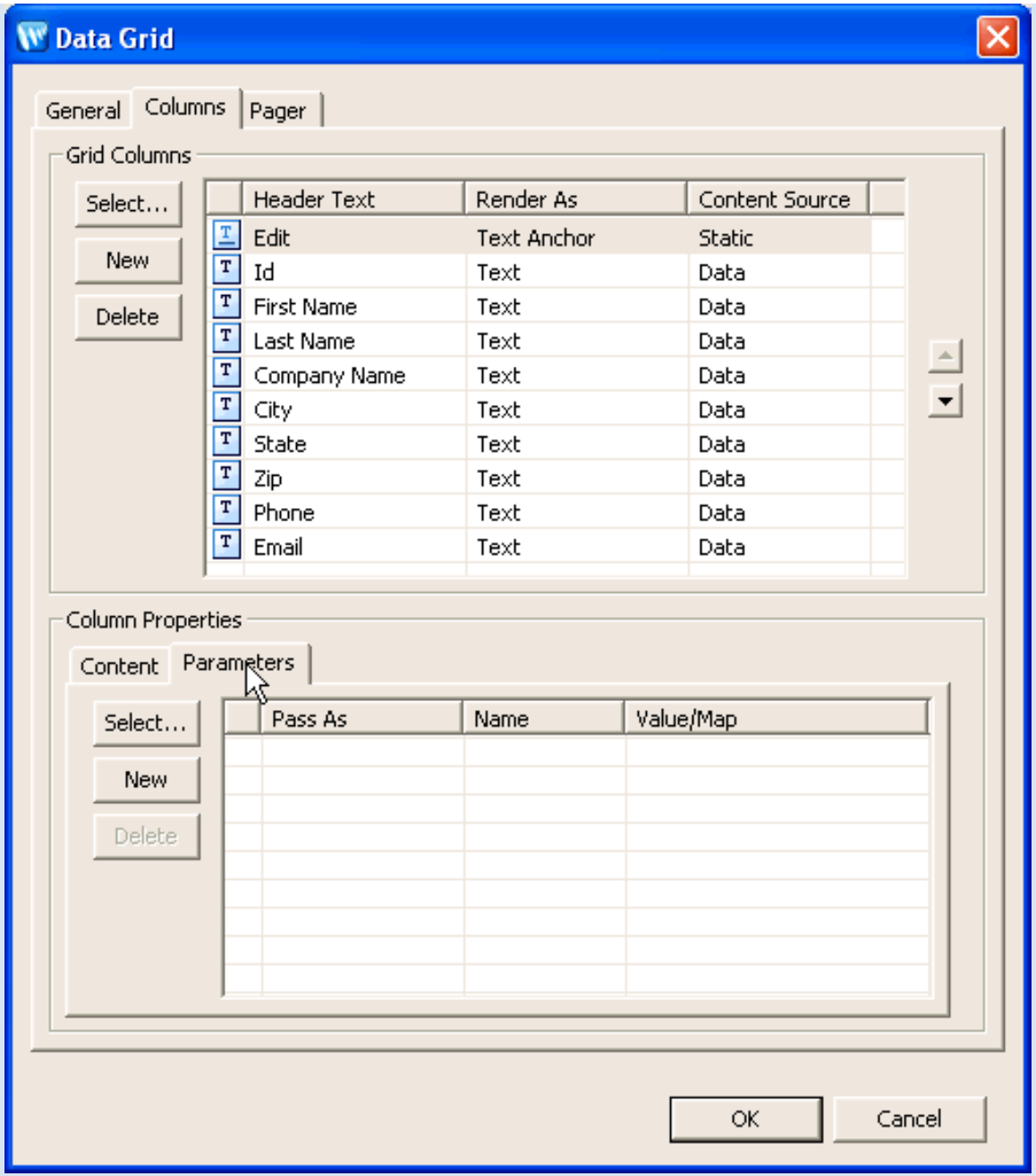


12. On the **New Action** dialog, on the **Input Mapping** page, click the **Finish** button.

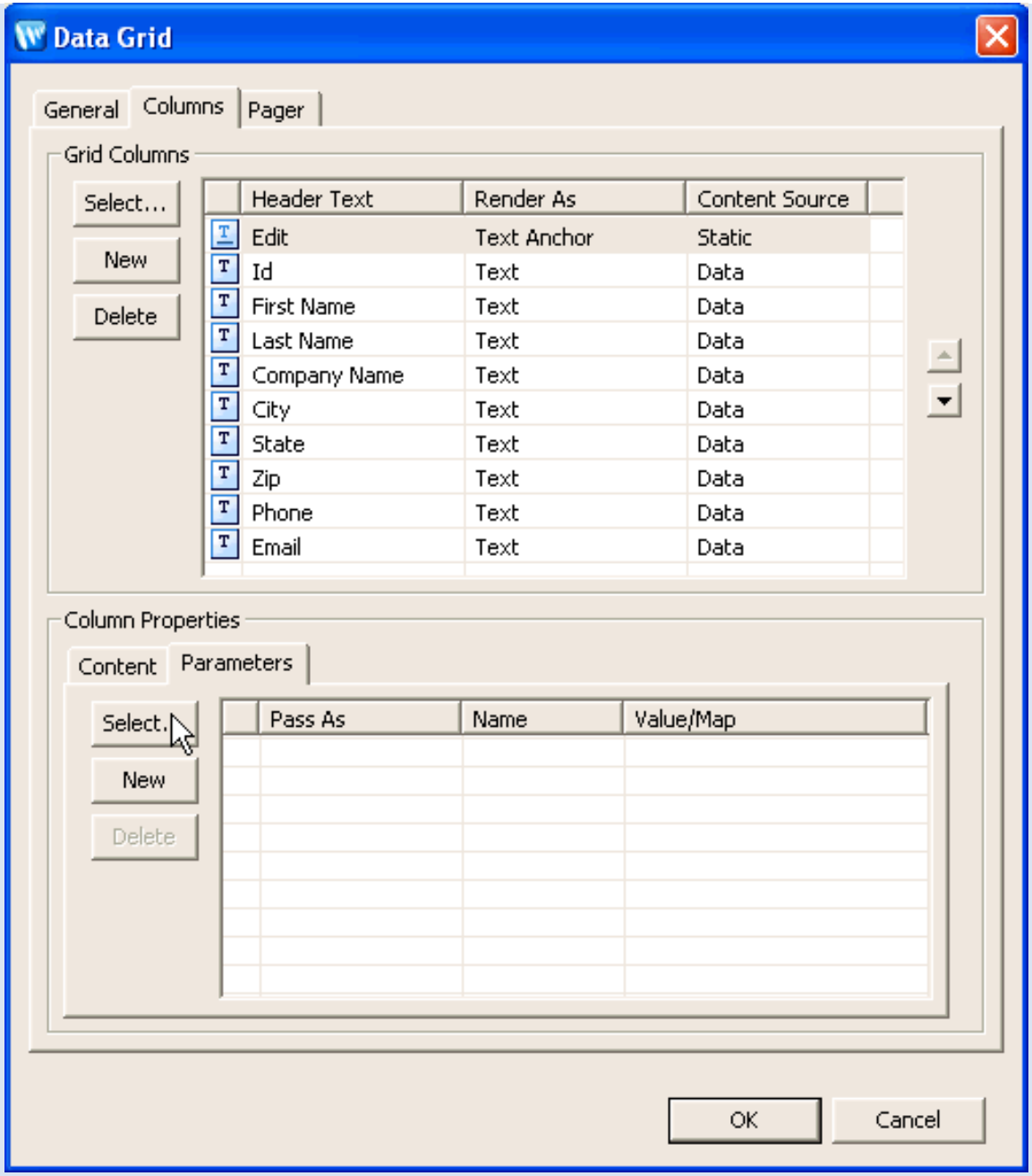


A new action called `getCustomerById` is created in the Page Flow controller file.

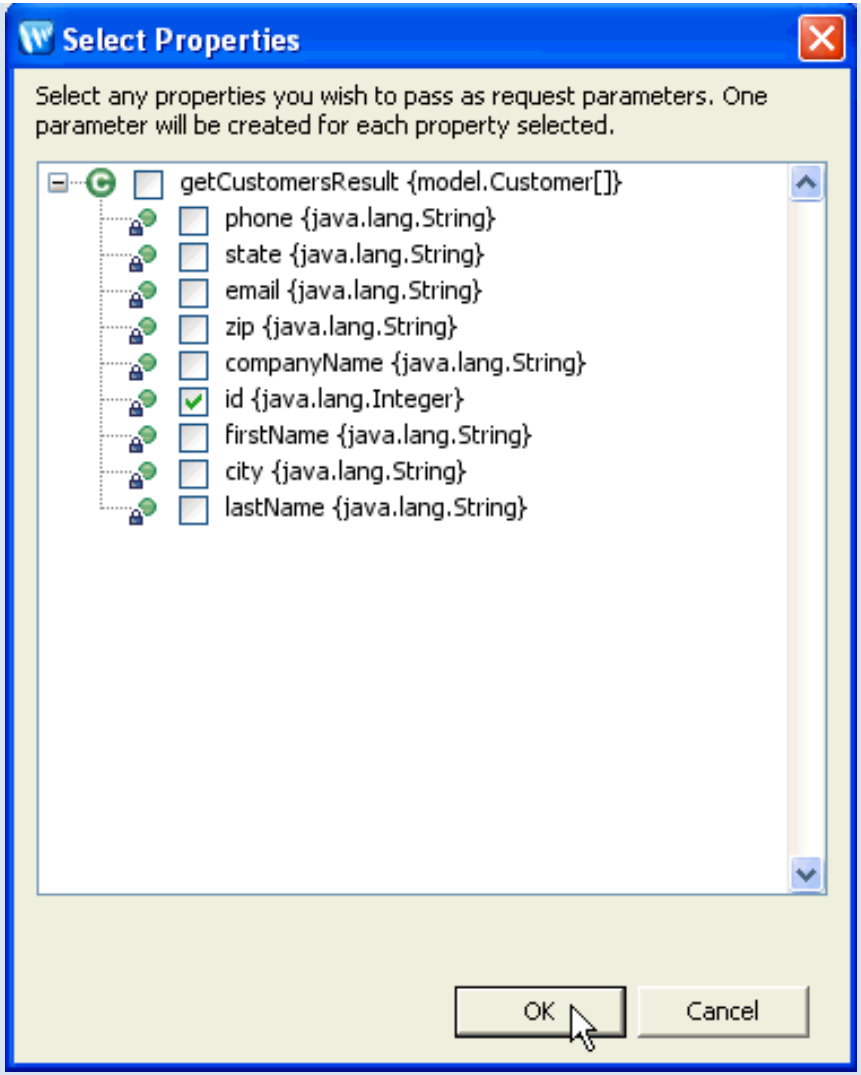
13. On the **Data Grid** dialog, click the **Parameters** tab.



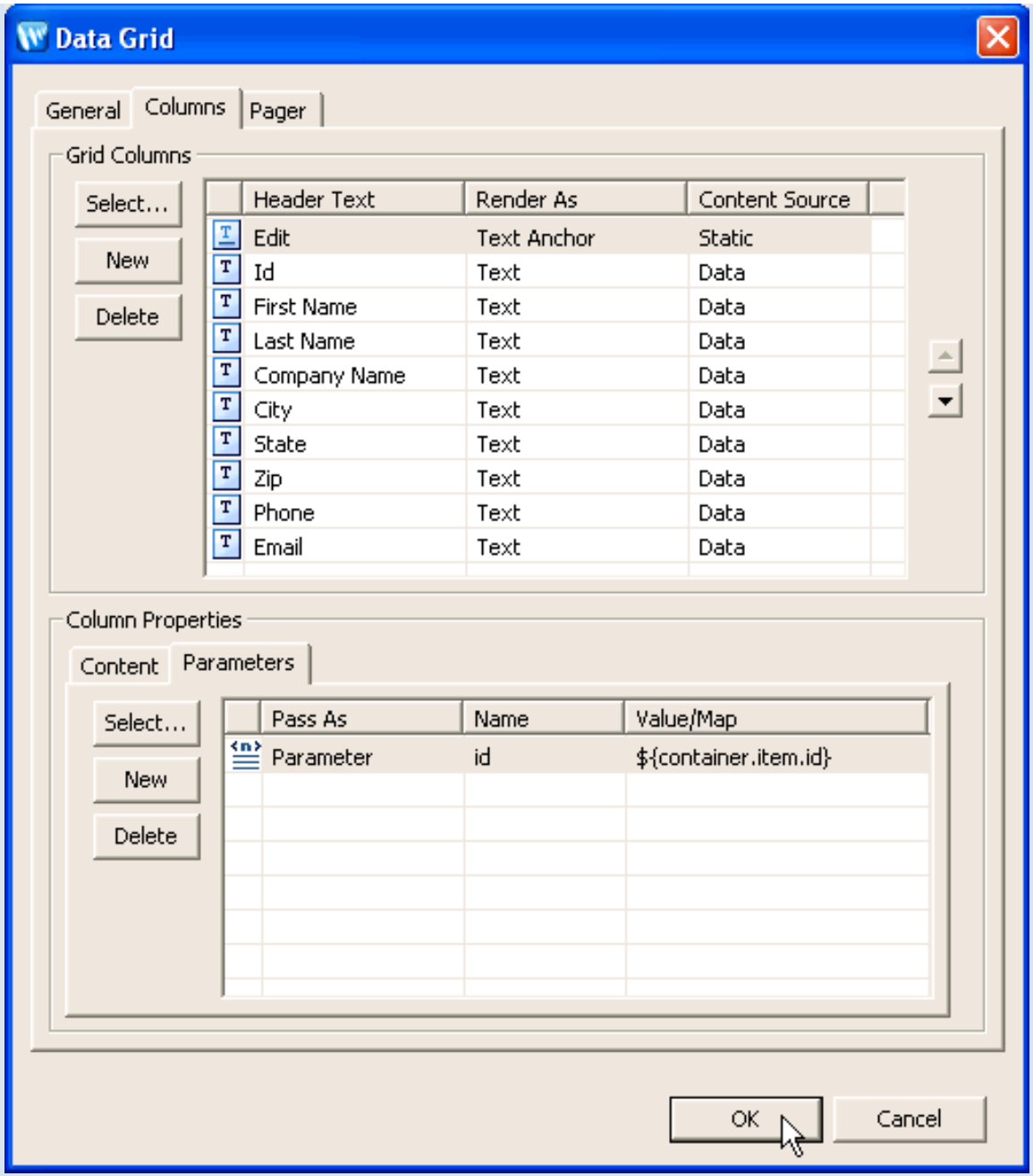
14. Click the **Select** button (on the **Parameters** tab, *not* the **Columns** tab).



15. Select the **id** property and click **OK**.



16. On the **Data Grid** dialog, click **OK**.



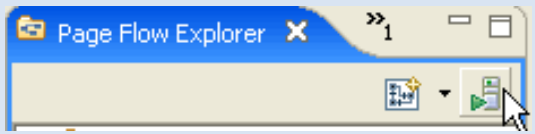
17. Press **Ctrl+Shift+S** to save your work.

You have just added the following data grid to the customer.jsp page.

```
<netui-data:dataGrid name="getCustomersResultGrid"
        dataSource="pageInput.getCustomersResult">
        <netui-data:configurePager disableDefaultPager="true" />
        <netui-data:header>
                <netui-data:headerCell headerText="Edit" />
                <netui-data:headerCell headerText="Id" />
                <netui-data:headerCell headerText="First Name" />
                <netui-data:headerCell headerText="Last Name" />
                <netui-data:headerCell headerText="Company Name" />
                <netui-data:headerCell headerText="City" />
                <netui-data:headerCell headerText="State" />
                <netui-data:headerCell headerText="Zip" />
                <netui-data:headerCell headerText="Phone" />
                <netui-data:headerCell headerText="Email" />
        </netui-data:header>
        <netui-data:rows>
                <netui-data:anchorCell value="Edit" action="getCustomerById">
                        <netui:parameter name="id" value="${container.item.id}" />
                </netui-data:anchorCell>
                <netui-data:spanCell value="${container.item.id}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.firstName}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.lastName}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.companyName}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.city}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.state}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.zip}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.phone}">
                </netui-data:spanCell>
                <netui-data:spanCell value="${container.item.email}">
                </netui-data:spanCell>
        </netui-data:rows>
</netui-data:dataGrid>
```

To Run the Page Flow

1. On the **Page Flow Explorer** tab, click the server icon to deploy and run the Page Flow.



2. In the **Run on Server** dialog, confirm that **BEA WebLogic v9.2 Server** is selected, and click **Finish**.

   Wait a minute for the server to start and the EAR to deploy.
   You will see a browser tab appear, displaying a grid of customer data.

3. Close the browser tab for **http://localhost:7001/customerCare/customerManagement/CustomerManagementController.jpf**.

Related Topics

None.

Click one of the following arrows to navigate through the tutorial:


Step 4: Create a Page to Edit Customer Data

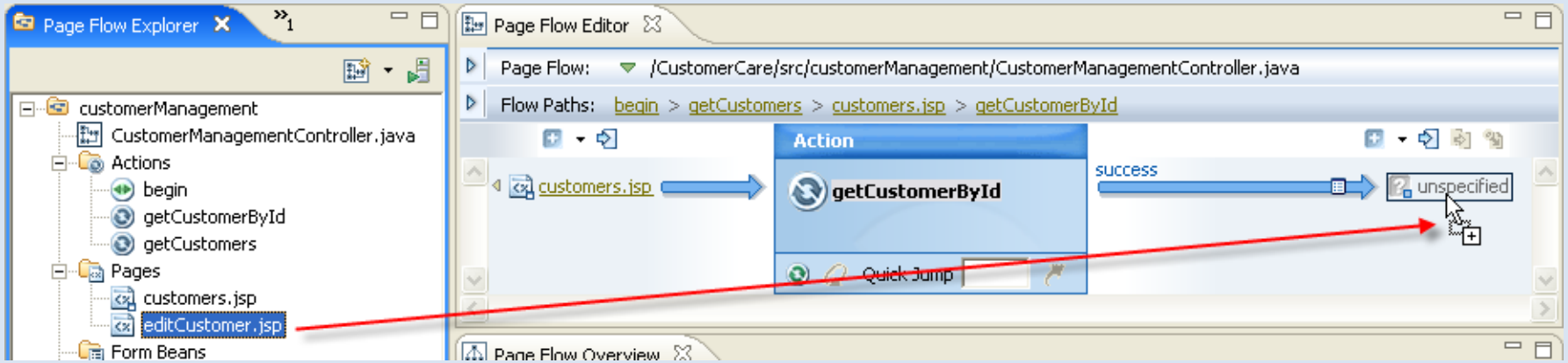
In this step you will add a JSP page for editing individual customer records.

