# Introduction to JavaServer Faces 2.x

JavaServer Faces (JSF) is a user interface (UI) framework for Java web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client. JSF provides ease-of-use in the following ways:

- Makes it easy to construct a UI from a set of reusable UI components
- Simplifies migration of application data to and from the UI
- Helps manage UI state across server requests
- Provides a simple model for wiring client-generated events to server-side application code
- Allows custom UI components to be easily built and re-used

For an in-depth description of the JSF framework, see the Java EE 7 Tutorial, Chapter 12 Developing with JavaServer Faces Technology.

This tutorial demonstrates how you can apply JSF 2.x support to a web application using the NetBeans IDE. You begin by adding JSF 2.x framework support to a basic web application, and then proceed to perform the following tasks:

- create a JSF managed bean to handle request data,
- wire the managed bean to the application's web pages, and
- convert the web pages into Facelets template files.

The NetBeans IDE has provided long-standing support for JavaServer Faces. Starting with the release of JSF 2.0 and Java EE 6, NetBeans IDE has provided support for JSF 2.0 and JSF 2.1. For more information, see JSF 2.x Support in NetBeans IDE.

## Contents

To complete this tutorial, you need the following software and resources.

| Software or Resource | Version Required |
|---|---|

| | |
|---|---|
| NetBeans IDE | 7.2, 7.3, 7.4, 8.0, Java EE bundle |
| Java Development Kit (JDK) | 7 or 8 |
| GlassFish server | Open Source Edition 3.x or 4 |
| `jsfDemo` web application project | n/a |

**Notes:**

- The NetBeans IDE Java Bundle also includes the GlassFish server, a Java EE-compliant server, which you require for this tutorial.
- To compare your project with a working solution, download the completed sample project.

## Adding JSF 2.x Support to a Web Application

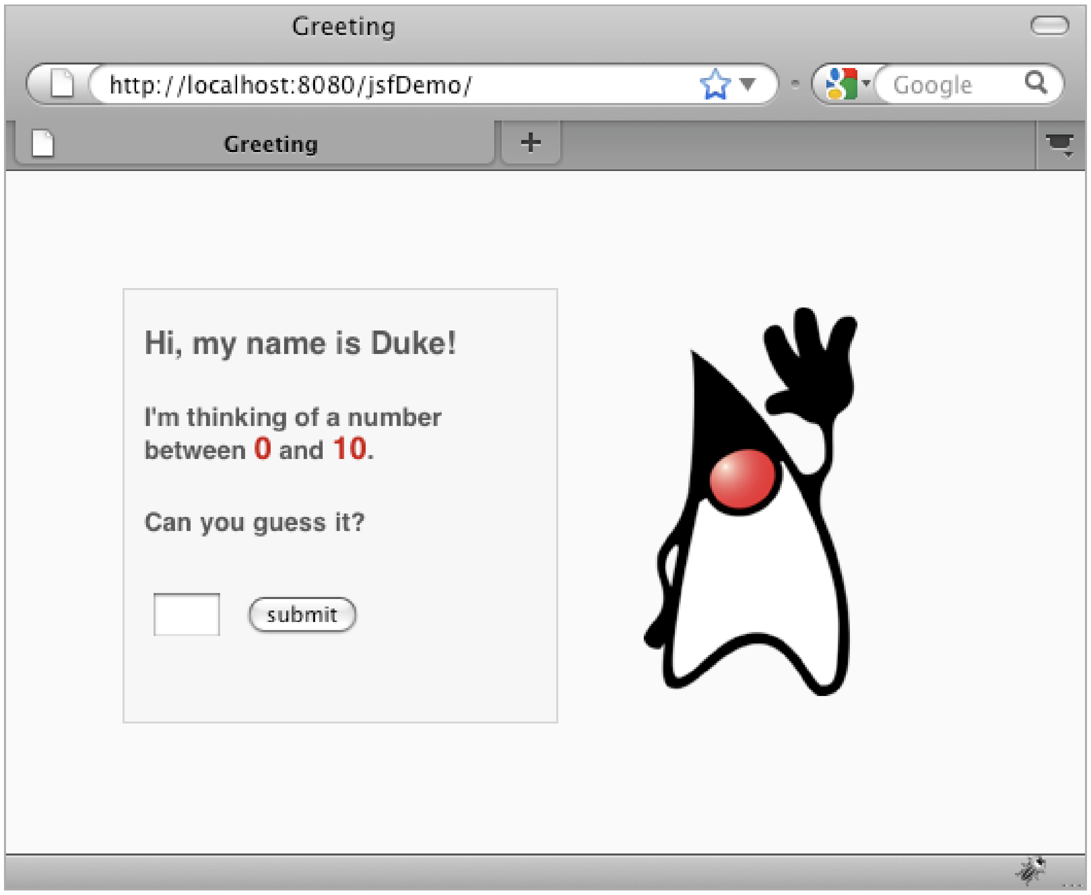
Begin by opening the `jsfDemo` web application project in the IDE. Once you have the project opened in the IDE, you can add framework support to it using the project's Properties window.

The IDE also allows you to create new projects with JSF 2.x support. For more information, see Creating a New Project with JSF 2.x Support.

1. Click the Open Project ( ) button in the IDE's main toolbar, or press Ctrl-Shift-O (⌘-Shift-O on Mac).
2. In the Open Project dialog, navigate to the location on your computer where you stored the unzipped tutorial project. Select it, then click Open Project to open it in the IDE.

   **Note.** You might be prompted to resolve the reference to the JUnit libraries when you open the NetBeans project if you did not install the JUnit plugin when you installed the IDE.

3. Run the project to see what it looks like in a browser. Either right-click the `jsfDemo` project node in the Projects window and choose Run, or click the Run Project ( ) button in the main toolbar. The project is packaged and deployed to the GlassFish server, and your browser opens to display the welcome page (`index.xhtml`).

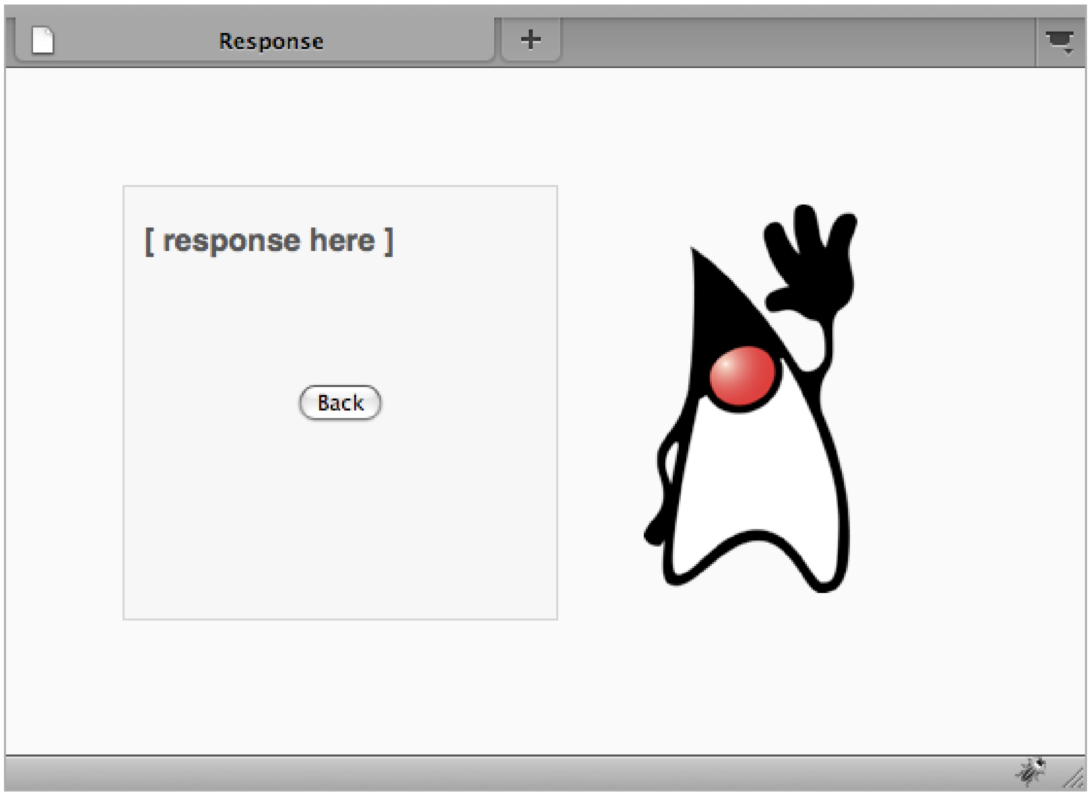


Hi, my name is Duke!

I'm thinking of a number between 0 and 10.

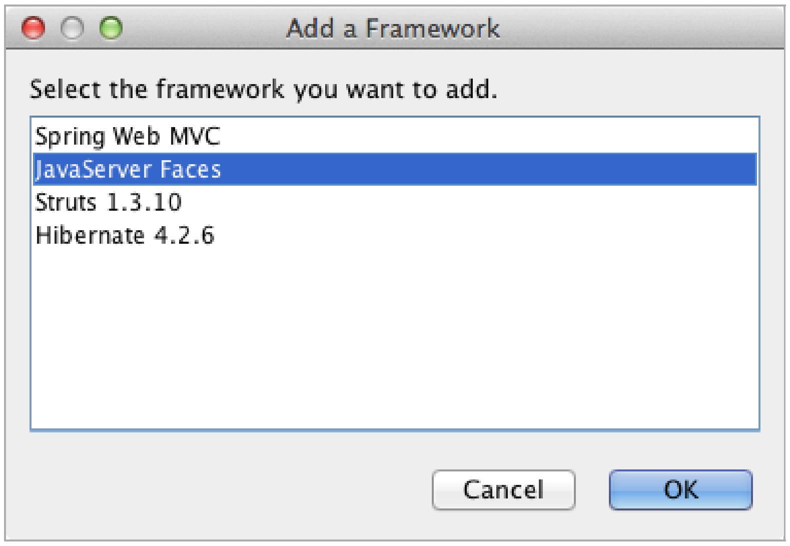
Can you guess it?

4. Click the Submit button. The response page (`response.xhtml`) displays as follows:

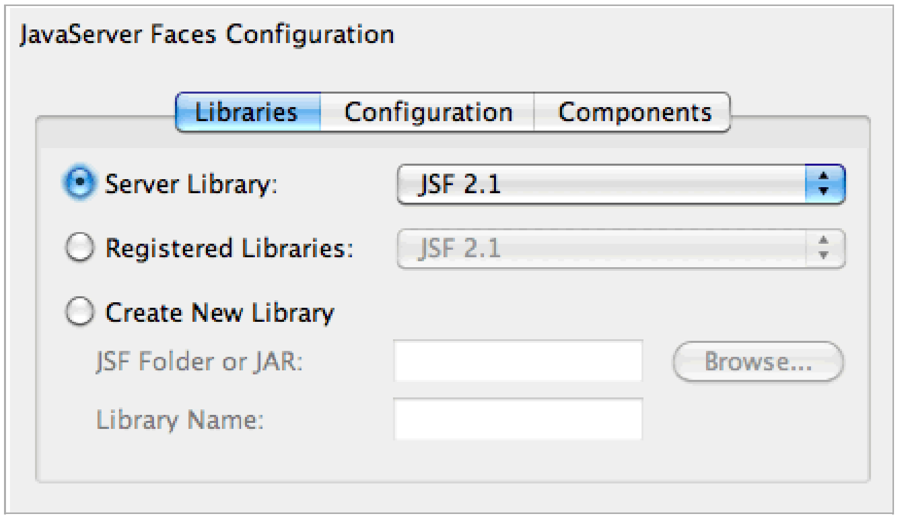


Currently the welcome and response pages are static and, together with the `stylesheet.css` file and `duke.png` image, are the only application files accessible from a browser.

5. In the Projects window (Ctrl-1; ⌘-1 on Mac), right-click your project node and choose Properties to open the Project Properties window.
6. Select the Frameworks category and then click the Add button.
7. Select JavaServer Faces in the Add a Framework dialog box. Click OK.



After selecting JavaServer Faces, various configuration options become available. Under the Libraries tab, you can specify how the project accesses JSF 2.x libraries. The JSF version that is available will depend upon the version of the IDE and the GlassFish server. The default option is to use the libraries included with the server (the GlassFish server). However, the IDE also bundles the JSF 2.x libraries. (You can select the Registered Libraries option if you want your project to use these.)


JavaServer Faces Configuration

Libraries | Configuration | Components

[✓ Server Library:] JSF 2.1

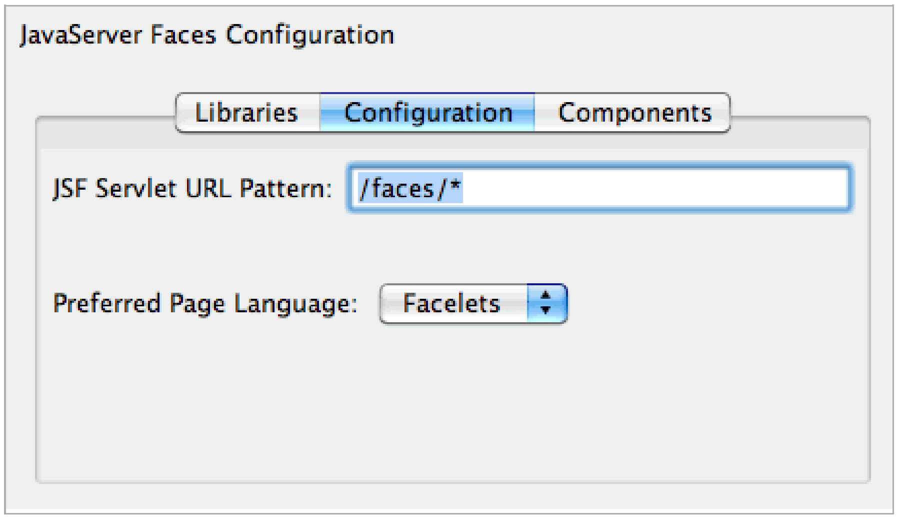
Registered Libraries: JSF 2.1

Create New Library

JSF Folder or JAR:

Library Name:

8. Click the Configuration tab. You can specify how the Faces servlet is registered in the project's deployment descriptor. You can also indicate whether you want Facelets or JSP pages to be the used with the project.

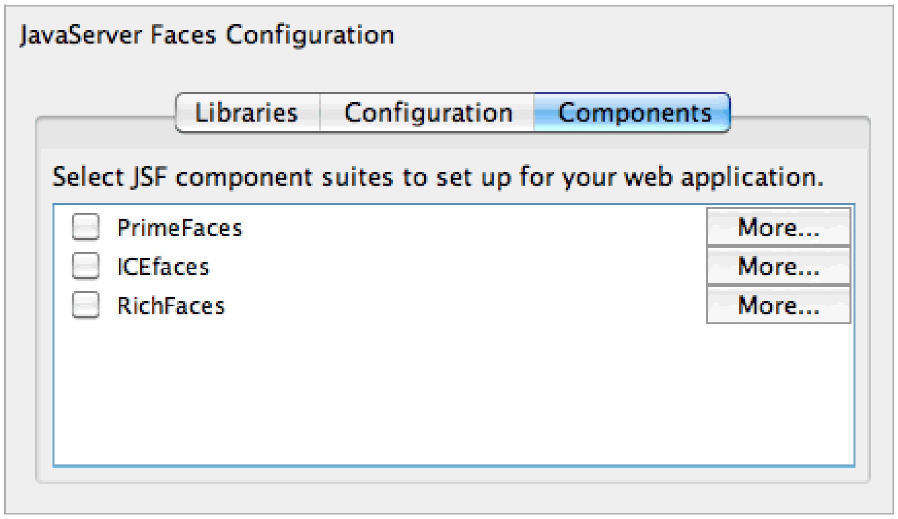

JavaServer Faces Configuration

Libraries | Configuration | Components

JSF Servlet URL Pattern: /faces/*

Preferred Page Language: Facelets

You can also easily configure your project to use various JSF component suites in the Components tab. To use a component suite you will need to download the required libraries and use the Ant Library manager to create a new library with the component suite libraries.