The tasks in this step are:

To Create a Record Editing Page

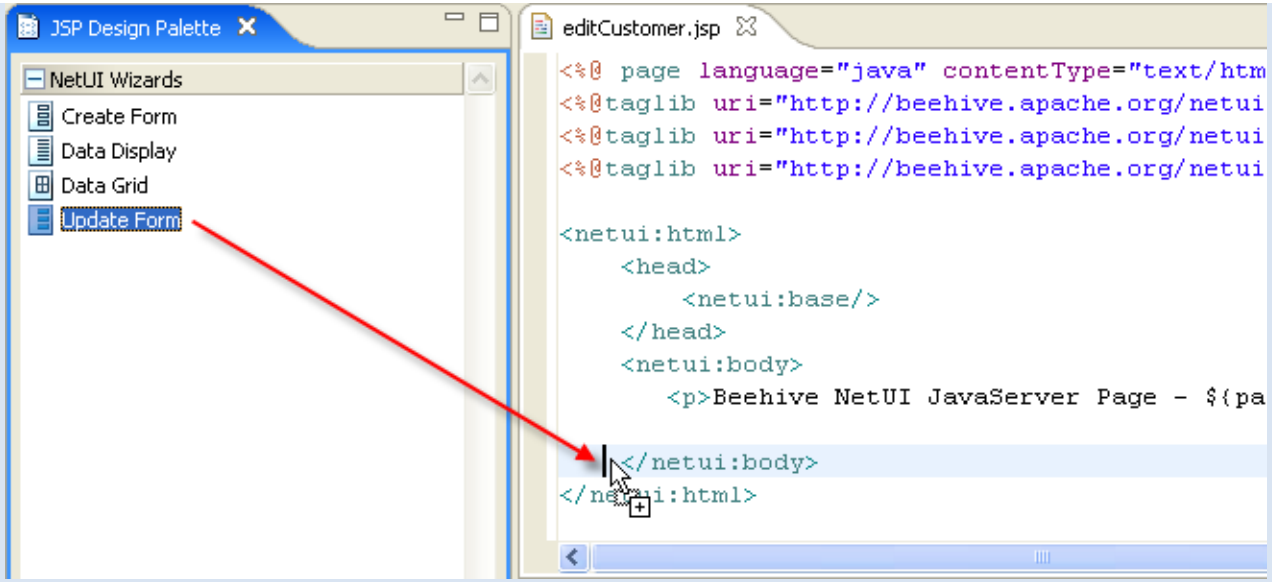
1. On the **Page Flow Explorer** tab, right-click on the **Pages** node and select **New JSP Page**.
2. Rename the page to `editCustomer.jsp`. Press **Enter**.
3. On the **Page Flow Editor** tab, place the cursor in the **Quick Jump** field, enter getCustomerById, and press the **Enter** key. This will display the getCustomerById node in the center pane. (Alternatively, you can also place the cursor in the Quick Jump field and press Ctrl+Space Bar to view a list of available nodes.)
4. Drag **editCustomer.jsp** icon (located on the **Page Flow Explorer** tab), onto the **unspecified** node (located on the **Page Flow Editor** tab).

   Note: make sure to drop *directly on* the **unspecified** node as shown below.

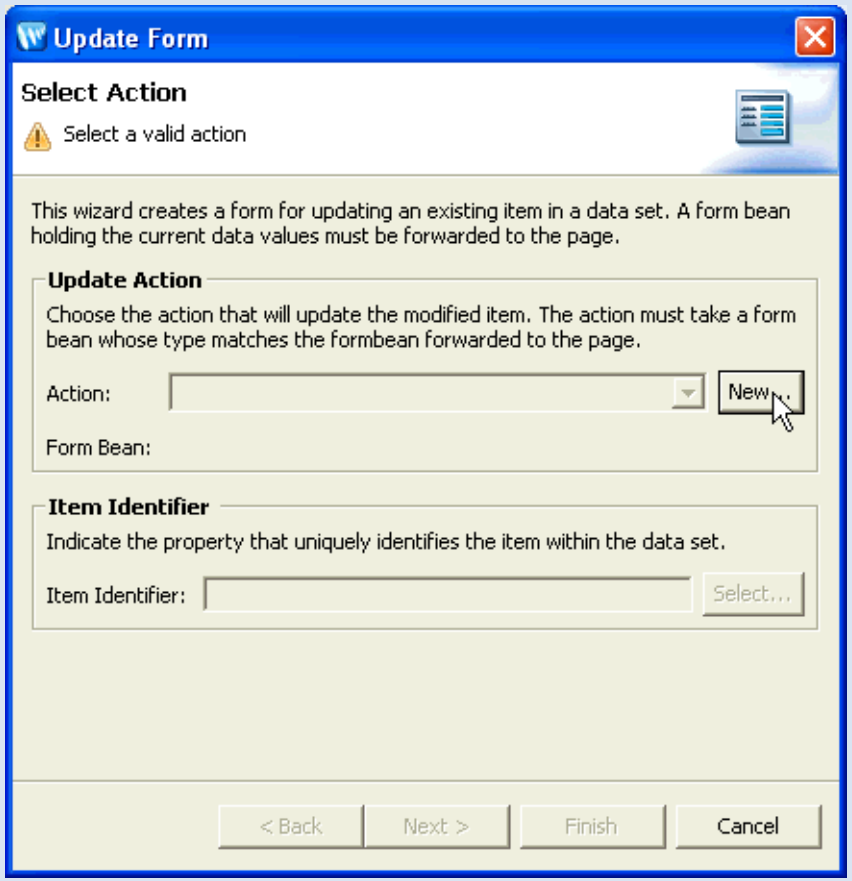


To Make a Form for Updating the Customer Data

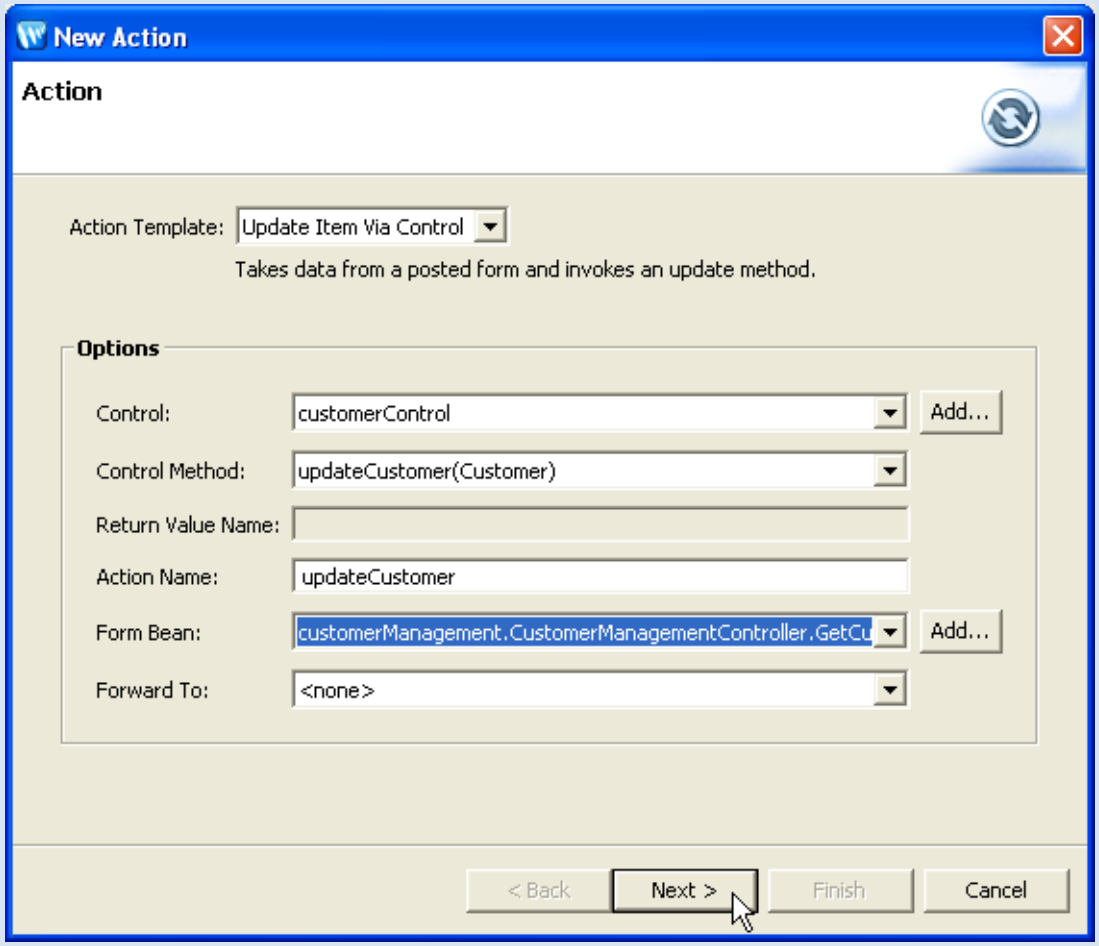
1. On the **Page Flow Explorer** tab, double-click **editCustomer.jsp** to open its source code.
2. On the **JSP Design Palette**, in the **NetUI Wizards** section, drag the **Update Form** pattern onto the JSP source editor and drop it directly before the **</body>** tag.



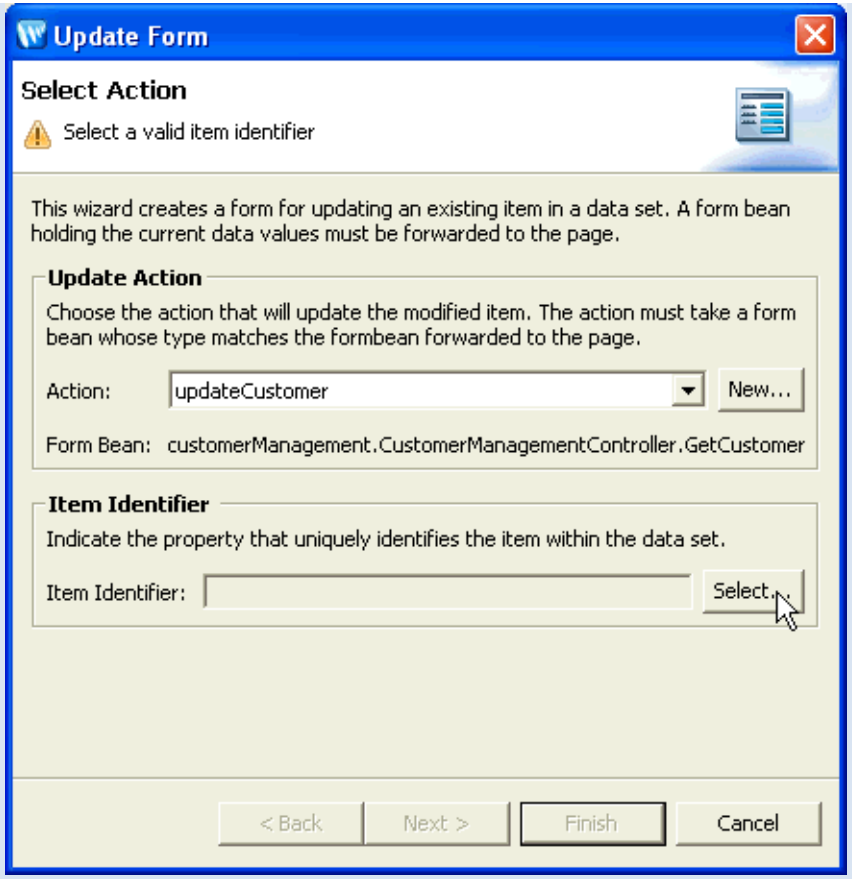
3. On the **Update Form** dialog, next to the **Action** field, click the **New** button.



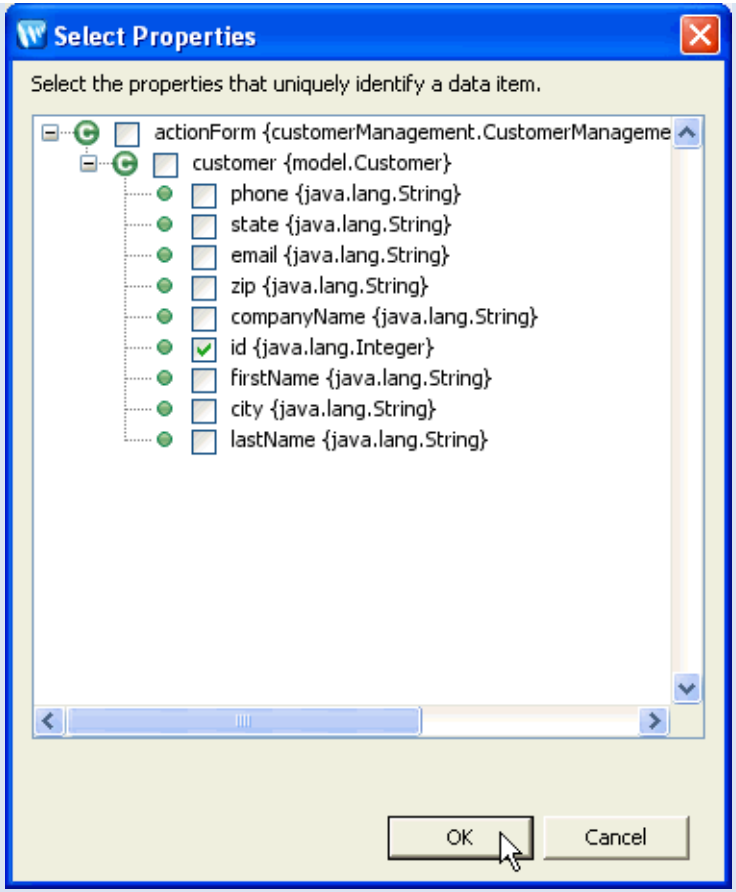
4. On the **New Action** dialog,
   from the **Control Method** dropdown list, choose the **updateCustomer(Customer)** method,
   from the **Form Bean** dropdown list, select **customerManagement.CustomerManagementController.GetCustomerByIdFormBean**.
   Click the **Next** button.



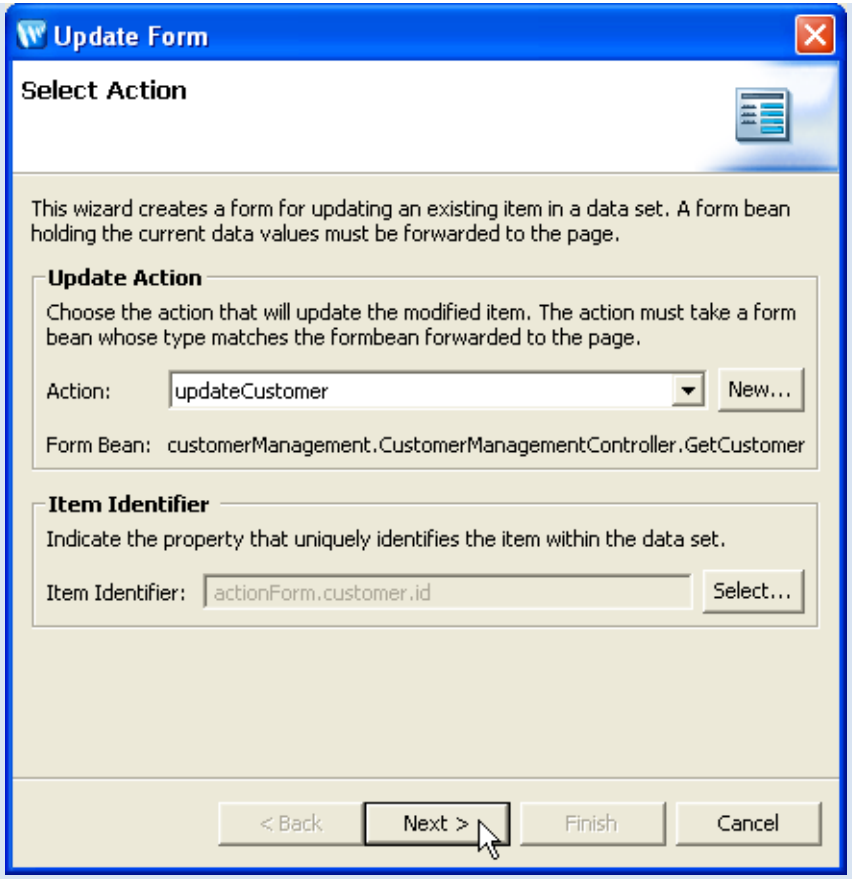
5. In the **New Action** dialog, on the **Input Mapping** page, click **Finish**.
6. Click the **Select** button next to the **Item Identifier** field.