JavaServer Faces Configuration

Select JSF component suites to set up for your web application.

- [ ] PrimeFaces
- [ ] ICEfaces
- [ ] RichFaces

9. Click OK to finalize changes and exit the Project Properties window.

After adding JSF support to your project, the project's `web.xml` deployment descriptor is modified to look as follows. (Changes in **bold**.)

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

**Important:** Confirm that the `web.xml` contains only one `<welcome-file>` entry and that the entry contains '`faces/`' as shown in the example. This ensures that the project's welcome page (`index.xhtml`) passes through the Faces servlet before being displayed in a browser. This is necessary in order to render the Facelets tag library components properly.

The Faces servlet is registered with the project, and the `index.xhtml` welcome page is now passed through the Faces servlet when it is requested. Also, note that an entry for the `PROJECT_STAGE` context parameter has been added. Setting this parameter to '`Development`' provides you with useful information when debugging your application. See http://blogs.oracle.com/rlubke/entry/jsf_2_0_new_feature2 for more information.

You can locate the JSF libraries by expanding the project's Libraries node in the Projects window. If you are using the default libraries included with GlassFish Server 3.1.2 or GlassFish Server 4 this is the `javax.faces.jar` that is visible under the GlassFish Server node. (If you are using an older version of GlassFish you will see the `jsf-api.jar` and `jsf-impl.jar` libraries instead of `javax.faces.jar`.)

The IDE's JSF 2.x support primarily includes numerous JSF-specific wizards, and special functionality provided by the Facelets editor. You explore these functional capabilities in the following steps. For more information, see JSF 2.x Support in NetBeans IDE.

## Creating a Managed Bean

You can use JSF's managed beans to process user data and retain it between requests. A managed bean is a POJO (Plain Old Java Object) that can be used to store data, and is managed by the container (e.g., the GlassFish server) using the JSF framework.

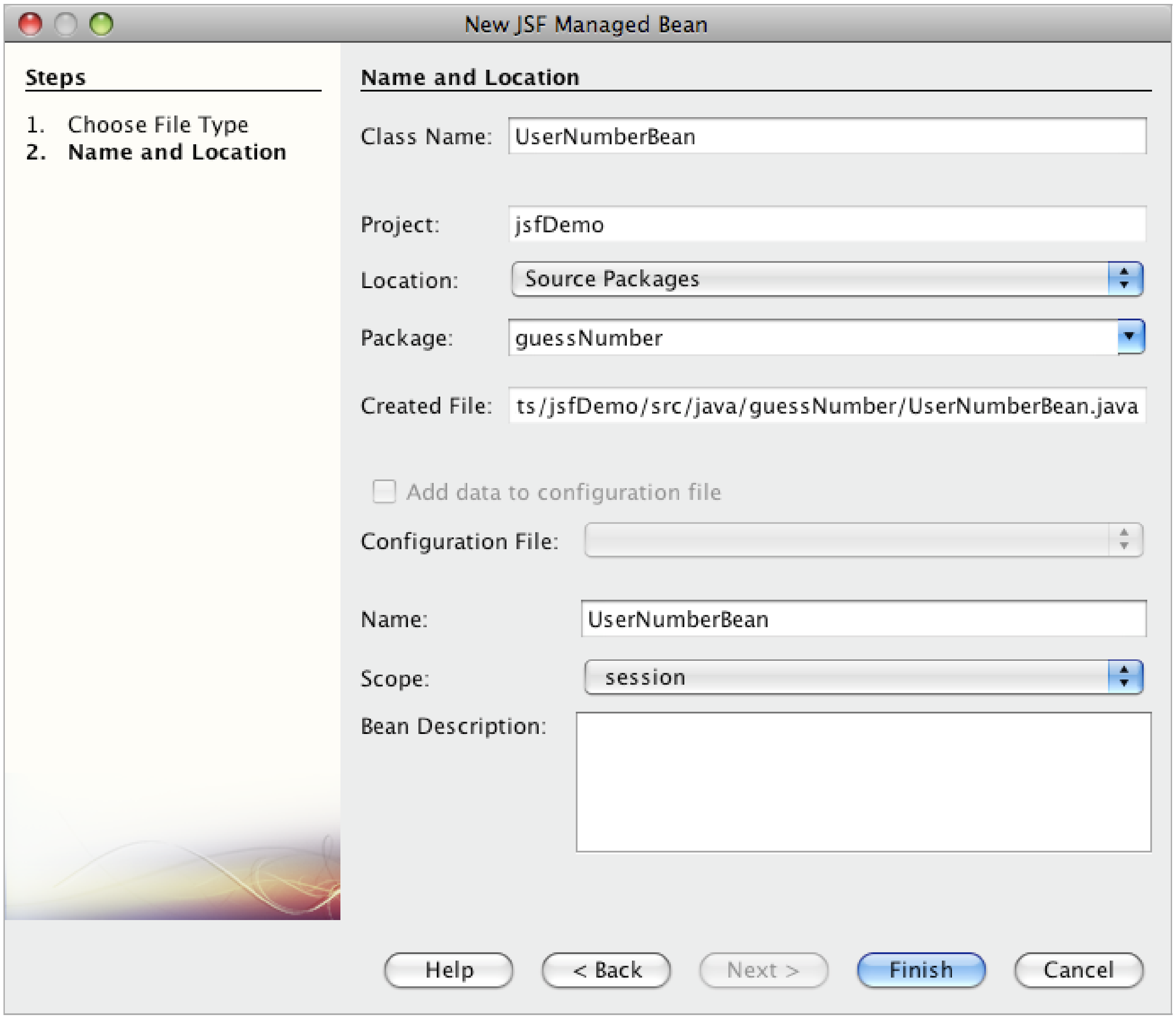
> A POJO is essentially a Java class that contains a public, no argument constructor and conforms to the JavaBeans naming conventions for its properties.

Looking at the static page produced from running the project, you need a mechanism that determines whether a user-entered number matches the one currently selected, and returns a view that is appropriate for this outcome. Use the IDE's Managed Bean wizard to create a managed bean for this purpose. The Facelets pages that you create in the next section will need to access the number that the user types in, and the generated response. To enable this, add `userNumber` and `response` properties to the managed bean.

- Using the Managed Bean Wizard
- Creating a Constructor
- Adding Properties

### Using the Managed Bean Wizard

1. In the Projects window, right-click the `jsfDemo` project node and choose New > JSF Managed Bean. (If Managed Bean is not listed, choose Other. Then select the JSF Managed Bean option from the JavaServer Faces category. Click Next.)
2. In the wizard, enter the following:
    - **Class Name:** UserNumberBean
    - **Package:** guessNumber
    - **Name:** UserNumberBean
    - **Scope:** Session


New JSF Managed Bean

Steps

1. Choose File Type
2. **Name and Location**

**Name and Location**

Class Name: UserNumberBean

Project: jsfDemo

Location: Source Packages

Package: guessNumber

Created File: ts/jsfDemo/src/java/guessNumber/UserNumberBean.java

☐ Add data to configuration file

Configuration File:

Name: UserNumberBean

Scope: session

Bean Description:

3. Click Finish. The `UserNumberBean` class is generated and opens in the editor. Note the following annotations (shown in **bold**):

```
package guessNumber;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

/**
 *
 * @author nbuser
 */
@ManagedBean(name="UserNumberBean")
@SessionScoped
public class UserNumberBean {
    /** Creates a new instance of UserNumberBean */
    public UserNumberBean() {
    }

}
```

Because you are using JSF 2.x, you can declare all JSF-specific components using annotations. In previous versions, you would need to declare them in the Faces configuration file (`faces-config.xml`).

To view the Javadoc for all JSF 2.1 annotations, see the Faces Managed Bean Annotation Specification.