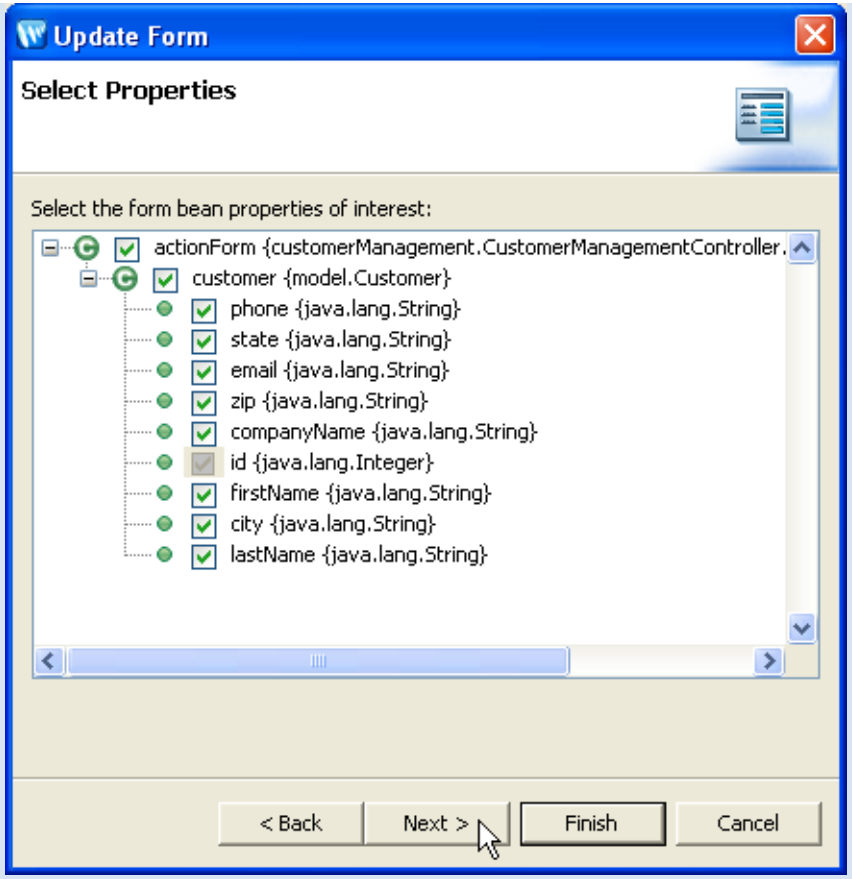
7. Select the **id** property and click **OK**.



8. On the **Update Form** dialog, on the **Select Action** page, click the **Next** button.



9. On the **Update Form** dialog, on the **Select Properties** page, click the **Next** button.



10. Arrange the fields so that they have the following order:

    Id
    First Name
    Last Name
    Company Name
    City
    State
    Zip
    Phone
    Email

    Click the **Finish** button.

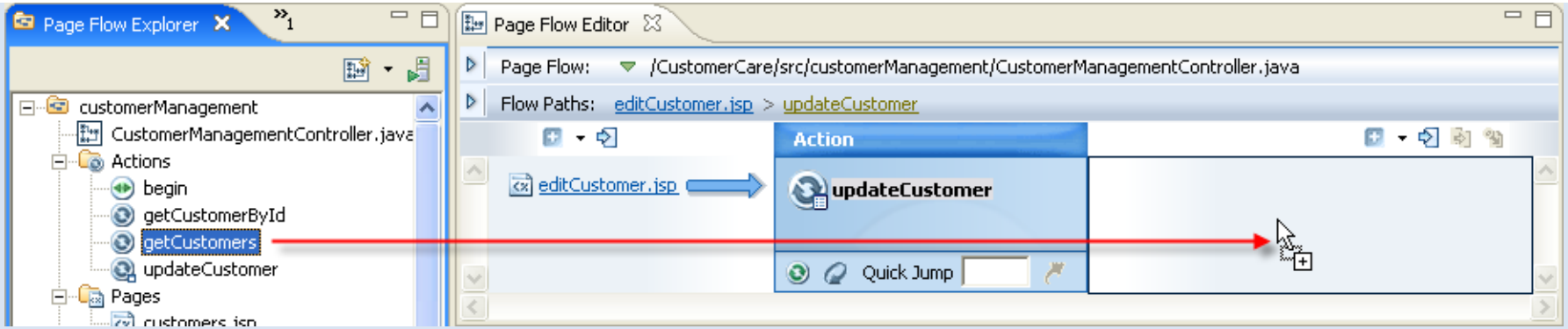
    By clicking Finish, you have added the following form to **editCustomer.jsp**.

```
<netui:form action="updateCustomer">
                <netui:hidden dataSource="actionForm.customer.id"></netui:hidden>
                <table>
                        <tr valign="top">
                                <td>Customer:</td>
                                <td>
                                <table>
                                        <tr valign="top">
                                                <td>FirstName:</td>
```

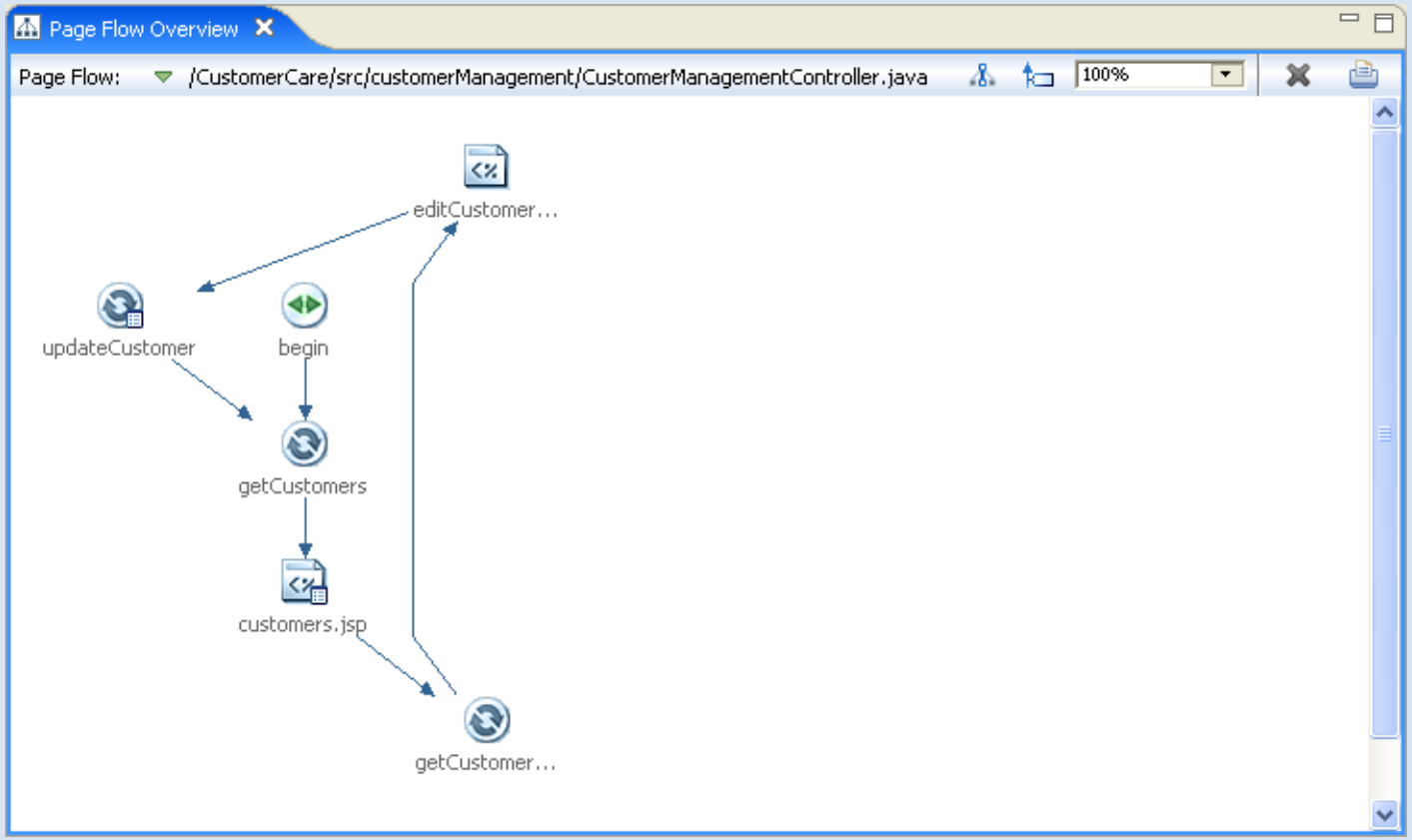
```
                                        <td><netui:textBox dataSource="actionForm.customer.firstName"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>LastName:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.lastName"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>CompanyName:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.companyName"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>City:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.city"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>State:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.state"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>Zip:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.zip"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>Phone:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.phone"></netui:textBox>
                                        </td>
                                </tr>
                                <tr valign="top">
                                        <td>Email:</td>
                                        <td><netui:textBox dataSource="actionForm.customer.email"></netui:textBox>
                                        </td>
                                </tr>
                        </table>
                        </td>
                </tr>
        </table>
        <br />
<netui:button value="updateCustomer" type="submit" />
</netui:form>
```

To Set Up Navigation Back to the Customer List

1. On the **Page Flow Editor** tab, place the cursor in the **Quick Jump** field, enter **updateCustomer**, and press the **Enter** key. This will display the updateCustomer node in the center pane. (Alternatively, you can also place the cursor in the Quick Jump field and press Ctrl+Space Bar to view a list of available nodes.)
2. Drag the **getCustomers** action (located on the **Page Flow Explorer** tab) onto the right-hand side of the **Page Flow Editor** tab .



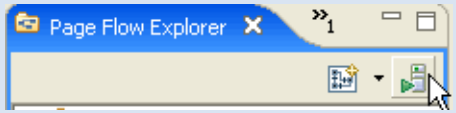
3. Press **Shift+Ctrl+S** to save your work.
4. The **Page Flow Overview** should appear as follows:



5. Close the source file for **editCustomer.jsp**.

To Run the Page Flow

1. On the **Page Flow Explorer** tab, click the server icon to deploy and run the Page Flow.



2. In the **Run on Server** dialog, confirm that **BEA WebLogic v9.2 Server** is selected, and click **Finish**.

   Wait a minute for the EAR and web application project to deploy.
   You will see a browser tab appear, displaying a grid of customer data

3. Click the **Edit** link for "David Owen".
4. Update the information for David Owen and click **updateCustomer**.
5. Note that the information is updated on the grid page.

Related Topics

none.

Click one of the following arrows to navigate through the tutorial:

Tutorial: Accessing a Database from a Web Application

In this tutorial you learned:

- how to provide user access to a database through a web application
- how Page Flows work
- how a database control queries a database
- how databinding is used to pass data around a Page Flow
- how a data grid renders complex data as an HTML table

Click the arrow below to navigate through the tutorial:

Tutorial: Java Server Faces Integration

What This Tutorial Teaches

This tutorial teaches you how to enable and use Java Server Faces in a Workshop for WebLogic web application.

The application you build here is a hybrid application that uses both JSF and Beehive NetUI technology. JSF supplies the user interface portion of the application, while Beehive NetUI supplies centralized backend data processing.

> **Note:** This tutorial requests that you create a new workspace; if you already have a workspace open, this will restart the IDE. Before beginning, you might want to launch help in standalone mode to avoid an interruption the restart could cause, then locate this topic in the new browser. See Using Help in a Standalone Mode for more information.

The tutorial contains step-by-step instructions for building a simple web application querying and viewing customer data. As you progress through the tutorial you will learn:

- how Workshop for WebLogic uses JSF and Beehive NetUI technologies to simplify web application development
- how to enable JSF in a Workshop for WebLogic web application
- how to use JSF tags to create user data submission forms
- how to use JSF tags to display complex Java objects as simple HTML tables
- how to call a Beehive NetUI action from a JSF page

Note: this JSF tutorial assumes that you have a basic knowledge of Beehive NetUI web application technology, including the roles of controller classes, JSP pages, form beans and action methods. If you are unfamiliar with these concepts you may want to complete Tutorial: Accessing a Database from a Web Application before continuing.

Tutorial Synopsis

Step 1: Create a JSF-Enabled Web Project

In the first step of this tutorial you will create the foundation of your application by creating two projects: an EAR project and a Web Application Project.

The EAR project has two main purposes: (1) it is a composite application that acts as a container for other applications and (2) it contains resources, in the form of library modules and JARs, for the applications contained in it.

For the purposes of this tutorial, the most important JARs contained in the EAR project are (1) the Beehive NetUI JARs and (2) the JSF JARs.

The Web Application Project accesses these JAR resources in the EAR simply by referencing them, not by copying the directly. This allows multiple web projects to point to the same resources in an EAR, without unnecessary duplication of resources.

Step 2: Create a JSF Web Application

In step you will create a simple web application that uses JSF tags to define the user interface.

The web app contains one page where users can submit queries and another page for viewing the results.

Related Topics

None.

Click the arrow below to navigate through the tutorial:

Step 1: Create a JSF Enabled Web Project

In this step you will set up a JSF-enabled web project. The

The tasks in this step are:

- To Create a New Workspace
- To Create a New EAR Project and a New Web Project
- To Import Files into the Web Project
- To Add a WebLogic Server

To Create a New Workspace

If you haven't started Workshop for WebLogic yet, use these steps to do so.

... on Microsoft Windows

If you are using a Windows operating system, follow these instructions.

- From the **Start** menu, click **All Products > BEA Products > Workshop for WebLogic Platform 9.2**

...on Linux

If you are using a Linux operating system, follow these instructions.

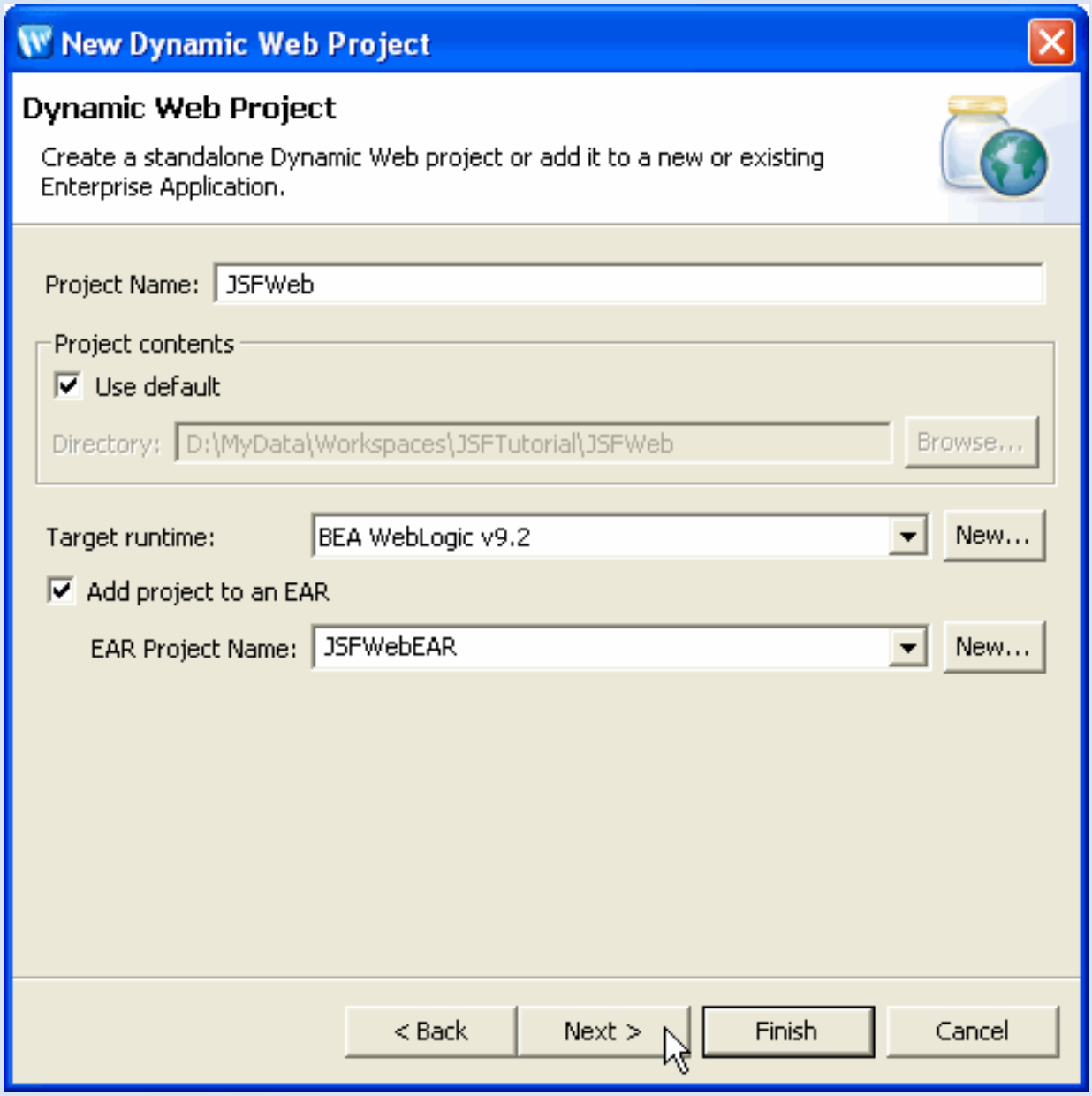
- Run BEA_HOME/workshop92/workshop4WP/workshop4WP.sh

1. In the **Workspace Launcher** dialog, click the **Browse** button.
2. In the **Select Workspace Directory** dialog, navigate to a directory of your choice and click **Make New Folder**.
3. Name the new folder `JSFTutorial`, press the **Enter** key and click **OK**.
4. In the **Workspace Launcher** dialog, click **OK**.

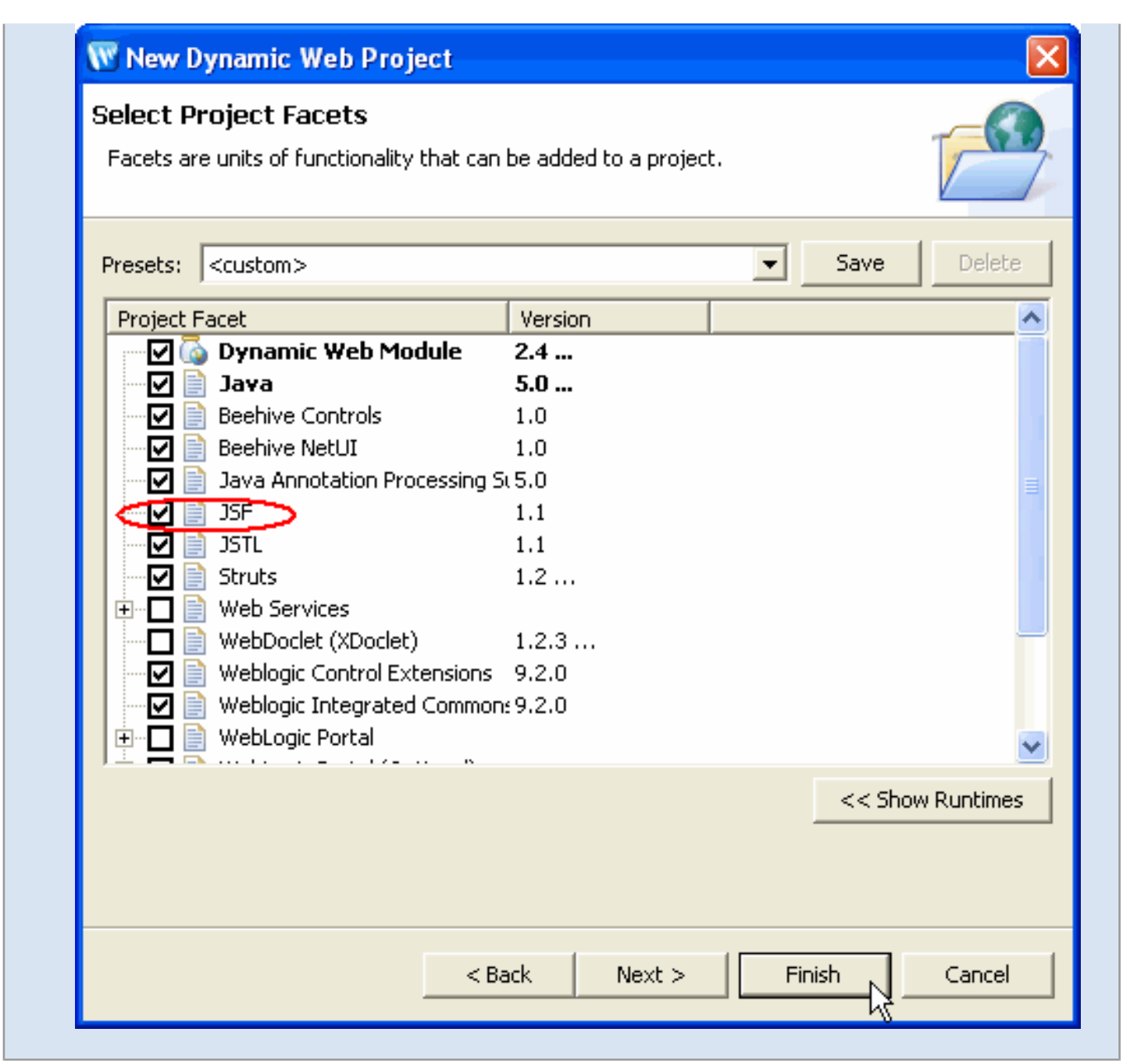
To Create a New Web Project and a New EAR Project

1. From the **File** menu, select **New** > **Project**.
2. In the **New Project** dialog select the node **Web> Dynamic Web Project**. Click **Next**.

3. In the **Project Name** field, enter `JSFWeb`.
   Place a checkmark next to **Add project to an EAR**.
   Confirm that the field **EAR Project Name** shows the value: `JSFWebEAR`.
   Click **Next**.



4. Place a check mark next to the facet **JSF** (circled in red in the image below).
   Click **Finish**.

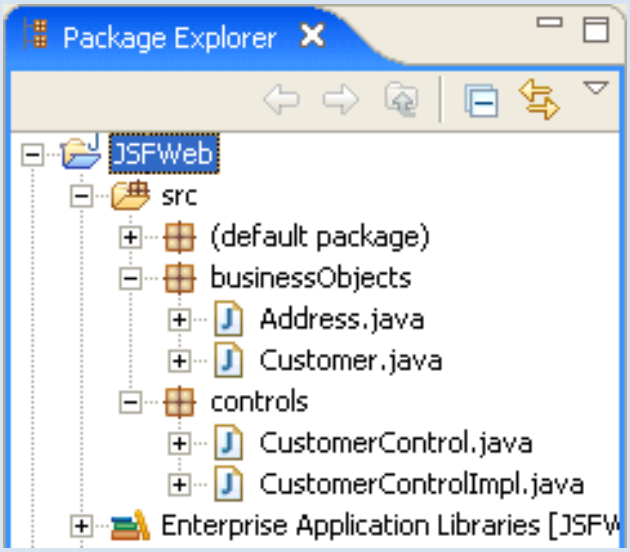


To Import Files into the Web Project

In this step you will import control files into your web project, control files that provide access to customer data.

1. On the **Package Explorer** tab, open the **JSFWeb** folder.
2. Open Windows Explorer (or your operating system's equivalent) and navigate to the directory **BEA_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples_9.2.1/tutorials/resources/jsf/**
3. Drag the folders **businessObjects** and **controls** (located at **BEA_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples_9.2.1/tutorials/resources/jsf/**) into the **Package Explorer** tab directly onto the folder **JSFWeb/src**.

4. Confirm that the following directory and file structure exists before proceeding.



To Add a WebLogic Server

In this step you will point to a server where you can deploy your application.

Note: If you have executed the JSF tutorial before, it is recommended that you either (1) remove previous JSF tutorial code from your server or (2) create a new server domain.

1. Confirm that you are in the Workshop perspective (**Window > Show Perspective > Workshop**).
2. Click the **Servers** tab.
3. Right-click anywhere within the **Servers** tab, and select **New > Server**.
4. In the **New Server** dialog, select **BEA Systems Inc.**> **BEA WebLogic v9.2 Server**. Click **Next**.
5. In the **Domain home** field, use the pulldown to set the domain to **BEA_HOME/weblogic92/samples/domains/workshop**. (Note: if you are using a newly created server domain for the JSF tutorial, then use the Browse button to navigate to that new server domain, e.g., BEA_HOME/user_projects/domains/base_domain.)
   Click **Next**.
6. In the **Available projects** column, select **JSFWebEAR**. Click the **Add** button to move the select project to the **Configured projects** column.
7. Click **Finish**.

   A new server is added to the Servers tab.

You can use the Servers tab to manage your servers and project deployments as you develop you applications.

To deploy or undeploy a project from a server, right-click the server and select **Add and Remove Projects**.

For more properties, double-click a server.

Related Topics

Integrating Java Server Faces into a Web Application

Click one of the following arrows to navigate through the tutorial:

Step 2: Create a JSF Web Application

The tasks in this step are:

To Add a Control to the Page Flow

In this step you will add a control to the web application. The control is designed to return customer data in the form of an ArrayList of Customer objects. In a more real world scenario this control might call out to a database or a web service to retrieve the customer data. But for the sake of testing the JSF components, the control in this scenario simply returns a fixed ArrayList of Customer objects.

1. **Select Window > Open Perspective > Page Flow**. (For a description of the Page Flow perspective, see Page Flow Perspective.)
2. On the **Page Flow Explorer** tab, right-click on the **Referenced Controls** node and select **Add Control**.
3. In the **Select Control** dialog, select **Existing Project Controls** > **CustomerControl - controls** and click **OK**.
4. Click **Ctrl+S** to save your work.

You have just added four lines of code to the Page Flow controller class:

```
import org.apache.beehive.controls.api.bean.Control;
import controls.CustomerControl;

    ...

@Control
private CustomerControl customerControl;
```

These lines declare the **Customer** control on the Page Flow, allowing you to call control methods.

Add a JSF Form and a NetUI Action for Submitting Search Queries

In this step you will add a JSF form (`<h:form>`) for submitting search queries on the customer data.

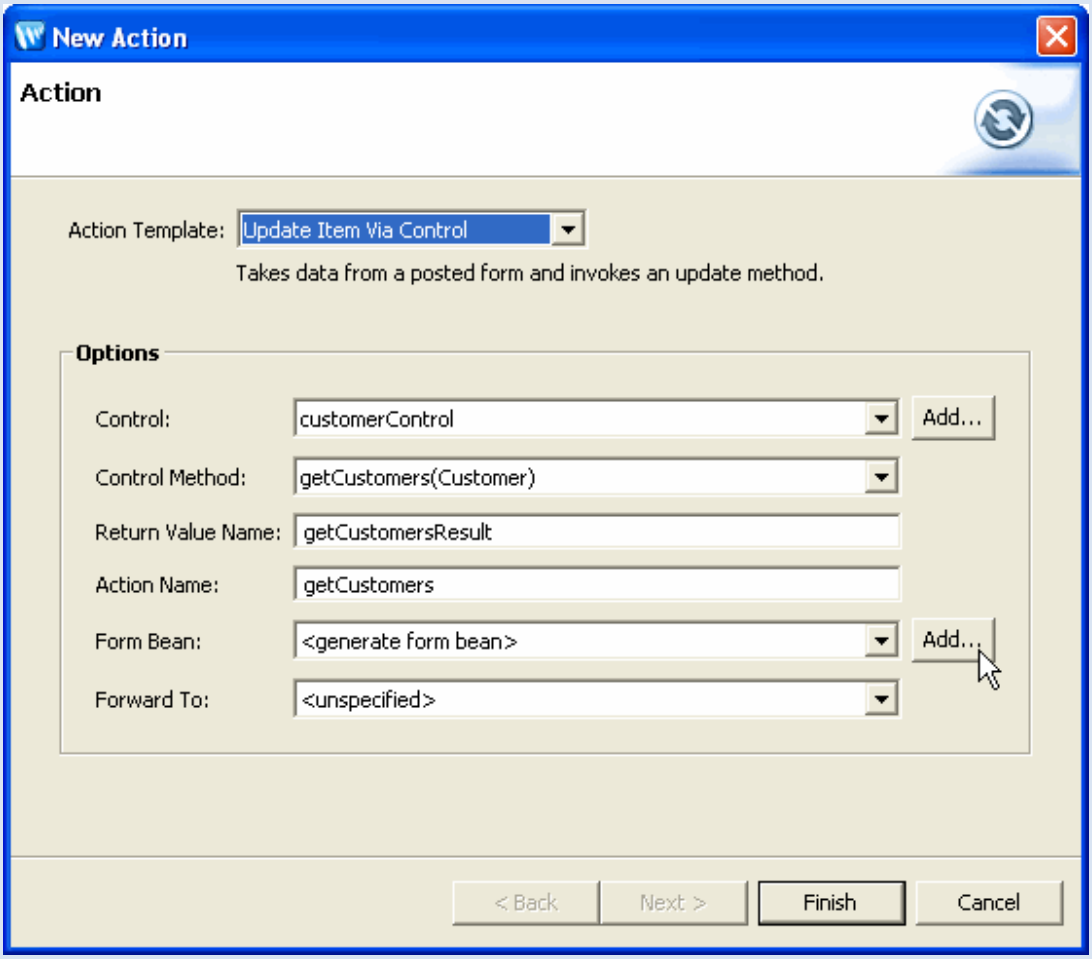
You will also add a new NetUI action (getCustomers) to the controller class. The JSF form will call this action through the form's attribute `action`. This action has a form bean parameter of type Customer: form beans are Java representations of HTML form data.

When a user submits data though the form, the following events occur:

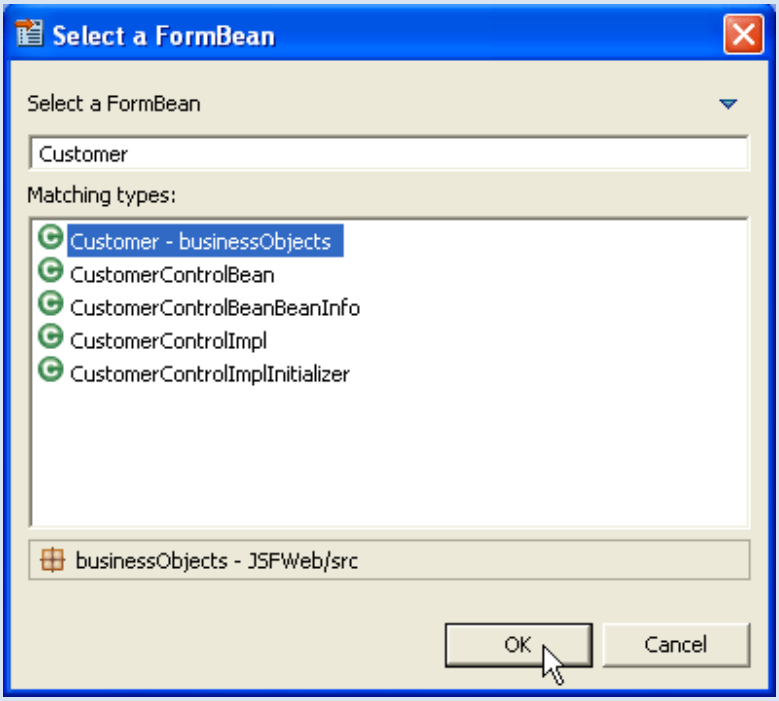
- A **Customer object** (= a form bean) is created based on the submitted data. (This is the responsibility of the JSF backing class.)
- The Customer object form bean is passed to the action **getCustomers action**. (The `<h:form>` tag passes the Customer object.)
- The getCustomers action performs a search based on the properties of the Customer object.

1. On the **Page Flow Explorer** tab, double-click the node **Pages > index.jsp** to open the JSP's source code.
2. From the **JSP Design Palette**, drag **Create Form** into index.jsp's source code. Drop it directly before the **`</f:view>`** element .

   Note: You can accomplish the same thing (creating a new form) by dragging the **getCustomers** method (on the **Page Flow Explorer** view and dropping it directly on top of the index.jsp page (in the **Page Flow Editor**).
3. In the **Create Form** wizard, to create a new action, click **New**.
4. In the **New Action** wizard, in the **Action Template** dropdown field, select **Update Item Via Control**.
   Next to the **Form Bean** field, click **Add**.

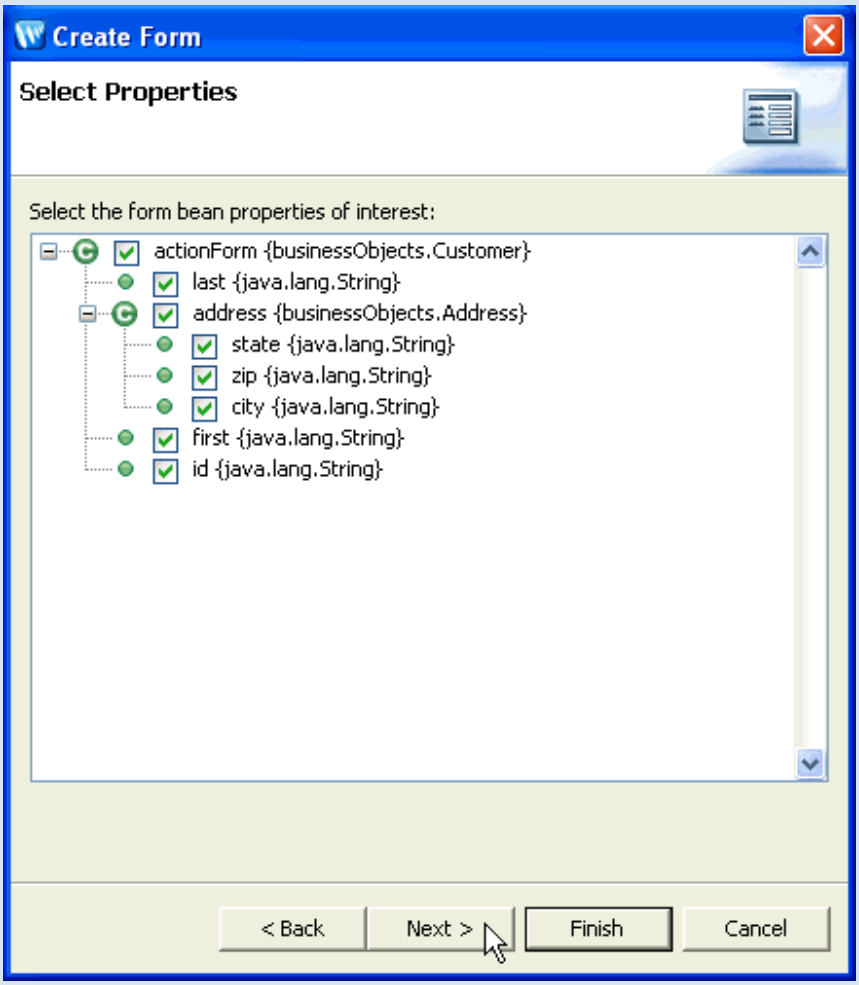


5. In the **Select a FormBean** dialog, type `Customer`.
   Under **Matching Types**, select **Customer - businessObjects**.
   Click **OK**.



6. In the **New Action** dialog, click **Next** and then click **Finish**.
7. In the **Create Form** dialog click **Next**.

8. Confirm that all fields are checked.
   Click **Next**.



9. In the **Create Form** dialog, order the fields in the following sequence:

   id
   first
   last
   address
      city
      state
      zip

   Click **Finish**.

10. Press **Ctrl+Shift+S** to save your work.

You have just added the following form to the page index.jsp.

The form works by constructing a Customer object from the search data entered by the user. The Customer object is constructed by loading the entered data into the the backing bean's from bean: `<h:inputText value="#{backing.formBean1.last}" id="field1" />`. Note that `backing.formBean1` refers is a Customer object field on the backing bean.

The form, with the Customer object attached as an attribute, is then submitted to the NetUI action `getCustomers`.