## Creating a Constructor

The `UserNumberBean` constructor must generate a random number between 0 and 10 and store it in an instance variable. This partially forms the business logic for the application.

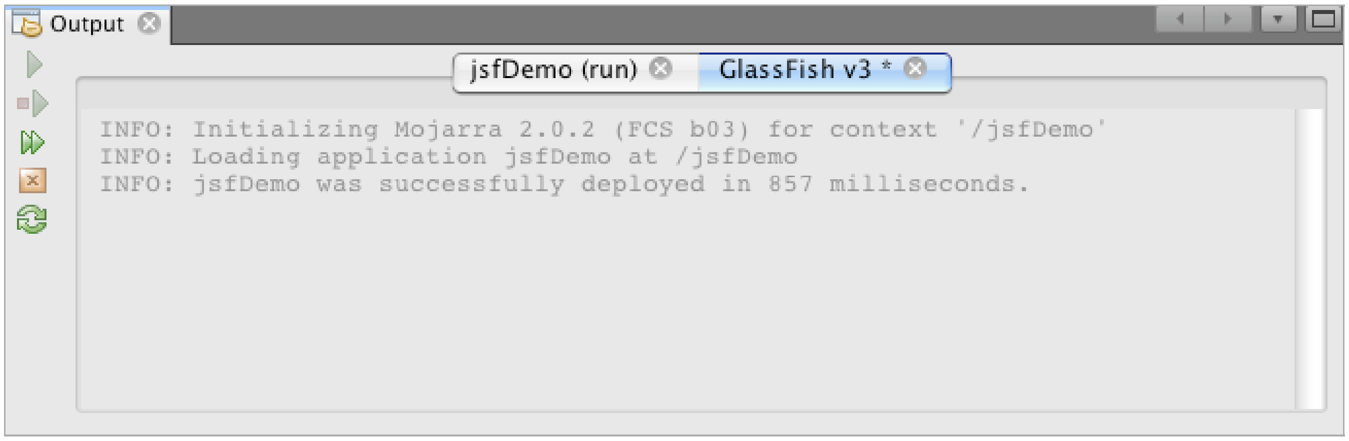
1. Define a constructor for the `UserNumberBean` class. Enter the following code (changes displayed in **bold**).

```
public class UserNumberBean {

    Integer randomInt;

    /** Creates a new instance of UserNumberBean */
    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Duke's number: " + randomInt);
    }

}
```

   The above code generates a random number between 0 and 10, and outputs the number in the server log.

2. Fix imports. To do so, click the hint badge ( ) that displays in the editor's left margin, then choose the option to import `java.util.Random` into the class.

3. Run the project again (click the Run Project ( ) button, or press F6; fn-F6 on Mac). When you run your project, the server's log file automatically opens in the Output window.



```
INFO: Initializing Mojarra 2.0.2 (FCS b03) for context '/jsfDemo'
INFO: Loading application jsfDemo at /jsfDemo
INFO: jsfDemo was successfully deployed in 857 milliseconds.
```

Notice that you do not see "`Duke's number: `" listed in the output (as would be indicated from the constructor). A `UserNumberBean` object was not created because JSF uses *lazy instantiation* by default. That is, beans in particular scopes are only created and initialized when they are needed by the application.

The Javadoc for the `@ManagedBean` annotation states:

> *If the value of the `eager()` attribute is `true`, and the `managed-bean-scope` value is "application", the runtime must instantiate this class when the application starts. This instantiation and storing of the instance must happen before any requests are serviced. If eager is unspecified or `false`, or the `managed-bean-scope` is something other than "application", the default "lazy" instantiation and scoped storage of the managed bean happens.*

4. Because `UserNumberBean` is session-scoped, have it implement the `Serializable` interface.

```
@ManagedBean(name="UserNumberBean")
@SessionScoped
public class UserNumberBean implements Serializable {
```

Use the hint badge ( ) to import `java.io.Serializable` into the class.

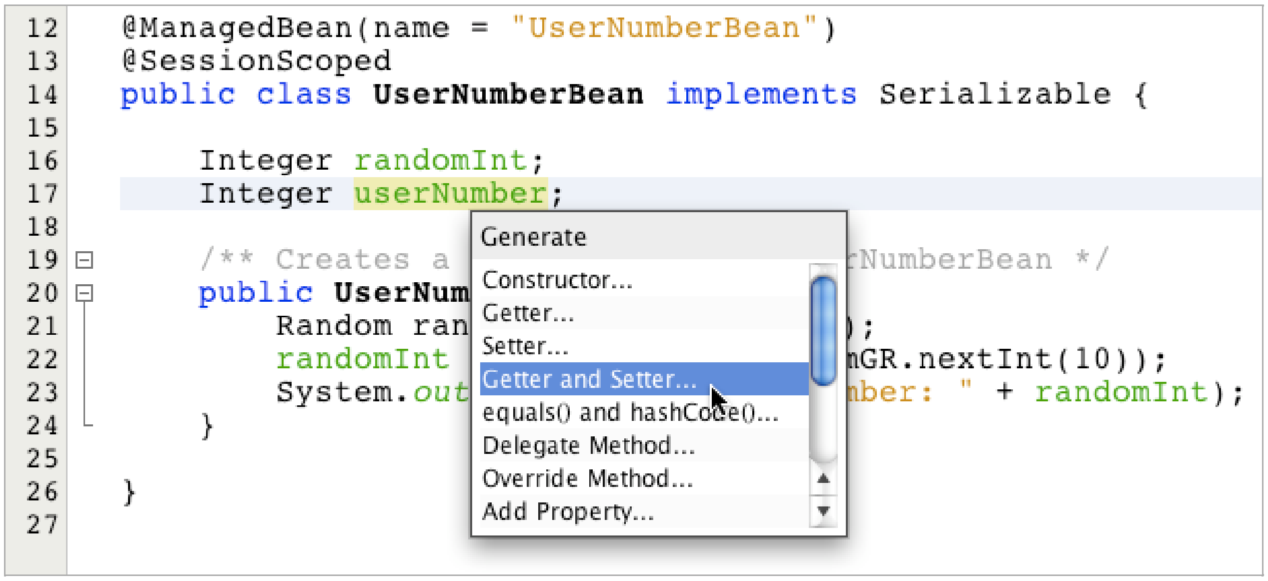
## Adding Properties

The Facelets pages that you create in the next section will need to access the number that the user types in, and the generated response. To facilitate this, add `userNumber` and `response` properties to the class.

1. Start by declaring an `Integer` named `userNumber`.

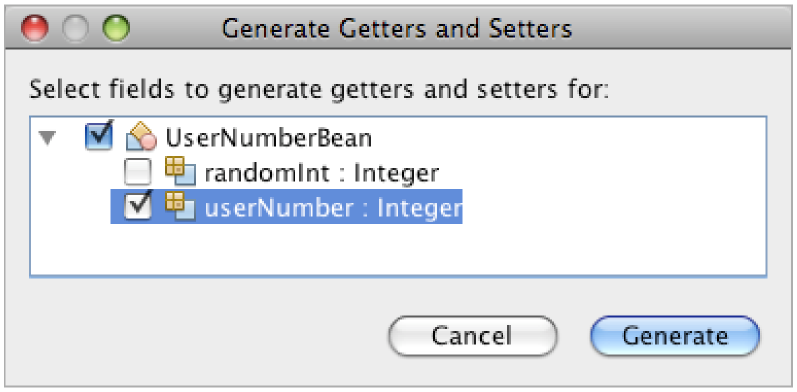
```
@ManagedBean(name="UserNumberBean")
@SessionScoped
public class UserNumberBean implements Serializable {

    Integer randomInt;
    Integer userNumber;
```

2. Right-click in the editor and choose Insert Code (Alt-Insert; Ctrl-I on Mac). Choose Getter and Setter.



```
@ManagedBean(name = "UserNumberBean")
@SessionScoped
public class UserNumberBean implements Serializable {

    Integer randomInt;
    Integer userNumber;

    /** Creates a                                 rNumberBean */
    public UserNum
        Random ran                                );
        randomInt                                 mGR.nextInt(10));
        System.out                                mber: " + randomInt);
    }

}
```

3. Select the `userNumber` : `Integer` option. Click Generate.



Select fields to generate getters and setters for:

- [✓] UserNumberBean
  - [ ] randomInt : Integer
  - [✓] userNumber : Integer

Note that the `getUserNumber()` and `setUserNumber(Integer userNumber)` methods are added to the class.