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

    <h:form>
        ....
        <h:outputLabel value="Last:" for="field1" />
        <h:inputText value="#{backing.formBean1.last}" id="field1" />
```

```
        ...
        <h:commandButton action="getCustomers" value="getCustomers">
                        <f:attribute name="submitFormBean" value="backing.formBean1" />
                </h:commandButton>
        </h:form>
```

The attached form bean is submitted as the action's method parameter:

```
    getCustomers(businessObjects.Customer form)
```

You have also added the following action to the controller file Controller.java.

Notice that the action takes a Customer object parameter--this is the form bean submitted with the form.

```
        @Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "", actionOutputs = { @Jpf.ActionOutput(name
= "getCustomersResult", type = java.util.ArrayList.class, typeHint = "java.util.ArrayList<businessObjects.
Customer>") }) })
        public Forward getCustomers(businessObjects.Customer form) {
                Forward forward = new Forward("success");
                businessObjects.Customer criteria = form;
                java.util.ArrayList<businessObjects.Customer> getCustomersResult = customerControl
                                .getCustomers(criteria);
                forward.addActionOutput("getCustomersResult", getCustomersResult);
                return forward;
        }
```

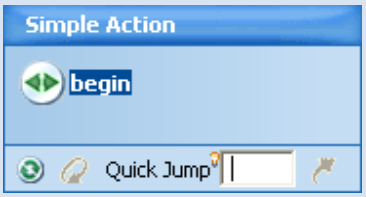
Add a JSF Page that Displays Query Results

In this step you will create a new JSF page and add a JSF tags for displaying query results.

You will add a <h:dataTable> tag that renders an HTML table when appropriate data is passed to it. In this case, a java.util.ArrayList of Customer objects is passed to the <h:dataTable> tag. The tag iterates over the Customer objects rendering each object as a row in a standard HTML table.

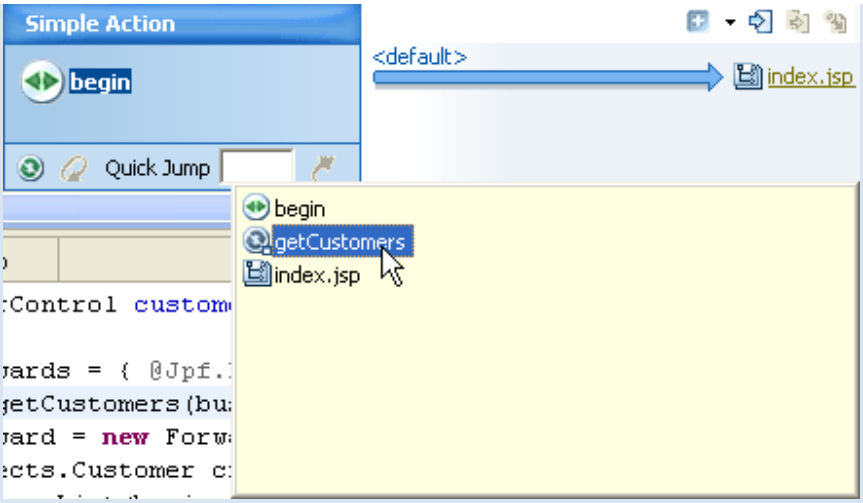
Note that when you create a new JSF page, Workshop for WebLogic automatically creates the page's backing Java bean.

1. On the **Page Flow Editor** view, place the cursor in the field labeled **Quick Jump**.

   Press **Ctrl+Space** to bring up the content assistant dropdown.

   Double-click **getCustomers**.



The **getCustomers** action will be given focus in the **Page Flow Editor**.

2. In the **Page Flow Explorer** tab, right-click the **Pages** node and select **New JSF Page**.
3. Name the page `customers.jsp` and press **Enter**.
4. In the **Rename 'newPage1.java to 'customers'** dialog, click **Continue**.

   At this point Workshop for WebLogic creates both (1) the page customers.jsp and (2) the backing Java class customer.java. (To examine the backing class, right-click the page and select **Open Backing File**.)
5. Drag **customers.jsp** from the **Page Flow Explorer** viewto the **Page Flow Editor** tab. Drop it directly on the **unspecified** node.
6. In the **Add Page Inputs** dialog, click **OK**. (To learn more about page inputs, see Tutorial: Accessing Controls from a Web Application: Step 3: Create a Data Grid.)
7. On the **Page Flow Explorer** view(don't confuse this with the **Page Flow Editor**), double-click **customers.jsp** to open its source code.
8. From the **JSP Data Palette** drag **getCustomersReport** into the source view for **customer.jsp**. Drop it directly before the **</f:view>** tag.
9. In the **Data Display Wizard**, confirm that all fields are checked and click **Finish**.
10. Click **Ctrl+Shift+S** to save your work.

You have just added the following code to the customers.jsp file.

Notice that the data table gets it's input data through the *NetUI* implicit object `pageInput`. This is one of the most common ways to integrate Beehive NetUI and JSF technologies. For information about integrating these technologies, see Integrating Java Server Faces into a Web Application.

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0"
        prefix="netui-data"%>
<netui-data:declarePageInput required="true"
        type="java.util.ArrayList<businessObjects.Customer>"
        name="getCustomersResult" />


<html>
<head>
</head>
<body>
<f:view>
        <f:verbatim>
                <p>Beehive NetUI-JavaServer Faces Page -
                ${pageContext.request.requestURI}</p>
        </f:verbatim>
```

```
        <h:dataTable value="#{pageInput.getCustomersResult}" var="item0"
                border="1">
                <h:column>
                        <f:facet name="header">
                                <h:outputLabel value="Last" />
                        </f:facet>
                        <h:outputText value="#{item0.last}" />
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputLabel value="Address" />
                        </f:facet>
                        <h:panelGrid columns="2">
                                <h:outputLabel value="State: " />
                                <h:outputText value="#{item0.address.state}" />
                                <h:outputLabel value="Zip: " />
                                <h:outputText value="#{item0.address.zip}" />
                                <h:outputLabel value="City: " />
                                <h:outputText value="#{item0.address.city}" />
                        </h:panelGrid>
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputLabel value="First" />
                        </f:facet>
                        <h:outputText value="#{item0.first}" />
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputLabel value="Id" />
                        </f:facet>
                        <h:outputText value="#{item0.id}" />
                </h:column>
        </h:dataTable>
</f:view>
</body>
</html>
```

You have also specified the navigational target of the getCustomers action:

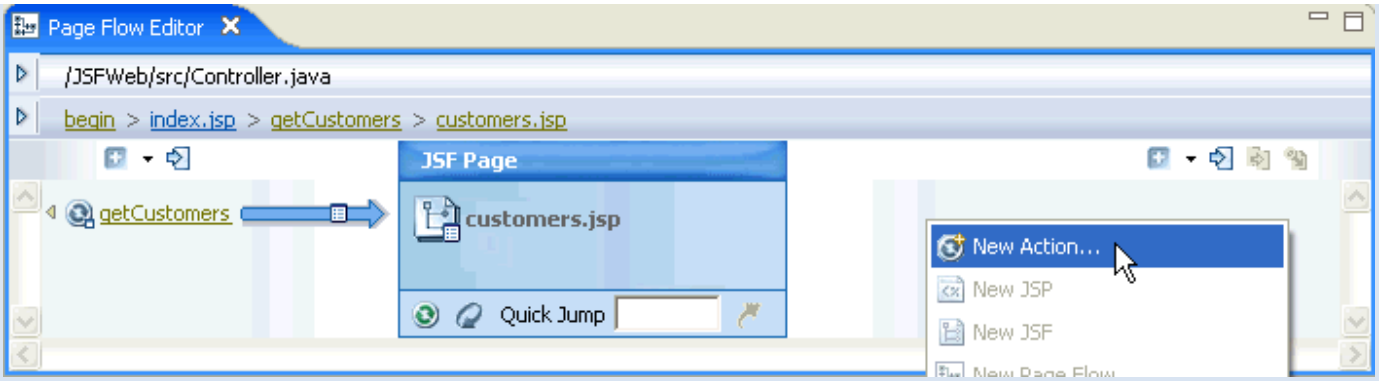
```
        @Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "customers.faces", actionOutputs = { @Jpf.
ActionOutput(name = "getCustomersResult", type = java.util.ArrayList.class, typeHint = "java.util.ArrayList") }) })
        public Forward getCustomers(businessObjects.Customer form) {
                Forward forward = new Forward("success");
                businessObjects.Customer criteria = form;
                java.util.ArrayList<businessObjects.Customer> getCustomersResult = customerControl
                                .getCustomers(criteria);
                forward.addActionOutput("getCustomersResult", getCustomersResult);
                return forward;
        }
```

Notice that the action forwards to the `customers.jsp` page using the `.faces` file extension: `path = "customers.faces"`.

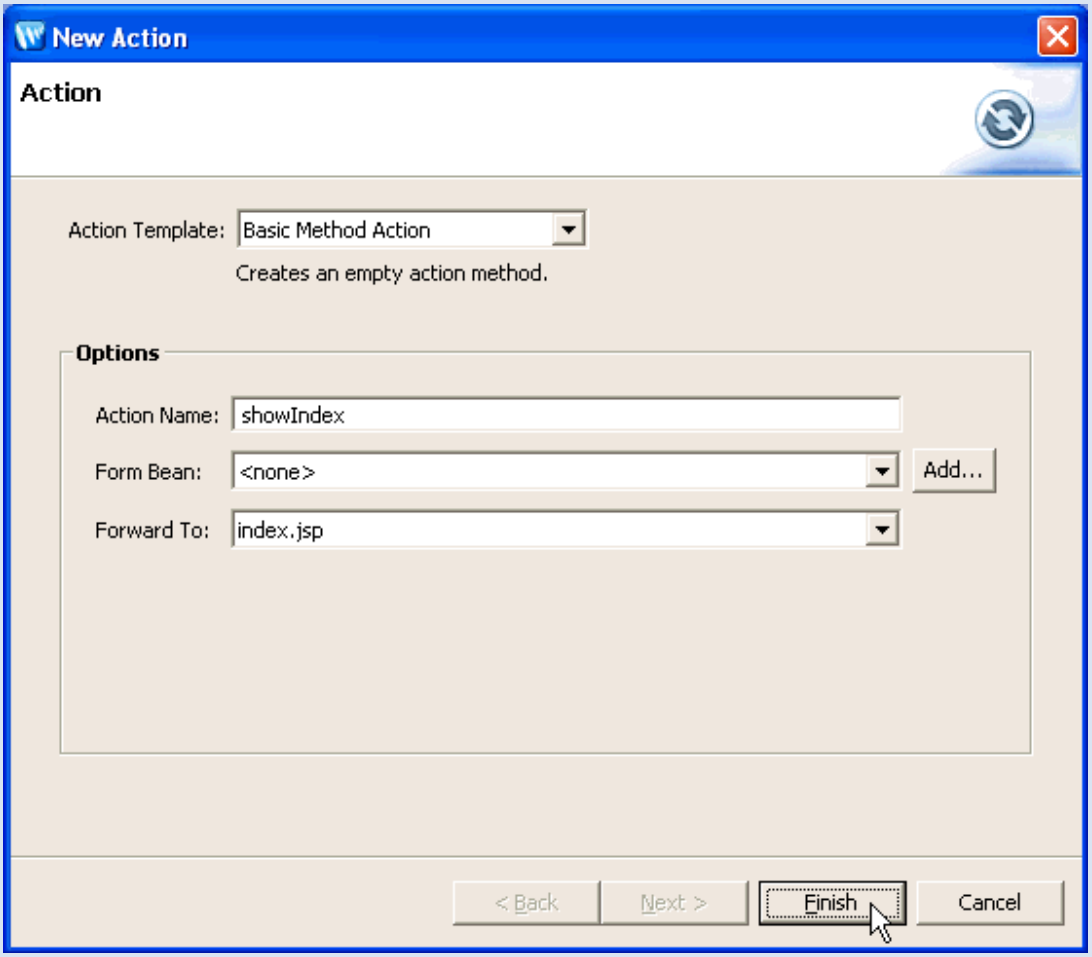
Add a Link Back to the Search Form Page

In this step you will add a link on the results page that will navigate the user back to the search form page. The link you add will be a JSF link that directly raises a NetUI action.

1. On the **Page Flow Editor** view, click the **customer.jsp** node so that customer.jsp is displayed in the center pane of the view.
2. On the **Page Flow Editor** view, right-click in the right-hand side of the view (also called the "downstream" pane) and select **New Action**.



3. In the **New Action** dialog, in the **Action Name** field, enter showIndex.
   In the **Form Bean** field, confirm that **<none>** is selected.
   In the **Forward To** field, select **index.jsp**.
   Click **Finish**.



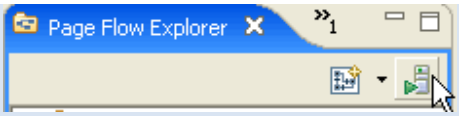
4. On the **Page Flow Explorer** view, double click the **customers.jsp** node to open its source code.
5. Add the following code to customer.jsp directly after the </h:dataTable> tag.

```
<h:form>
    <h:commandButton action="showIndex" value="Back to Search Page"/>
</h:form>
```

6. Press **Ctrl+Shift+S** to save your work.

Run the Web Application

1. On the **Page Flow Explorer** view, click the server icon to deploy and run the Page Flow.



2. In the Run On Server view, click **Finish**.

   Wait while the application compiles, the server starts, and the application is deployed.

3. Enter search criteria in the fields provided and click the **getCustomers** button.

   Note: you can use partial First or Last names only as search criteria on the input form. Submit a blank form to retrieve all customers.

Related Topics

Integrating Java Server Faces into a Web Application

Click one of the following arrows to navigate through the tutorial:

Summary: Java Server Faces Integration

In this tutorial you learned:

- how JSF creates user interfaces for web applications
- how Beehive NetUI provides backend event handling for a web application
- how JSF and Beehive NetUI can work together in a web application

Further Information

Sun Site: JavaServer Faces Technology

Beehive Documentation: Java Server Faces

dev2dev Site: Integrating JavaServer Faces with Beehive Page Flow

Related Topics

None.

Click the arrow below to navigate through the tutorial:

Building Web Applications: Introduction

BEA Workshop for WebLogic Platform provides tooling support for NetUI: the Apache Beehive framework for web applications. This topic explains the basic concepts behind Beehive NetUI.

Why Use Beehive NetUI?

By using Beehive NetUI, 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

NetUI allows 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, business logic, and the core functionality of the web application can be implemented in Java controller classes, which form the nerve center of your web application.

The basic division of labor between JSP files and controller classes can be summarized as follows: Java controller classes implement the functionality of the web application; JSP files surface that functionality to the user.

The presentation and processing aspects of a Beehive NetUI web app are highly modular: it's easy to change one without impacting the other. For example, its easy to change the look and feel of the web app by updating the JSP pages with little or no changes required to the underlying controller classes. Similarly, you can re-implement the controller classes without changing the JSP pages, because the core functionality of the web app is encapsulated in the controller classes instead of spread throughout the JSP pages.

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 Java class, instead of having to search through many JSP files and make multiple updates. You can also encapsulate similar web application functions in single Java classes, creating 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.

Components of the Beehive NetUI Programming Model

This section gives an overview of the basic parts of the Beehive NetUI implementation.

JSP Files

JSPs form the user interface of a NetUI web application, without the need to include Java code snippets on those pages. In a Beehive NetUI web app, the JSP pages contain JSP tags and references to JSP implicit objects, but no Java code. This makes the application behavior more predictable, testable, and it allows for stricter separation of labor between Java code developers and JSP developers.

Beehive NetUI provides the JSP developer special libraries of JSP tags, the <netui> tag libraries, that supplement the functionality of the standard JSP tag libraries.

The <netui> tag library contains JSP tags specifically designed to work with controller classes (see below). 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 refer to data in the controller class. The <netui> tags allow 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.

Java Server Faces (JSF) files can also be added to your web application, either as a replacement or complement to the JSPs.

Controller Classes

Data processing code is contained in Java classes called **controller** classes. Controller classes handle use navigation through the JSPs in the web application, handle user data submissions, call external resources such as web services and backend databases, and generally implement the core functionality of the web application.

For more infomation on the syntax of Controller classes, see the Apache Beehive documentation: Page Flow Controllers

Actions

Actions are methods in the Controller class that has been decorated with specially designed set of **metadata annotations**, or "annotations" for short. Annotations, a new feature in Java 5, are property setters for methods or classes. Beehive NetUI defines its own set of annotations that allow the controller class to easily communicate with the JSP pages, control navigation with the web application and manage application state.

For more infomation about actions, see the Apache Beehive documentation: Fleshing out the Controller and Actions in NetUI.

Page Flows

JSPs and Controller classes are arranged in modular units called **page flows**. A page flow consists of a single controller class and any number of JSP files. Typically, a single page flow reflects some unit of functionality within a web application. For example, a company's web application might contain many different page flows, one for browsing the company's catalogue of products, another for collecting the products in a shopping cart, and another for managing customer accounts.

For more information on Page Flow modules, see the Apache Beehive documentation: Nested Page Flows, Page Flow Inheritance, and Shared Flow.

Implicit Objects

Beehive NetUI provides two types of implicit objects that can be used to move data around the application and save application state.

1. JSP implicit objects: these are the standard set of JSP objects provided by the JSP implementation, such as session, pageContext, etc.
2. NetUI implicit objects: these objects are provided by the Beehive NetUI framework allowing access to objects in the Controller class, etc.

   For a list of the available objects see: Data binding to NetUI Implicit Objects in the Apache Beehive documentation.

Form Beans

Form Beans are a Java representation of a HTML form. When a user submits an HTML form, the submitted data is captured as a Form Bean and (typically) is passed to an action for further processing.

For more information on Form Beans and their role in a NetUI web application, see NetUI Form Control Tags in the Apache Beehive documentation.

Validation and Exception Handling

Validation and exception handling are defined using a declarative programming model using annotations. For more information see Validation and Exception Handling in the Apache Beehive documentation.

Related Topics

NetUI: Getting Started

The Page Flow Perspective

This topic describes Workshop for WebLogic's tooling features for building Beehive NetUI web applications.