4. Create a `response` property. Declare a `String` named `response`.

```
@ManagedBean(name="UserNumberBean")
@SessionScoped
public class UserNumberBean implements Serializable {

    Integer randomInt;
    Integer userNumber;
    String response;
```

5. Create a getter method for `response`. (This application will not require a setter.) You could use the IDE's Generate Code pop-up shown in step 2 above to generate template code. For purposes of this tutorial however, just paste the below method into the class.

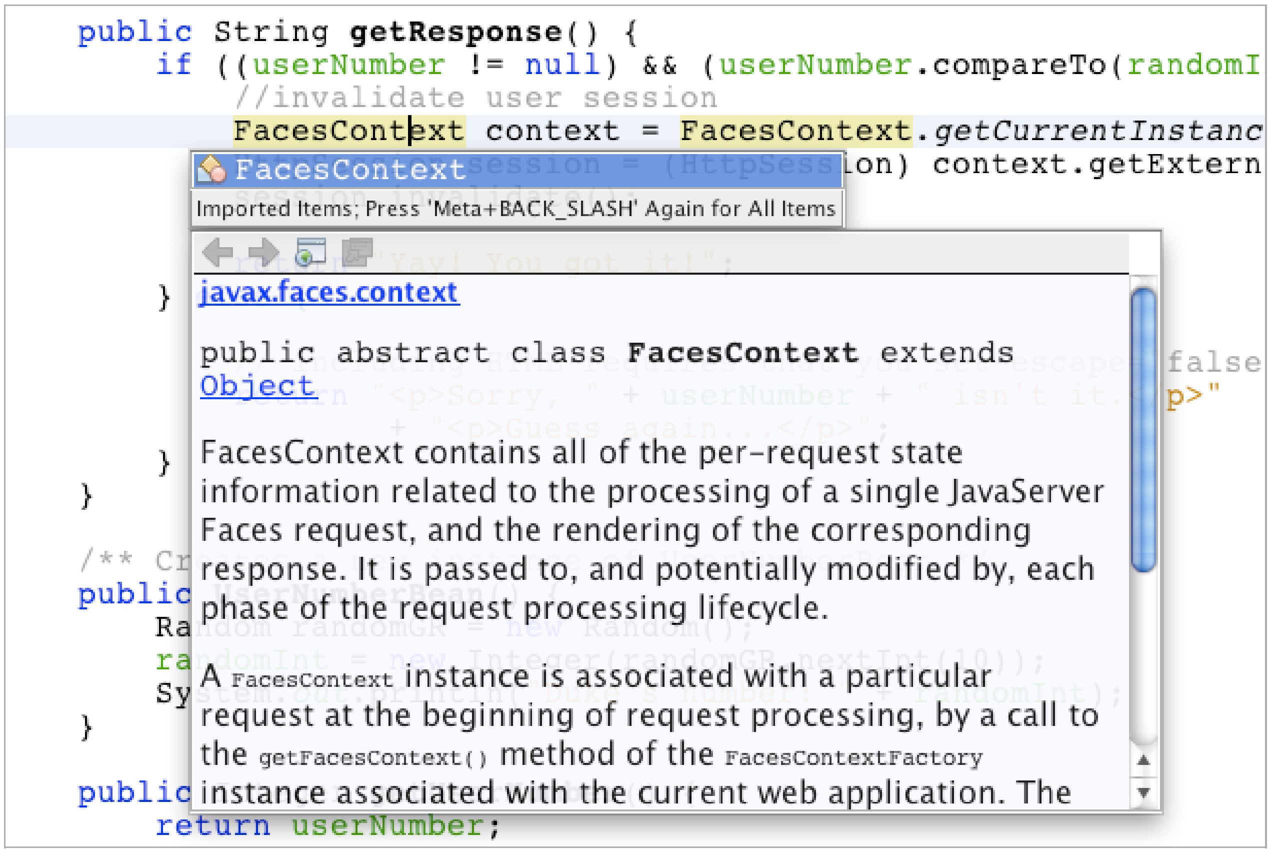
```
public String getResponse() {
    if ((userNumber != null) && (userNumber.compareTo(randomInt) == 0)) {

        //invalidate user session
        FacesContext context = FacesContext.getCurrentInstance();
        HttpSession session = (HttpSession)
context.getExternalContext().getSession(false);
        session.invalidate();

        return "Yay! You got it!";
    } else {

        return "<p>Sorry, " + userNumber + " isn't it.</p>"
                + "<p>Guess again...</p>";
    }
}
```

The above method performs two functions:

1. It tests whether the user-entered number (`userNumber`) equals the random number generated for the session (`randomInt`) and returns a `String` response accordingly.
2. It invalidates the user session if the user guesses the right number (i.e., if `userNumber` equals `randomInt`). This is necessary so that a new number is generated should the user want to play again.

6. Right-click in the editor and choose Fix Imports (Alt-Shift-I; ⌘-Shift-I on Mac). Import statements are automatically created for:
   - `javax.servlet.http.HttpSession`
   - `javax.faces.context.FacesContext`

   You can press Ctrl-Space on items in the editor to invoke code-completion suggestions and documentation support. Press Ctrl-Space on `FacesContext` to view the class description from the Javadoc.



```
    public String getResponse() {
        if ((userNumber != null) && (userNumber.compareTo(randomI
            //invalidate user session
            FacesContext context = FacesContext.getCurrentInstanc
            FacesContext                          ion) context.getExtern
```

javax.faces.context

```
public abstract class FacesContext extends
Object
```

FacesContext contains all of the per-request state information related to the processing of a single JavaServer Faces request, and the rendering of the corresponding response. It is passed to, and potentially modified by, each phase of the request processing lifecycle.

A `FacesContext` instance is associated with a particular request at the beginning of request processing, by a call to the `getFacesContext()` method of the `FacesContextFactory` instance associated with the current web application. The

```
    public
        return userNumber;
```

Click the web browser ( ) icon in the documentation window to open the Javadoc in an external web browser.

## Wiring Managed Beans to Pages

One of the primary purposes of JSF is to remove the need to write boilerplate code to manage POJOs and their interaction with the application's views. You saw an example of this in the previous section, where JSF instantiated a `UserNumberBean` object when you ran the application. This notion is referred to as Inversion of Control (IoC), which enables the container to take responsibility for managing portions of the application that would otherwise require the developer to write repetitious code.

In the previous section you created a managed bean that generates a random number between 0 and 10. You also created two properties, `userNumber`, and `response`, which represent the number input by the user, and the response to a user guess, respectively.

In this section, you explore how you can use the `UserNumberBean` and its properties in web pages. JSF enables you to do this using its expression language (EL). You use the expression language to bind property values to JSF's UI components contained in your application's web pages. This section also demonstrates how you can take advantage of JSF 2.x's implicit navigation feature to navigate between the index and response pages.

The IDE provides support for this work through its code completion and documentation facilities, which you can invoke by pressing Ctrl-Space on items in the editor.

Start by making changes to `index.xhtml`, then make changes to `response.xhtml`. In both pages, replace HTML form elements with their JSF counterparts, as they are defined in the JSF HTML tag library. Then, use the JSF expression language to bind property values with selected UI components.

- index.xhtml
- response.xhtml

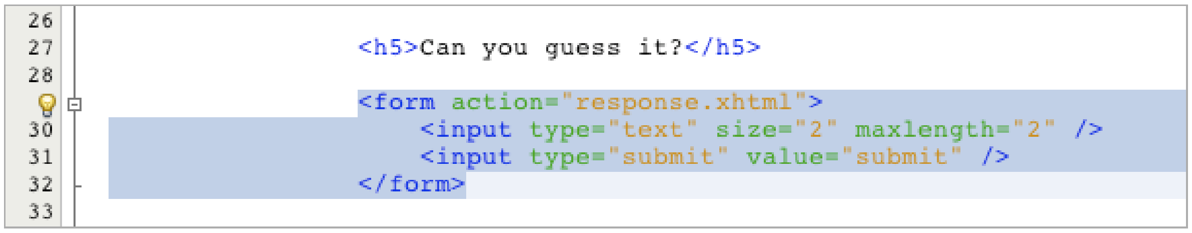
## index.xhtml

1. Open the `index.xhtml` page in the editor. Either double-click the `index.xhtml` node from the Projects window, or press Alt-Shift-O to use the Go to File dialog.

   Both index and response pages already contain the JSF UI components you require for this exercise. Simply uncomment them and comment out the HTML elements currently being used.

2. Comment out the HTML form element. To do so, highlight the HTML form element as in the image below, then press Ctrl-/ (⌘-/ on Mac).

   **Note:** To highlight, either click and drag in the editor with your mouse, or, using the keyboard, hold Shift and press the arrow keys.

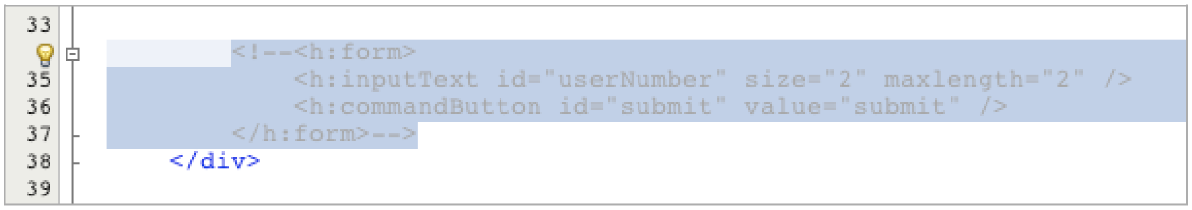
   

```
                <h5>Can you guess it?</h5>

                <form action="response.xhtml">
                    <input type="text" size="2" maxlength="2" />
                    <input type="submit" value="submit" />
                </form>
```

   Use Ctrl-/ (⌘-/ on Mac) to toggle comments in the editor. You can also apply this keyboard shortcut to other file types, such as Java and CSS.

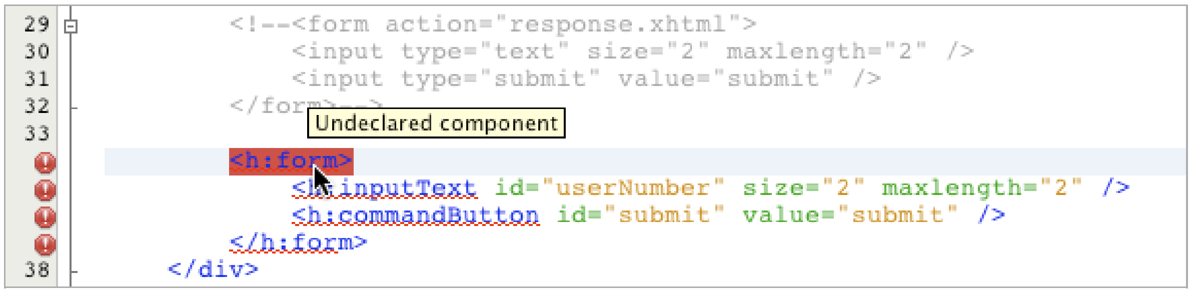
3. Uncomment the JSF HTML form component. Highlight the component as in the image below, then press Ctrl-/ (⌘-/ on Mac).

   **Note.** You might need to press Ctrl-/ twice to uncomment the code.

```
                <!--<h:form>
                    <h:inputText id="userNumber" size="2" maxlength="2" />
                    <h:commandButton id="submit" value="submit" />
                </h:form>-->
            </div>
```

   After uncommenting the JSF HTML form component, the editor indicates that the `<h:form>`, `<h:inputText>`, and `<h:commandButton>` tags haven't been declared.



```
        <!--<form action="response.xhtml">
            <input type="text" size="2" maxlength="2" />
            <input type="submit" value="submit" />
        </form>-->
        <h:form>
            <h:inputText id="userNumber" size="2" maxlength="2" />
            <h:commandButton id="submit" value="submit" />
        </h:form>
    </div>
```

4. To declare these components, use the IDE's code completion to add the tag library namespace to the page's `<html>` tag. Place your cursor on any of the undeclared tags and press Alt-Enter and click Enter to add the suggested tag library. (If there are multiple options, make sure to select the tag that is displayed in the editor before clicking Enter.) The JSF HTML tag library namespace is added to the `<html>` tag (shown in **bold** below), and the error indicators disappear.

   **Note.** If the IDE does not provide the option to add the tag library you will need to manually modify the `<html>` element.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

5. Use the JSF expression language to bind `UserNumberBean`'s `userNumber` property to the `inputText` component. The `value` attribute can be used to specify the current value of the rendered component. Type in the code displayed in **bold** below.

```
<h:form>
    <h:inputText id="userNumber" size="2" maxlength="2" value="#
{UserNumberBean.userNumber}" />
```

   JSF expression language uses the `#{}` syntax. Within these delimiters, you specify the name of the managed bean and the bean property you want to apply, separated by a dot (`.`). Now, when the form data is sent to the server, the value is automatically saved in the `userNumber` property using the property's setter (`setUserNumber()`). Also, when the page is requested and a value for `userNumber` has already been set, the value will automatically display in the rendered `inputText` component. For more information, see the Java EE 7 Tutorial: 12.1.2 Using the EL to Reference Managed Beans.

6. Specify the destination for the request that is invoked when clicking the form button. In the HTML version of the form, you were able to do this using the `<form>` tag's `action` attribute. With JSF, you can use the `commandButton`'s `action` attribute. Furthermore, due to JSF 2.x's implicit navigation feature, you only need to specify the name of the destination file, without the file extension.
   Type in the code displayed in **bold** below.

```
<h:form>
    <h:inputText id="userNumber" size="2" maxlength="2" value="#
{UserNumberBean.userNumber}" />
    <h:commandButton id="submit" value="submit" action="response" />
</h:form>
```

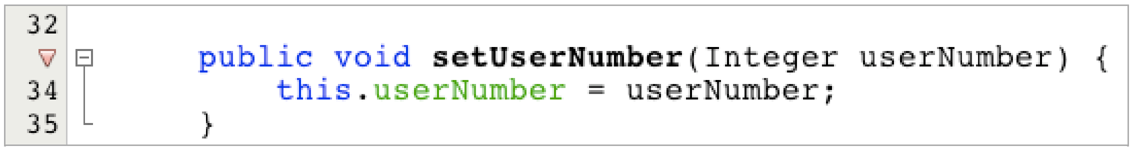
The JSF runtime searches for a file named `response`. It assumes the file extension is the same as the extension used by file from which the request originated (`index`**`.xhtml`**) and looks for for the `response.xhtml` file in the same directory as the originating file (i.e., the webroot).