Workshop for WebLogic provides a variety of views and graphical user interface tools to help you design, conceptualize and implement NetUI web applications.

Individual icons used in the Page Flow Perspective are described in the Page Flow Perspective Visual Glossary.

Page Flow Perspective

The **Page Flow Perspective** gives a graphical summary of an individual page flow.

Open the Page Flow Perspective by selecting **Window > Open Perspective > Page Flow**.

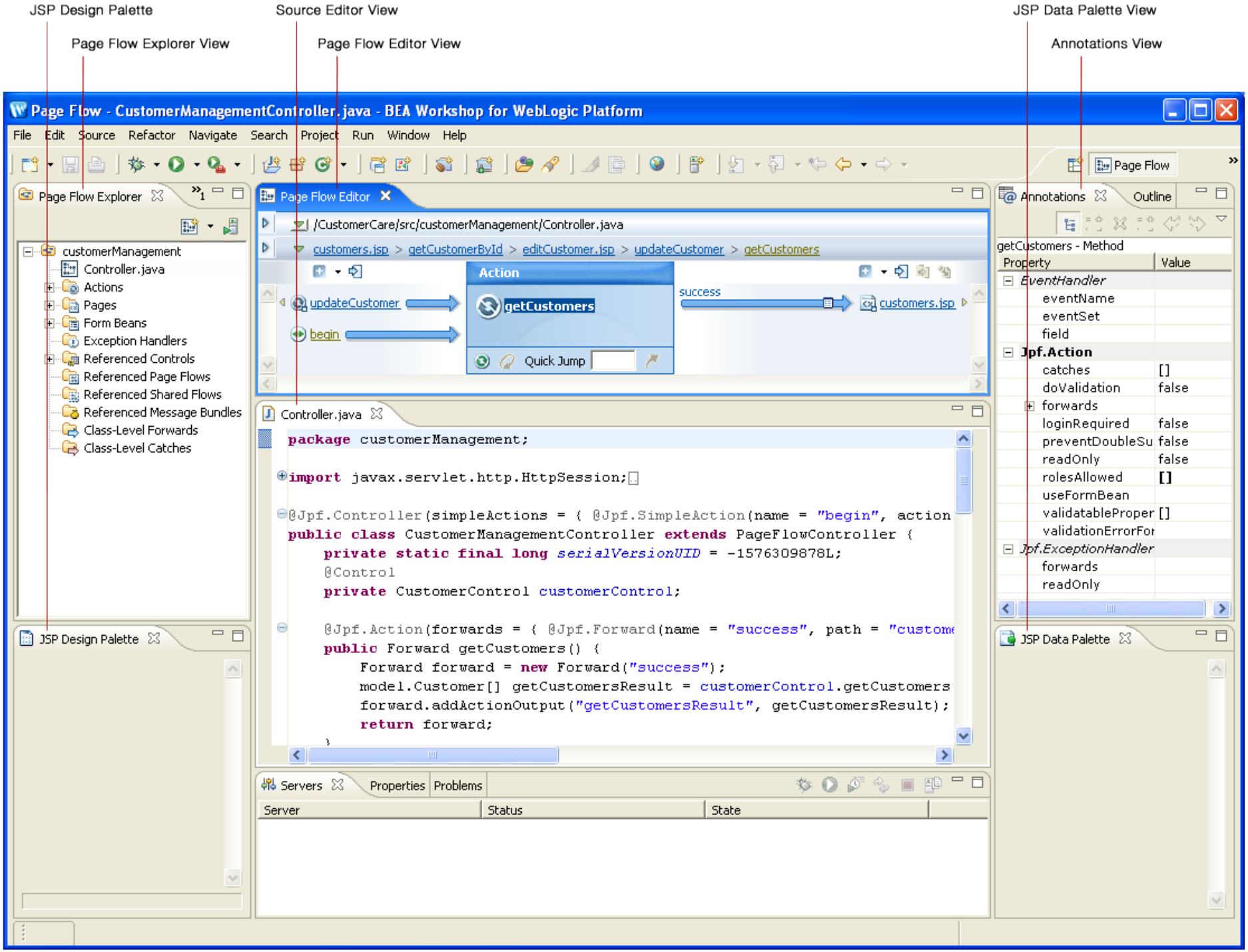
> **Note:** If you already have a page flow-related file open, the Page Flow Perspective will display that file's page flow. If you don't have a page flow-related file open, the Page Flow Perspective opens to the first page flow in the first page flow-enabled project that it finds. To switch the page flow displayed, you must explicitly switch to another page flow through one of the views, or make a page flow-related file from a different page flow the active document in the Source Editor View.

The Page Flow Perspective consists of these views:

- **Page Flow Explorer View**: shows a view of the functional parts of the current page flow
- **Page Flow Editor View**: shows a graphical view of a specific page flow node (action or page) and its neighboring nodes
- **Page Flow Overview**: shows a diagram of the navigational structure between actions and pages
- **Source Editor View**: shows the Java source of a page flow artifact
- **Annotations View**: shows the annotation currently selected in one of the above views
- **JSP Design Palette**: shows available design elements that can be added to the current JSP page
- **JSP Data Palette**: shows available data elements that can be added to the current JSP page

The following diagram shows the default locations for these views when the Page Flow Perspective is first opened. Only those views that are page flow-specific are described below. Other views, such as the Servers and Problems views are displayed by default, but they are not specifically designed to show page flow-related information.

Each of these views is described in detail below.



Page Flow Explorer View

For more information see Page Flow Explorer View.

Page Flow Editor View

For more information see Page Flow Editor View.

Page Flow Overview

For more information see Page Flow Overview.

Source Editor View

For more information see Source Editor View.

Annotations View

For more information see Annotations View.

JSP Design Palette

For more information see JSP Design Palette.

JSP Data Palette

For more information see JSP Data Palette View.

Related Topics

The following tutorials use many of the views and wizards described above:

Tutorial: Accessing a Database from a Web Application

Tutorial: Java Server Faces Integration

Also see the following topic:

Page Flow Perspective Visual Glossary

Integrating Java Server Faces into a Web Application

Java Server Faces (JSF) is a web user interface technology that can be used to supplement the user interface technology native to Beehive NetUI (the <netui> tag library).

Enabling JSF in a Web Project

To install the default JSF implementation, add the JSF facet to your web project. (**Project > Properties > Project Facets > Add/Remove Project Facets > place check next to JSF**).

Adding the JSF facet will install JSF Reference Implementation 1.1.

Integrating JSF and Beehive NetUI

Beehive NetUI and JSF can be fully integrated in a web application. Below are described the most typical ways to make the two frameworks communicate.

Forwarding from a NetUI Action to a JSF Page

To forward from a NetUI action to a JSF page, refer to the JSF page with the `.faces` file extension, even though Workshop for WebLogic creates JSF pages on disk with the `.jsp` file extension.

Suppose you have a JSF page named `myJSFPage.jsp`. To forward to this page from an action, use the following syntax:

```
@Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "myJSFPage.faces") } )
public Forward navigate() {
        return new Forward("success");
}
```

Raising NetUI Actions from JSF Pages

JSF pages can raise NetUI actions through the *action* attribute.

For example, assume you have the following action in a NetUI controller file.

```
@Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "myJSFPage.faces") } )
public Forward navigate() {
        return new Forward("success");
}
```

You can invoke this action from a JSF by referencing `navigate` in an `action` attribute:

```
<h:form>
...
    <h:commandButton action="navigate" value="Go"/>
</h:form>
```

Raising NetUI Actions from JSF Backing Beans

Suppose you have an action `navigate` in a Controller class. To invoke `navigate` from within a JSF backing bean, use a `command` handler decorated with the annotation set `@Jpf.CommandHandler/@Jpf.RaiseAction`:

```
    @Jpf.CommandHandler(
        raiseActions={
            @Jpf.RaiseAction(action="navigate")
        }
    )
    public String invokeNavigate()
    {
        return "navigate";
    }
```

You bind to the command handler from the JSF page in the usual way:

```
    <h:commandButton action="#{backing.invokeNavigate}" value="Go"/>
```

Calling Controls from JSF Backing Beans

You call a control from a backing bean just as you would call a control from any Java class.

First you declare the control on the client Java class.

```
    import org.apache.beehive.controls.api.bean.Control;

        ...

    @Control
    private CustomerControl customerControl;
```

Then you invoke methods on that control.

```
    public Customer[] getCustomers() {
                return customerControl.someMethod();
        }
```

Passing Data Between JSF Pages and NetUI Actions

JSF pages can reference NetUI implicit objects using JSF expressions. For example, the following JSF page receives a page input from the NetUI controller class:

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0" prefix="netui-data"%>

<netui-data:declarePageInput required="true" type="java.util.ArrayList<businessObjects.Customer>"
name="getCustomersResult" />

...

    <h:dataTable value="#{pageInput.getCustomersResult}" var="item0" border="1">

        ...

    </h:dataTable>
```

References are not limited to the pageInput implicit object; you can reference any implicit object using JSF-style expressions. For example, the following expression references the foo field on the controller class:

```
    <h:outputText value="#{pageInput.foo}"/>
```

You can also submit data (as a form bean) from a JSF page to a NetUI Controller class.

Suppose you have an action that has a form bean parameter:

```
        @Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "confirm.faces") } )
        public Forward getCustomers(Customer form) {

                // do something with the submitted form data...

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

A form bean a Java representation of an HTML form, where the bean properties correspond to the fields in the HTML form.

```
public class Customer implements Serializable
{
    private String first = "";
    private String last = "";

    public Customer()
    {
    }

    public Customer(String first, String last)
    {
        this.first = first;
        this.last = last;
    }

    public void setFirst(String value)
    {
        first = value;
    }

    public String getFirst()
    {
        return first;
    }

    public String getLast()
    {
        return last;
    }

    public void setLast(String value)
    {
        last = value;
    }
}
```

To submit this form bean to the action from a JSF page, reference the bean with a JSF style expression:

```
    <h:form>
        <h:outputLabel value="First:" for="field1" />
        <h:inputText value="#{backing.custFormBean.first}" id="field1" />
        <h:outputLabel value="Last:" for="field2" />
        <h:inputText value="#{backing.custFormBean.last}" id="field2" />
        <h:commandButton action="getCustomers" value="Submit">
```

```
            <f:attribute name="submitFormBean" value="backing.custFormBean" />
        </h:commandButton>
    </h:form>
```

Note that the form bean is reference via the backing bean. See the italic expressions above: `#{backing.custFormBean.first}`, `#{backing.custFormBean.last}`, and `backing.custFormBean`.

For these expressions to work, the backing bean must include the form bean as a field, with appropriate setters and getters on that field:

```
@Jpf.FacesBacking()
public class index extends FacesBackingBean {

    private Customer custFormBean = new Customer();

    public Customer getCustFormBean() {
        return custFormBean;
    }

    public void setCustFormBean(Customer bean) {
        this.custFormBean = bean;
    }
}
```

Other Integration Scenarios

Other integration scenarios are described in the document Integrating JavaServer Faces with Beehive Page Flow.

Mixing JSF and NetUI Tags

Mixing Beehive NetUI tags, or any JSP tags, with JSF tags can lead to surprising results. You should have a good understanding of the particular tags you are using before you mix the different tag libraries.

An exception you do not need to worry about is the use of the Beehive NetUI `<netui:declarePageInput>` tag. This tag can be used freely with JSF tags because it only sets up a contract with the NetUI controller class but not affect the view in any other way.

Workshop for WebLogic JSF Tooling Features

Workshop for WebLogic offers development support for may common JSF coding tasks, including:

1. automatic generation of backing beans
2. JSF-specific code generation for forms and data grids
3. support for authoring command handlers

To activate JSF development support you must be in the Page Flow perspective (**Window > Open Perspective > Page Flow**).

JSF-Specific Code Generation Through the JSP Data Palette

In the Page Flow perspective, the Data Palette supports JSF tags and JSF style expressions when composing JSF pages.

Note: the Data Palette recognizes JSF pages by the presence of the JSF tag `<f:view>` on the page. If the following

tag (and its associated library declaration) is present on the page, then the Data Palette will generate code in JSF mode:

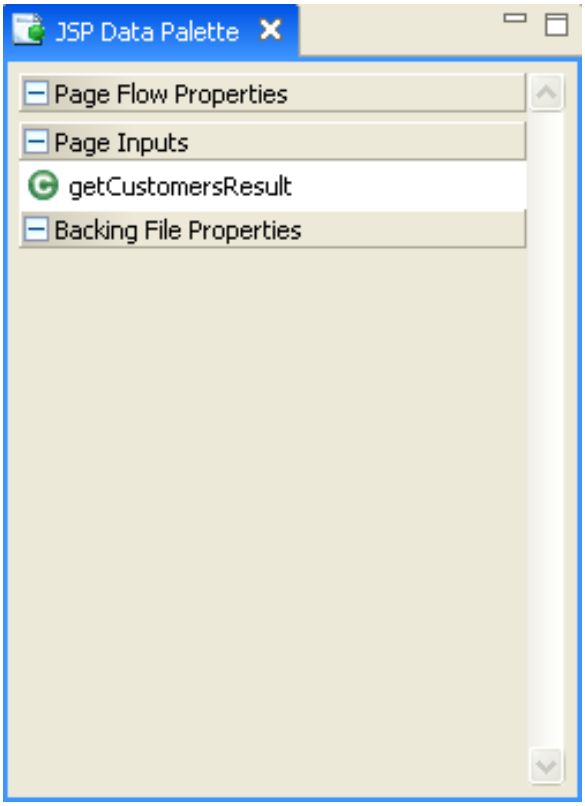
```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
...
<f:view>
```

Note that any prefix value is acceptable; the prefix value 'f' is shown only because it is the default value.

For example, suppose you have a JSF page with a page input declaration:

```
<netui-data:declarePageInput
        type="java.util.ArrayList<businessObjects.Customer>"
        name="getCustomersResult" />
```

The presence of a page input declaration will activate the Data Palette with a corresponding node:



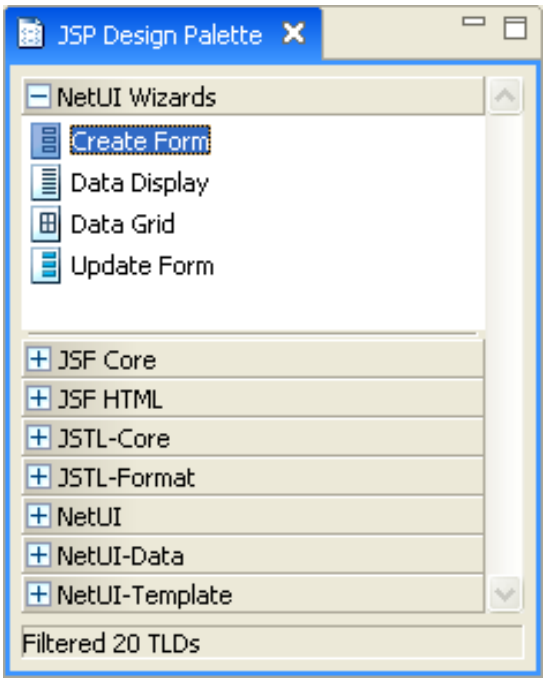
When this node is dragged and dropped onto the JSF page, JSF tags are JSF style expressions are used to construct the data display structures. For example:

```
<h:dataTable value="#{pageInput.getCustomersResult}" var="item0"
                border="1">
                <h:column>
                        <f:facet name="header">
                                <h:outputLabel value="Name" />
                        </f:facet>
                        <h:outputText value="#{item0.name}" />
                </h:column>
        </h:dataTable>
```

Workshop for WebLogic will create outputText fields for simple properties and launch the Data Display Wiz for complex and/or repeating type properties.

JSF-Specific Code Generation Through the JSP Design Palette

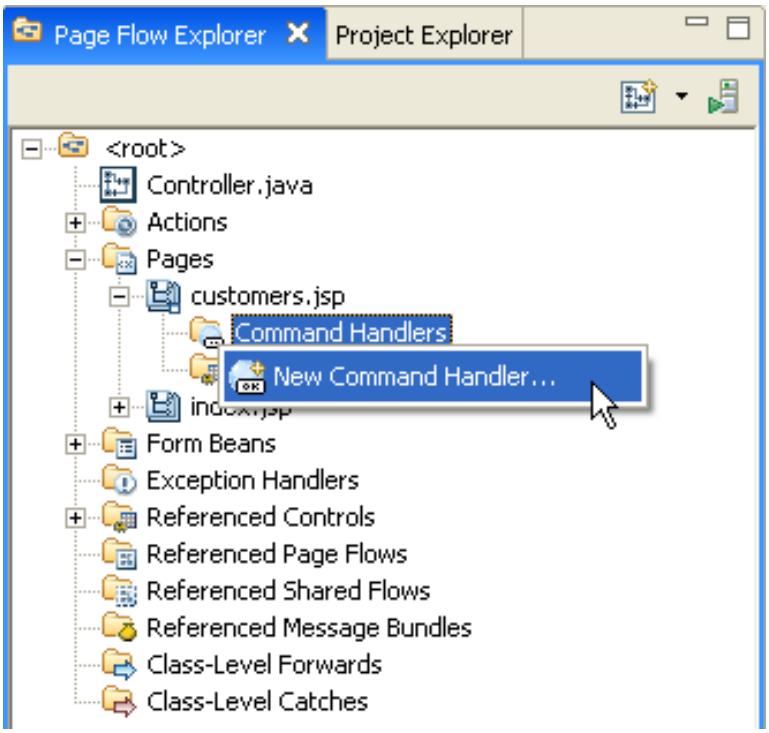
Similar support is provided for composing JSF forms through the Design Palette.



When a page contains a declaration for the core JSF core library (`<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>`), the the Design Palette will be in 'JSF mode'. Forms and data grids created from the Design Palette will use JSF tags and JSF expressions.