> **Note:** JSF 2.x aims to make developers' tasks much easier. If you were using JSF 1.2 for this project, you would need to declare a navigation rule in a Faces configuration file that would look similar to the following:

```
<navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>

    <navigation-case>
        <from-outcome>response</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Steps 7 through 12 below are optional. If you'd like to quickly build the project, skip ahead to `response.xhtml`.

7. Test whether the above EL expression does in fact call the `setUserNumber()` method when the request is processed. To do so, use the IDE's Java debugger.
   Switch to the `UserNumberBean` class (Press Ctrl-Tab and choose the file from the list.) Set a breakpoint on the `setUserNumber()` method signature. You can do this by clicking in the left margin. A red badge displays, indicating a method breakpoint has been set.



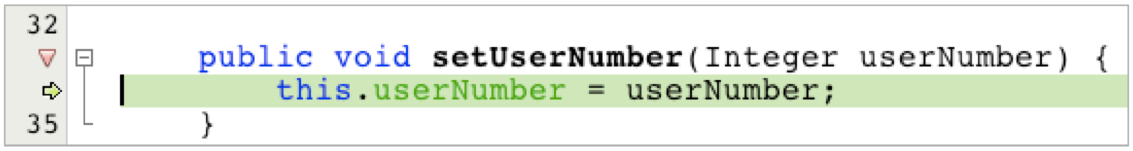
```
    public void setUserNumber(Integer userNumber) {
        this.userNumber = userNumber;
    }
```

8. Click the Debug Project ( ) button in the IDE's main toolbar. A debug session starts, and the project welcome page opens in the browser.

   **Notes.**

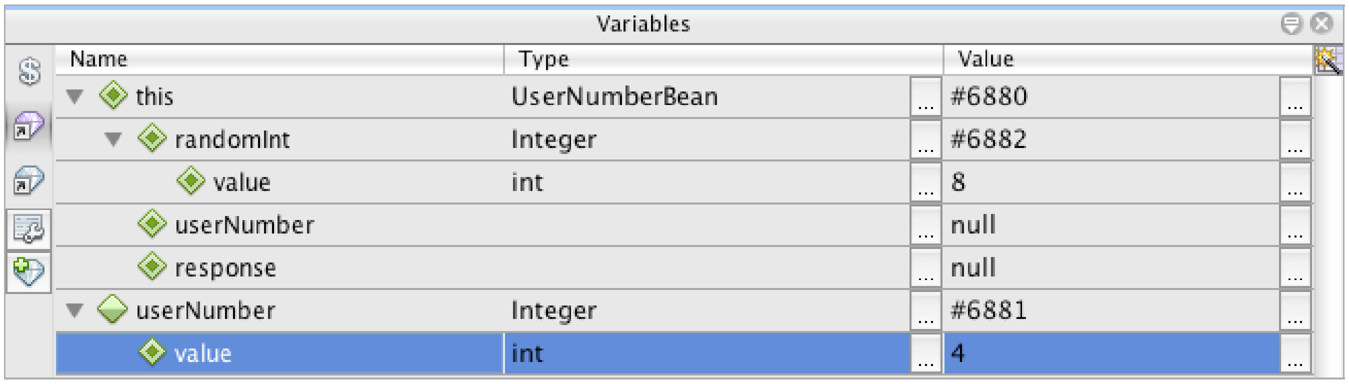
   - You might be prompted to confirm the server port for debugging the application.
   - If a Debug Project dialog displays, select the default 'Server side Java' option and click Debug.

9. In the browser, enter a number into the form and click the 'submit' button.
10. Switch back to the IDE and inspect the `UserNumberBean` class. The debugger is suspended within the `setUserNumber()` method.



```
    public void setUserNumber(Integer userNumber) {
        this.userNumber = userNumber;
    }
```

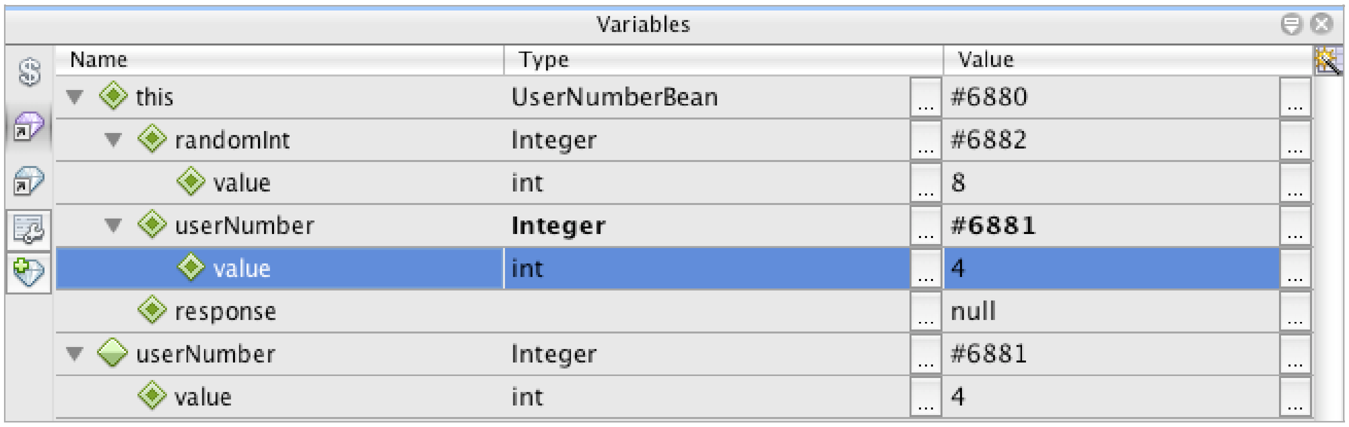
11. Open the Debugger's Variables window (Choose Window > Debugging > Variables, or press Ctrl-Shift-1). You see the variable values for the point at which the debugger is suspended.


Variables

| Name | Type | Value |
|---|---|---|
| this | UserNumberBean | #6880 |
| randomInt | Integer | #6882 |
| value | int | 8 |
| userNumber | | null |
| response | | null |
| userNumber | Integer | #6881 |
| value | int | 4 |

In the image above, a value of '`4`' is provided for the `userNumber` variable in the `setUserNumber()` signature. (The number 4 was entered into the form.) '`this`' refers to the `UserNumberBean` object that was created for the user session. Beneath it, you see that the value for the `userNumber` property is currently `null`.

12. In the Debugger toolbar, click the Step Into ( ) button. The debugger executes the line on which it is currently suspended. The Variables window refreshes, indicating changes from the execution.


Variables

| Name | Type | Value |
|---|---|---|
| this | UserNumberBean | #6880 |
| randomInt | Integer | #6882 |
| value | int | 8 |
| userNumber | **Integer** | **#6881** |
| value | int | 4 |
| response | | null |
| userNumber | Integer | #6881 |
| value | int | 4 |

The `userNumber` property is now set to the value entered in the form.

13. Choose Debug > Finish Debugger Session (Shift-F5; Shift-Fn-F5 on Mac) from the main menu to stop the debugger.

## response.xhtml

1. Open the `response.xhtml` page in the editor. Either double-click the `response.xhtml` node from the Projects window, or press Alt-Shift-O to use the Go to File dialog.
2. Comment out the HTML form element. Highlight the opening and closing HTML `<form>` tags and the code between them, then press Ctrl-/ (⌘-/ on Mac).

   **Note:** To highlight, either click and drag in the editor with your mouse, or, using the keyboard, hold Shift and press the arrow keys.

3. Uncomment the JSF HTML form component. Highlight the opening and closing `<h:form>` tags and the code between them, then press Ctrl-/ (⌘-/ on Mac).
   At this stage, your code between the `<body>` tags looks as follows:

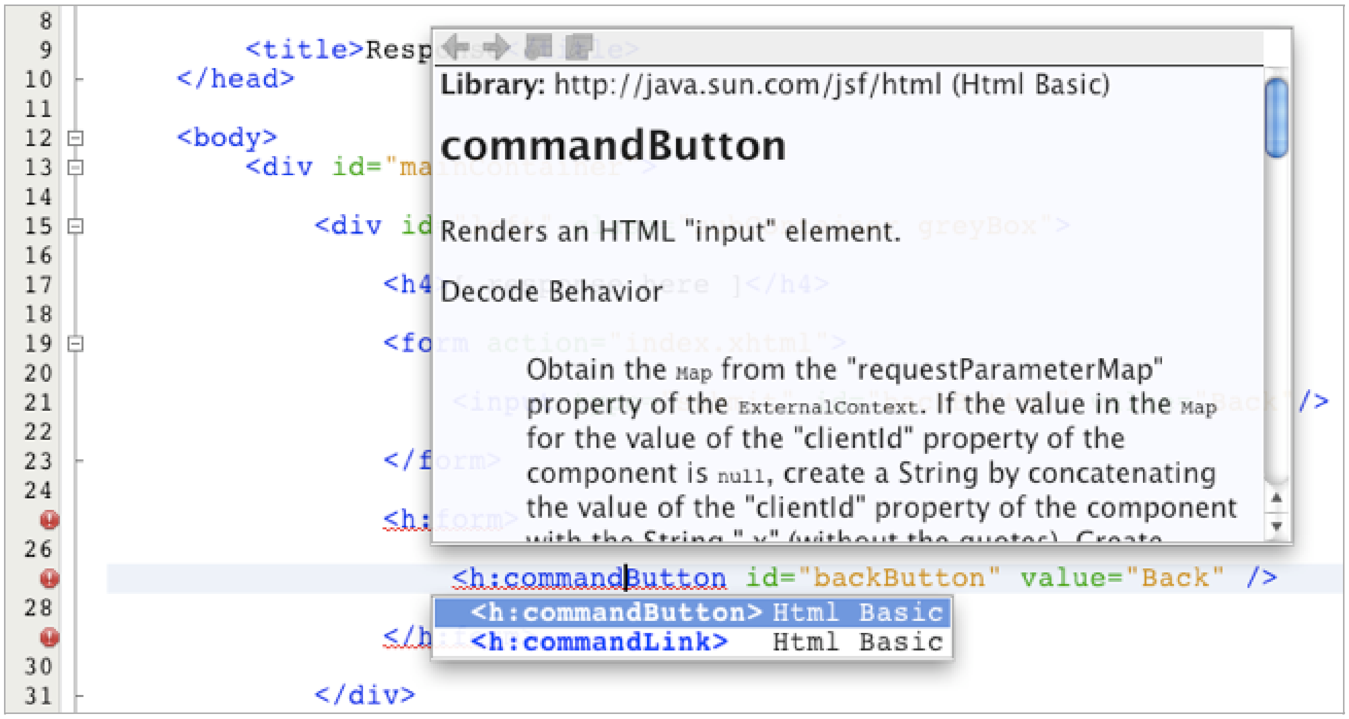
```
<body>
    <div id="mainContainer">