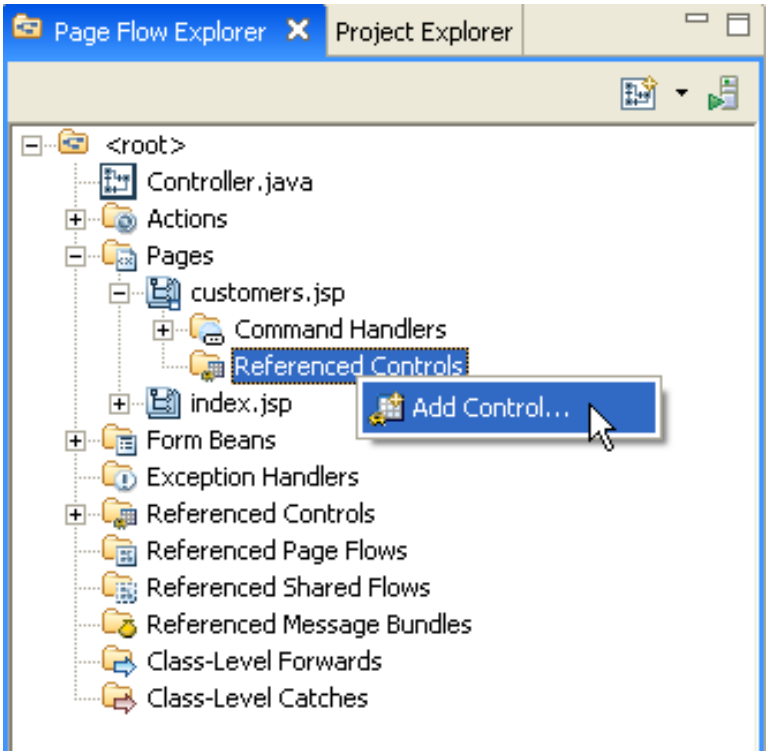
JSF Command Handler Support

You can easily add command handlers to a JSF backing bean by right-clicking the **Command Handler** node and selecting **New Command Handler**. (You must be in the Page Flow perspective to see the Command Handler node.)



The wizard allows you to setup command handler method that can raise actions in the controller class. The wizard also lets you specify a form bean that can be passed along to the raised action. For details on passing form bean data from a JSF page to a controller action, see Passing Data Between JSF Pages and NetUI Actions above.

You can also add control references to a backing bean in a similar manner:



Related Topics

dev2dev documentation: Integrating JavaServer Faces with Beehive Page Flow

Tutorial: Java Server Faces Integration

Web Application Technologies

This topic lists the versions and locations of the web application technologies used by BEA Workshop for WebLogic Platform.

Web Application Technologies Versions

The following table lists the versions of standard web technologies used by Workshop for WebLogic.

The JAR resources listed below are made available to a web application through **library modules**, essentially JARs packaged as WARs and EARs. You add these library modules to your web application by adding the corresponding facet to your web application. For instance to add the JSF library module, right-click your project and select **Properties** > **Project Facets** > **Add/Remove Project Facets** > **[Place a check next to JSF]**.

Technology	Version	Library Module Location	JARs
Struts	1.2	BEA_HOME/ weblogic92/ common/ deployable-libraries/struts-1.2.war	struts.jar
Beehive NetUI	1.0.1 (see Beehive Version note below)	BEA_HOME/ weblogic92/ common/ deployable-libraries/beehive-netui-1.0.war	beehive-netui-core.jar, beehive-netui-tags.jar
Beehive Controls	1.0.1 (see Beehive Version note below)	BEA_HOME/ weblogic92/ common/ deployable-libraries/beehive-controls-1.0.war	beehive-controls.jar, beehive-ejb-controls.jar, beehive-jdbc-controls.jar, beehive-jms-controls.jar
JSTL (JSP Standard Tag Library)	1.1	BEA_HOME/ weblogic92/ common/ deployable-libraries/beehive-jstl-1.1.war	jstl.jar, standard.jar
JSF (Java Server Faces)	1.1.01 (see JSF Implementations note below)	BEA_HOME/ weblogic92/ common/ deployable-libraries/jsf-1.1.war	jsf-api.jar, jsf-impl.jar

JSF Implementations

WebLogic Platform ships two JSF implementations: (1) Sun's reference implementation 1.1.01 and (2) MyFaces 1.1.1. Workshop for WebLogic uses Sun's reference implementation 1.1.01 by default when the JSF facet is added to a web project.

Beehive Version

The version of Beehive is 1.0.1 with some minor local fixes made by BEA. These fixes will be rolled back into the Apache Beehive code base at a later date.

Related Topics

none

Authoring Web-based User Interfaces

The following topics explain how to author web-based user interfaces using Workshop for WebLogic.

Workshop for WebLogic provides support for the following user interface technologies: (1) the JavaServer Pages Standard Tag Library and (2) the Beehive NetUI tag libraries (3) Java Server Faces and (4) Tiles.

Overview: NetUI Tag Libraries
: Explains the contents of the three NetUI tag libraries.

Creating Forms for Collecting User Data
: Explains how to create forms for user input.

Displaying Data with NetUI Data Grids
: Explains how to display tabular data using data grids.

Validating User Input Data
: Explains how to use Workshop for WebLogic's validation tools.

Using Tiles
: Explains how to use Tiles technology in a Beehive NetUI web application.

Rendering Trees
: Explains how to render HTML trees.

Controlling Web Application Look and Feel with JSP Templates
: Explains how to use JSP templates in a web application.

Authoring JSP Template Projects and Populating the Default Template List
: Explains how to author JSP template projects.

Related Topics

NetUI Tag Library Overview

Overview: Beehive NetUI Tag Library

Beehive NetUI provides three tag libraries:

1. core HTML library: renders basic HTML elements
2. data grid library: renders tables and filterable/sortable data grids
3. JSP template library: renders reusable page elements such as headers, footers, etc.

These three libraries are described in more detail below.

The Core HTML Tag Library

To use the core HTML library, enter the following declaration on your JSP page:

```
<%@taglib uri="http://beehive.apache.org/netui/tags-html-1.0" prefix="netui"%>
```

For a list of the tags available see:

netui Library

The Data Grid Tag Library

To use the data grid library, enter the following declaration on your JSP page:

```
<%@taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0" prefix="netui-data"%>
```

For a list of the tags available see:

netui-data Library

The Template Tag Library

To use the template library, enter the following declaration on your JSP page:

```
<%@taglib uri="http://beehive.apache.org/netui/tags-template-1.0" prefix="netui-template"%>
```

For a list of the tags available see:

netui-template Library

Related Topics

NetUI Tag Library Overview

Creating Forms for Collecting User Data

The following topic describes how Beehive NetUI supports the submission of user data.

HTML Forms and Form Beans

Suppose you want you web application to collect data from users, such as the user's name, email, etc. Beehive NetUI supports user data submission through a three step process: (1) First the user enters data into an ordinary HTML form. (2) Upon submission that data is loaded into a Java object called a **form bean**. (3) Once the submitted data has been packaged as a form bean, the Controller class is free to operate on the data: typically the form bean is passed to one of the Controller class's action methods for further processing.

Form beans are Java representations of the of user-facing HTML form. In particular they are standard JavaBean representations of HTML forms: for each data field in the HTML form, the form bean has a corresponding member field and getter/setter methods. For example, the following HTML form has two data fields: firstname and lastname.

```
  <netui:form action="updateCustomer">
              <netui:textBox dataSource="actionForm.customer.firstName" id="field2"></netui:textBox>
              <netui:textBox dataSource="actionForm.customer.lastName" id="field3"></netui:textBox>
  </netui:form>
```

Its corresponding form bean has two member fields, the Strings firstname and lastname, each with setter/getter fields:

```
public class Customer implements Serializable {

        private String firstName = "";

        private String lastName = "";

        public Customer(String firstName, String lastName) {
                super();
                this.firstName = firstName == null ? "" : firstName;
                this.lastName = lastName == null ? "" : lastName;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
```

An instance of this form bean gets passed to the action method for further processing by the Controller class.

```
public Forward updateCustomer(Customer form)
{
    ...
}
```

For more infomation about form beans, HTML forms and action methods see the Apache Beehive documentation Handling Forms.

Repeating Form Elements

The Beehive NetUI tag libraries supports advanced form element repeater tags. These tags allow you to render forms dynamically. For more information on dynamically rendered repeating forms, see the Apache Beehive documentation NetUI Repeating Form Control Tags.

Using the Create Form Wizard

Workshop for Weblogic Platform provides powerful tools for building Beehive NetUI forms. For more information see Tutorial: Accessing Controls from a Web Application: Step 4: Create a Page to Edit Customer Data.

Related Topics

NetUI Form Control Tags

NetUI Repeating Form Control Tags

Displaying Data with NetUI Data Grids

BEA Workshop for WebLogic Platform provides tools for creating Beehive NetUI data grids. Data grids provide a powerful way for users to interact with tabular data, such as a record set from a database. For example, a data grid can render a record set as a sortable and filterable HTML table.

Data grids are rendered using the Beehive NetUI tag <netui-data:dataGrid> and its associated children tags. To render a record set as an HTML table, pass a data set (for example, an Array of objects) to the <netui-data:dataGrid> tag's dataSource attribute:

```
<netui-data:dataGrid dataSource="pageInput.employeeArray" name="employeeGrid">
```

For more information about the <netui-data:dataGrid> syntax see Beehive NetUI Data Grids and netui-data:dataGrid Tag

Using the Data Display Wizard

Workshop for WebLogic provides a wizard that can define data grid properties. For more information using the data grid wizard see Tutorial: Accessing Controls from a Web Application: Step 3: Create a Data Grid

Using the <netui-data:repeater> and Related Tags

Related Topics

Beehive NetUI Data Grids

Sorting and Filtering in a Data Grid

Validating User Input Data

The Beehive NetUI tag libraries provide annotations for validating form data submitted by users.

You can develop validation processes by working with the validation annotations directly in source code or you can use Workshop for WebLogic's validation tools. Workshop for WebLogic's validation tools provide a graphical user interface for developing validation processes.

Related Topics

Validation Rules Dialog

Set Message Bundle Dialog

Apache Beehive documentation: Validation

Using Tiles

Apache Beehive supports Struts Tiles technology. Struts Tiles allows to you reuse common web application components such as menu bars, headers, and footers.

For more information about support for Struts Tiles see Tiles Support.

Related Topics

Tiles Support

Rendering Trees

The Beehive NetUI tag libraries provide tags for rendering tree structures, allowing you to display a list of links arranged as expandable/collapsible tree nodes.

For more infomation on rendering trees, see the Apache Beehive documentation Tree Tags.

Related Topics

Tree Tags

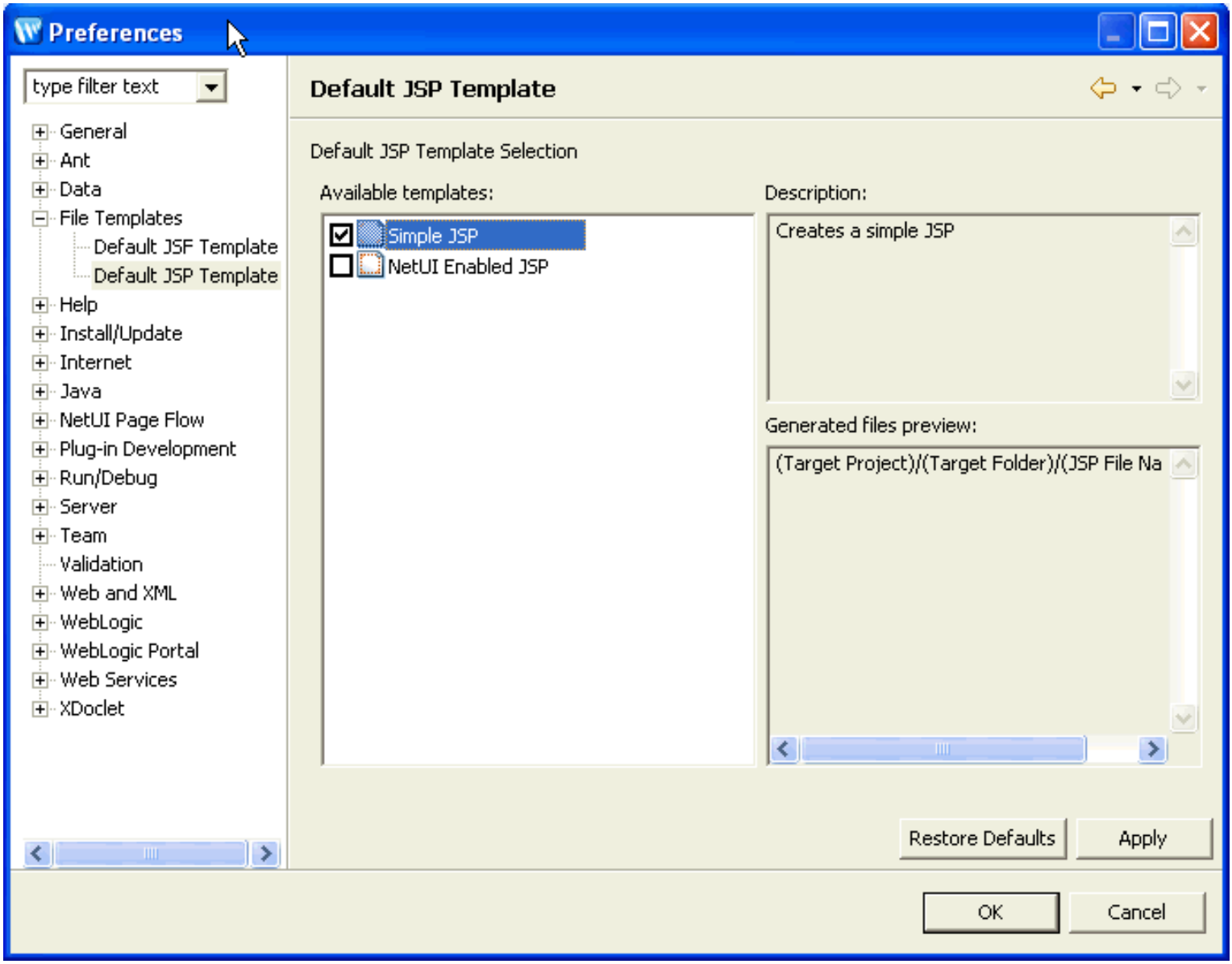
netui:tree Tag

Controlling Web Application Look and Feel with JSP/JSF Templates

BEA Workshop for WebLogic Platform allows you to set a default JSP template to use for all new JSP pages created with a given project or workspace.

Setting the Default Template for a Workspace

To set the default JSP template for all of the projects with a given workspace, select **Windows > Preferences > File Templates > Default JSP/JSF Template**:



The list of available templates is populated from any template projects in the workspace or any installed template plug-ins.

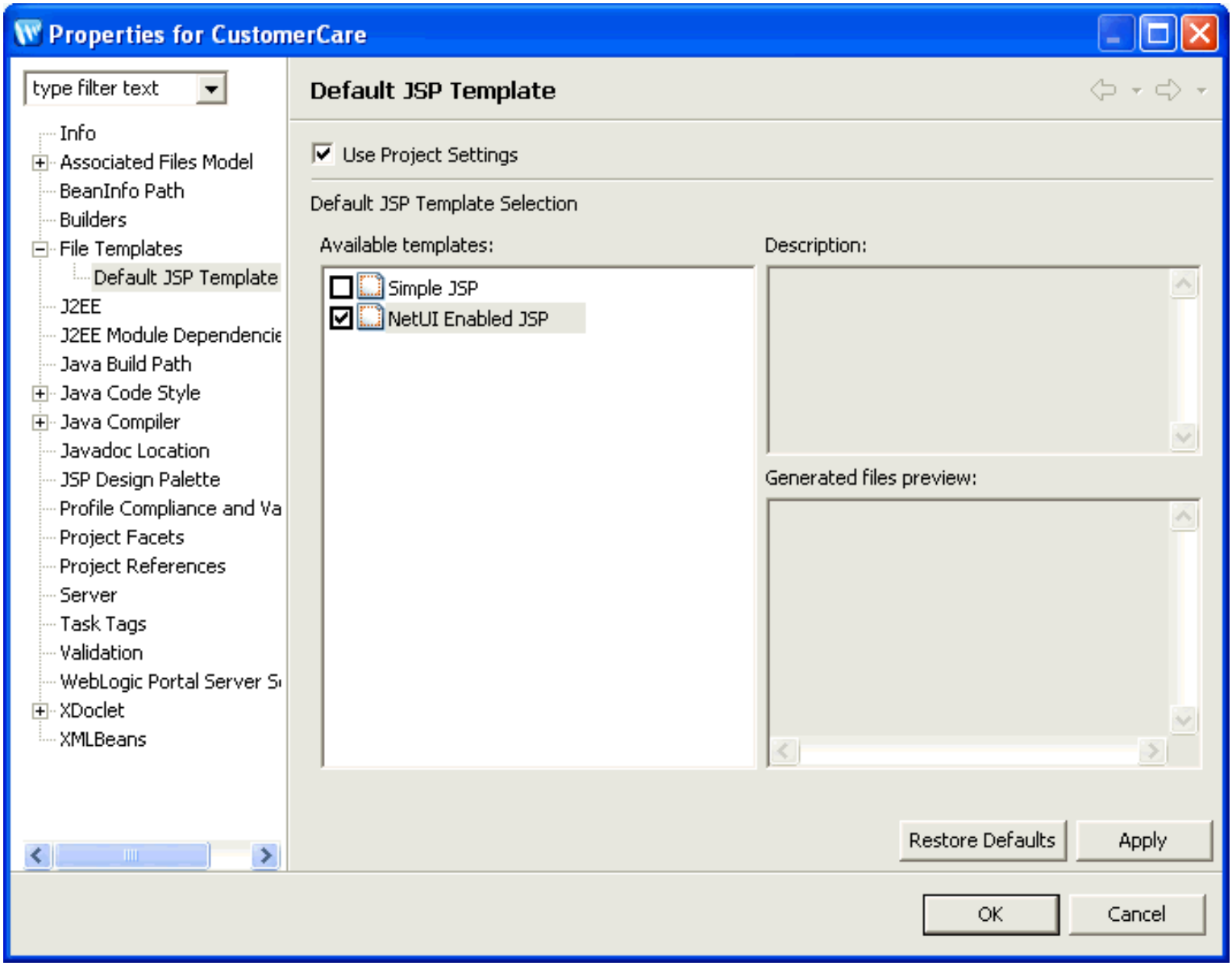
Clicking the checkbox next to a template designates it as the default template. But selecting a template label (not the checkbox) will show the template description and a preview of which files will be created. Note that the locations and names are shown in the abstract here since the actual location and file names are not known until the time of creation.

Setting the Default Template for a Project

To override the workspace default template setting for a given individual project, right-click the project folder (in the Workshop perspective) and select **Properties > File Templates > Default JSP Template**.

If the checkbox **Use Project Settings** is unchecked the workspace default settings are used. If checked then the project settings will override the workspace settings.

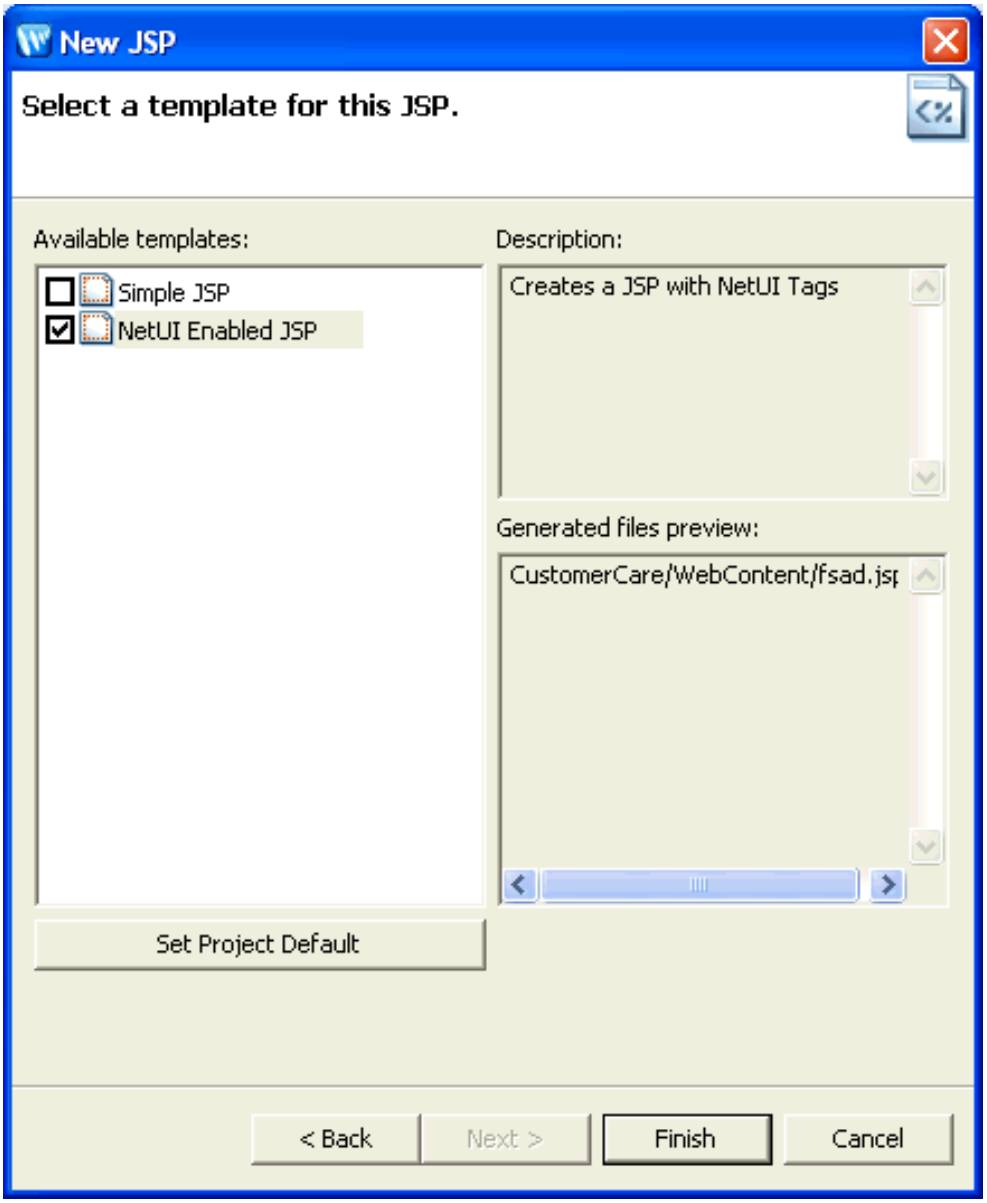
Note: In the Page Flow perspective, if you right-click the Pages node, and select **Set Default Page Template** the same project-level dialog will open.



Specifying a Template for a New JSP

In the Page Flow perspective, JSP template selection is part of the new JSP wizard.

To enter the new JSP wizard, select **File > New > JSP**.



Related Topics

Authoring JSP Template Projects

Authoring JSP Template Projects

This topic explains how to create a JSP template project. A JSP template project contains one or more JSP templates and adds those templates to the list of possible default JSP templates. For more information on setting the default JSP template see Controlling Web Application Look and Feel with JSP Templates.

Creating a JSP Template Project

Any project can be converted into a JSP template project, provided it has the appropriate the project nature. To define a project as a JSP template project, add the template project nature to the project's .project file:

```
    <natures>
        ...
        <nature>com.bea.wlw.filetemplate.core.templateProjectNature</nature>
        ...
    </natures>
```

The .project file resides in the root of a project directory. To view the .project file, switch to the Navigator view: **Window > Show View > Navigator**.

The template project nature will cause Workshop for WebLogic to recognize the project as a template project.

JSP Template Project Structure

A JSP template project consists of the following elements:

- the .project file is configured appropriately (see Creating a JSP Template Project above)
- a templateProject.xml file at the root of the template project directory
- any number of template resource files: JSP page templates, CSS files, image files, etc.

The set of files contained in a give template is defined by the templateProject.xml file. The following sample templateProject.xml file defines one template called "BEA Branded NetUI JSP". Multiple templates can be defined in a given templateProject.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<template-project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <template id="com.bea.demo.filetemplate.NetUIJSP"
            name="BEA Branded NetUI JSP"
            typeClass="com.bea.wlw.jsp.core.beans.JSPBaseBean">
    <description>A NetUI-enabled JSP with BEA Branding</description>
    <source-ref context="JSPBaseBean" source="com.bea.demo.filetemplate.NetUIJSP.source" />
    <source-ref context="FileTemplateBean" source="com.bea.demo.filetemplate.dataGrid.css.source" />
    <resource-ref resource="com.bea.demo.filetemplate.logo_bea_tl.gif.source" outputpath="WebContent/
resources/images/logo_bea_tl.gif" />
    <resource-ref resource="com.bea.demo.filetemplate.rt_blue_bkgnd.jpg.source" outputpath="WebContent/
resources/images/rt_blue_bkgnd.jpg" />
    <resource-ref resource="com.bea.demo.filetemplate.sp.gif.source" outputpath="WebContent/resources/
images/sp.gif" />
  </template>
  <source id="com.bea.demo.filetemplate.NetUIJSP.source" file="WebContent/index.jsp" type="jsp"></source>
  <source id="com.bea.demo.filetemplate.dataGrid.css.source" file="WebContent/resources/datagrid.css"
type="css"></source>
  <resource id="com.bea.demo.filetemplate.logo_bea_tl.gif.source" path="images/logo_bea_tl.gif" />
  <resource id="com.bea.demo.filetemplate.rt_blue_bkgnd.jpg.source" path="images/rt_blue_bkgnd.jpg" />
  <resource id="com.bea.demo.filetemplate.sp.gif.source" path="images/sp.gif" />
</template-project>
```

For information on creating a templateProject.xml file, see templateProject.xml Configuration File.

For an example JSP template project open the SamplesWorkspace. Instructions on opening the SamplesWorkspace are available at Opening a Sample Workspace.

Supported Character Encodings

Because of the way templates are processed, the files included in a template must to be encoded in UTF-8. Any other character encoding will result in an error.

JSP Template Plugins

A JSP template project can also be packaged as a plugin. Template plugins are nothing more than template projects that have the templateProject plugin point defined.

For an example of a template plugin see BEA_HOME/weblogic92/workshop/eclipse/plugins/com.bea.wlw.netui.core_9.2.0

Related Topics

Controlling Web Application Look and Feel with JSP Templates

templateProject.xml Configuration File

Tasks: Web Applications

This section contains instructions for common web application development tasks.

Topics included in this section:

How To Define an Action that Forwards Users to Another Page

This topic explains how to setup navigation between two JSP pages using an action method.

How to Submit User Data from a JSP

This topic explains how to setup user data submission from a JSP page to an action.

How to Change the Default Encoding for a New HTML Page

This topic explains how to change the default encoding for a new HTML page to UTF-8.

Related Topics

Tutorial: Accessing a Database from a Web Application

Tutorial: Java Server Faces Integration

How to Define an Action that Forwards Users to Another Page

This topic explains how to setup navigation between two JSP pages using a navigational action in a page flow Controller class.

These instructions assume that you have a dynamic web project (**New > Project > Other > Web > Dynamic Web Project**) that contains a page flow with at least two JSP pages.

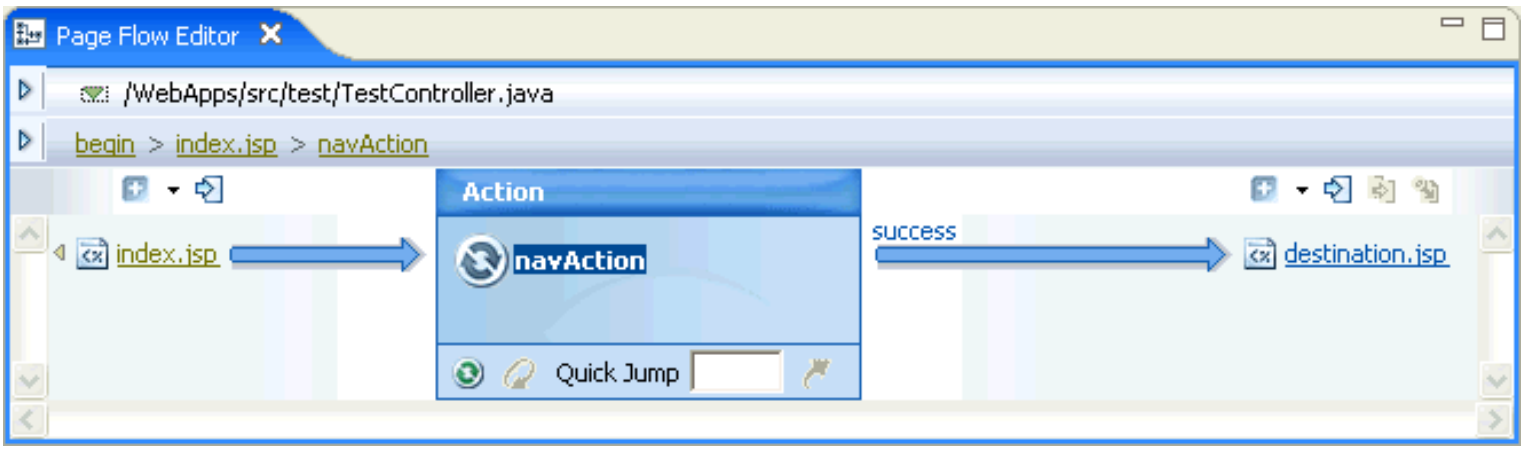
To Create the Navigational Action

1. Open the **Page Flow** perspective (**Window > Open Perspective > Page Flow**).
2. Right-click within the **Page Flow Editor** tab.
3. In the **New Action** dialog, in the **Action Template** field, confirm that `Basic Method Action` is selected.
   In the **Action Name** field, enter an appropriate name for the action. This will be the name of the action method.
   In the **Forward To** field, select the destination JSP page. (This is the page that users will navigate *to*.)
   Click **Finish**.
   In the **New Conditional Forward** dialog, do not enter any value and click **Cancel**. (This dialog lets you conditionalize the navigation action, if you wish.)

To Create a Link that Invokes the Navigational Action

1. Open the JSP page that will invoke the action. This is the starting page that users will navigate *from*.
2. On the **JSP Design Palette**, double-click the heading labeled **NetUI**. Under the **NetUI** heading, drag and drop the **anchor icon** onto the starting JSP page.
3. In the **New Anchor** dialog, in the **Anchor Type** dropdown, confirm that `Action` is selected.
   In the **Text** field, enter some appropriate text. This is the display text for the hyperlink.
   In the **Action** dropdown, select the action method you created above.
   Click **Ok**.

In the **Page Flow Editor** tab there should be two arrows: (1) an arrow pointing from the starting JSP page to the action method and (2) another arrow pointing from the action method to the destination JSP page.



The source code you have created should look something like the following:

index.jsp

```
<netui:anchor action="navAction">Navigate to destination.jsp!</netui:anchor>
```

Controller.java

```
@Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "destination.jsp") })
public Forward navAction() {
    Forward forward = new Forward("success");
    return forward;
}
```

Related Topics

Page Flow Editor

New Action

JSP Design Palette

How to Submit User Data from a JSP

This topic explains how to set up user data submission using an HTML form, a form bean (a Java representation of the HTML form), and an action.

The purpose of these instructions is to make you familiar with some of the main dialogs and wizards to help you accomplish this coding goal. These instructions are not intended to apply to every case where a form is required for user submitted data. For example, these instruction may not apply directly if you already have a pre-existing form beans. In that case, the instruction below can modified to utilize your pre-existing resources: where the instructions tell you to create a form bean, simply select the pre-existing item from the dropdown list.

These instructions assume that you have a dynamic web project (**New > Project > Other > Web > Dynamic Web Project**) that contains a page flow.

To Create a Form Bean to Model Submitted Data

(If you already have a form bean, you can skip this step.)

1. Open the **Page Flow** perspective (**Window > Open Perspective > Page Flow**).
2. On the **Page Flow Explorer** tab, right-click the **Form Bean** node and select **New Inner Class Form Bean**.
3. On the **Page Flow Explorer** tab, right-click the new form bean (named **NewFormBean** by default) and select **Rename**. Rename the form bean appropriately (e.g., Customer, Order, etc.).
4. On the **Page Flow Explorer** tab, double-click the form bean to view its source. Add private fields to the form bean class, for example:

```
@Jpf.FormBean
    public static class Customer implements java.io.Serializable {

    private String firstName;
    private String lastName;

}
```

5. Right-click within the body of the Controller class and select **Source > Generate Getters and Setters**. This will create public getter and setter methods for the form bean's private fields.

To Create an Action and a User Input Form Based on a Form Bean

1. Open the JSP page where you want the form to appear.
2. From the **JSP Design Palette** drag and drop the node **Create Form** onto the **JSP page**.
3. In the **Create Form** wizard, in the **Action** section, click **New**. (If you already have an action you want to use, do *not* click New. Instead select that action from the dropdown list and skip the next step.)
4. In the **New Action** wizard, in the **Action Template** field, select `Basic Method Action`.
   In the **Action Name** field, enter an appropriate name.
   In the **Form Bean** field, select the form bean created above.
   In the **Forward To** field, select an appropriate destination to forward the user to after data has been submitted.
   Click **Finish**.

5. In the **Create Form** wizard, click **Next**.
   On the **Select Properties** page, select the form bean fields that will appear in the user input form.
   Click **Next**.
   On the **Arrange Fields** page, select the order that the fields should appear on the JSP page.
   Click **Finish**.

The code created should look like something like the following:

form.jsp page

```
<netui:form action="nameAction">
    <table>
         <tr valign="top">
             <td><label for="field1"> FirstName: </label></td>
             <td><netui:textBox dataSource="actionForm.firstName" tagId="field1"></netui:textBox></td>
         </tr>
         <tr valign="top">
             <td><label for="field2"> LastName: </label></td>
             <td><netui:textBox dataSource="actionForm.lastName" tagId="field2"></netui:textBox></td>
         </tr>
    </table>
    <netui:button value="nameAction" type="submit" />
</netui:form>
```

Controller.java

```
    @Jpf.Action(forwards = { @Jpf.Forward(name = "success", path = "confirm.jsp") })
    public Forward nameAction(Controller.NameForm form) {
        Forward forward = new Forward("success");
        return forward;
    }

        ...

    @Jpf.FormBean
    public static class NameForm implements java.io.Serializable {
        private static final long serialVersionUID = 1815159769L;

        private String firstName;
        private String lastName;

        public String getFirstName() {
            return firstName;
        }
        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }
        public String getLastName() {
            return lastName;
        }
        public void setLastName(String lastName) {
            this.lastName = lastName;
        }
    }
```

Related Topics

Create Form Wizard

JSP Design Palette View

JSF Tutorial: Step 2: Create a JSF Web Application

How to Change the Default Encoding for a New HTML Page

Upon installation of Workshop for WebLogic, the default encoding for a new HTML page (**File > New > Other > Web > HTML**) is the same as the Java VM encoding. The Java VM encoding value will differ depending on the operating system configuration.

To change the default value to `charset=UTF-8` open the HTML Files dialog (**Window > Preferences > Web and XML > HTML Files**). In the section labeled **Creating files**, in the **Encoding** dropdown, select the value `ISO 10646/Unicode(UTF-8)`.

The default HTML encoding is a workspace level setting. This means that each new workspace will be initiated with a default encoding of `ISO-8859-1`. If another default encoding is desired, it must be reset upon the creation of each new workspace.

> **Note:** the preferences dialog **Window > Preferences > Web and XML > JSP Files > Encoding** has no effect on the default encoding for JSP files. To change the default encoding for new JSP pages, create a JSP template project and reset the default JSP template.

Related Topics

Controlling Web Application Look and Feel with JSP/JSF Templates

Authoring JSP Template Projects