        <div id="left" class="subContainer greyBox">

            <h4>[ response here ]</h4>

            <!--<form action="index.xhtml">

                <input type="submit" id="backButton" value="Back"/>

            </form>-->

            <h:form>

                <h:commandButton id="backButton" value="Back" />

            </h:form>

        </div>

        <div id="right" class="subContainer">

            <img src="duke.png" alt="Duke waving" />
             <!--<h:graphicImage url="/duke.png" alt="Duke waving" />-->

        </div>
    </div>
</body>
```

After uncommenting the JSF HTML form component, the editor indicates that the `<h:form>` and `<h:commandButton>` tags haven't been declared.

4. To declare these components, use the IDE's code completion to add the tag library namespace to the page's `<html>` tag.

    Use the editor's code completion support to add required JSF namespaces to the file. When selecting a JSF or Facelets tag through code completion, the required namespace is automatically added to the document's root element. For more information, see JSF 2.x Support in NetBeans IDE.

Place your cursor on any of the undeclared tags and press Ctrl-Space. Code completion suggestions and documentation support displays.



```
        <title>Resp
    </head>

    <body>
        <div id="ma

            <div id

                <h4

                <fo


                </f

                <h:

                    <h:commandButton id="backButton" value="Back" />

                </h

            </div>
```

Library: http://java.sun.com/jsf/html (Html Basic)

**commandButton**

Renders an HTML "input" element.

Decode Behavior

Obtain the `Map` from the "requestParameterMap" property of the `ExternalContext`. If the value in the `Map` for the value of the "clientId" property of the component is `null`, create a String by concatenating the value of the "clientId" property of the component with the String ".x" (without the quotes). Create

Click Enter. (If there are multiple options, make sure to select the tag that is displayed in the editor before clicking Enter.) The JSF HTML tag library namespace is added to the `<html>` tag (shown in **bold** below), and the error indicators disappear.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

5. Specify the destination for the request that is invoked when the user clicks the form button. You want to set the button so that when a user clicks it, he or she is returned to the index page. To accomplish this, use the `commandButton`'s `action` attribute. Type in the code displayed in **bold**.

```
<h:form>

    <h:commandButton id="backButton" value="Back" action="index" />

</h:form>
```

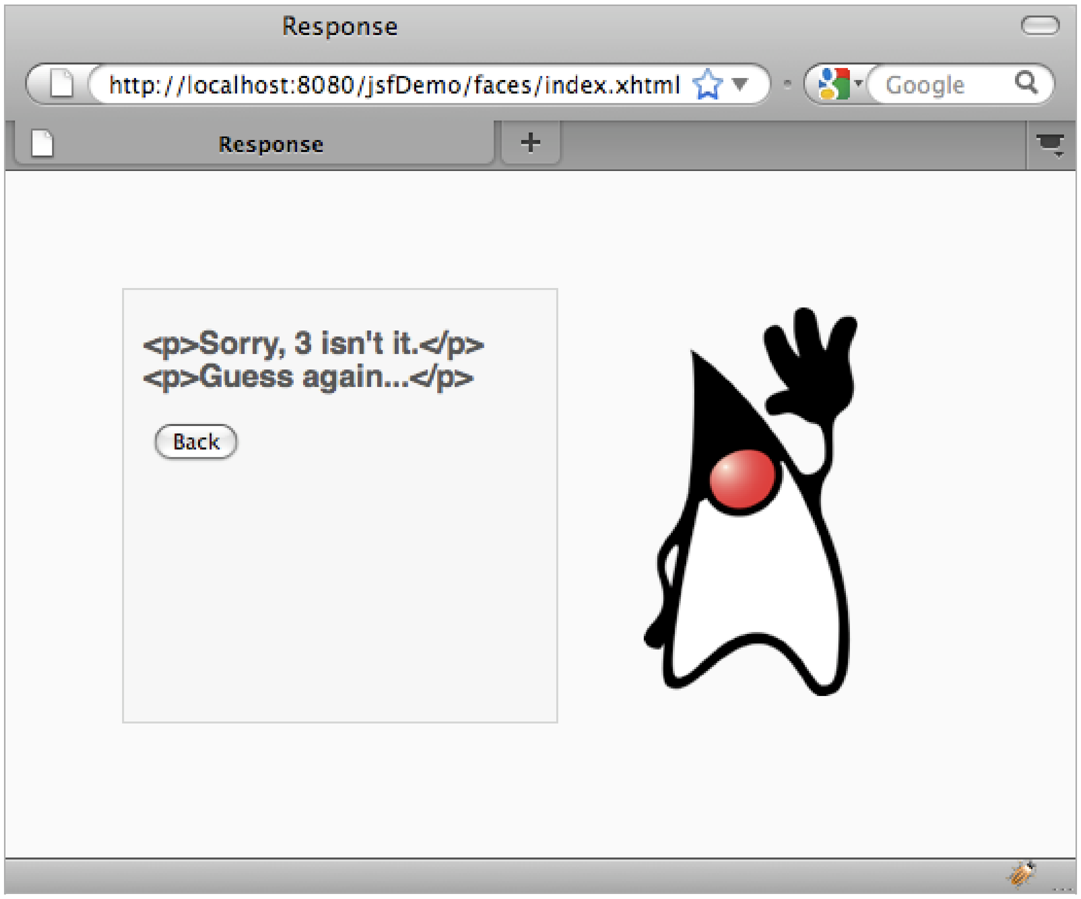
**Note:** By typing `action="index"`, you are relying on JSF's implicit navigation feature. When a user clicks the form button, the JSF runtime searches for a file named `index`. It assumes the file extension is the same as the extension used by file from which the request originated (`response`**`.xhtml`**) and looks for for the `index.xhtml` file in the same directory as the originating file (i.e., the webroot).

6. Replace the static "[ response here ]" text with the value of the `UserNumberBean`'s `response` property. To do this, use the JSF expression language. Enter the following (in **bold**).

```
<div id="left" class="subContainer greyBox">
```

```
<h4><h:outputText value="#{UserNumberBean.response}"/></h4>
```

7. Run the project (click the Run Project ( ▷ ) button, or press F6; fn-F6 on Mac). When the welcome page displays in the browser, enter a number and click `submit`. You see the response page display similar to the following (provided you did not guess the correct number).



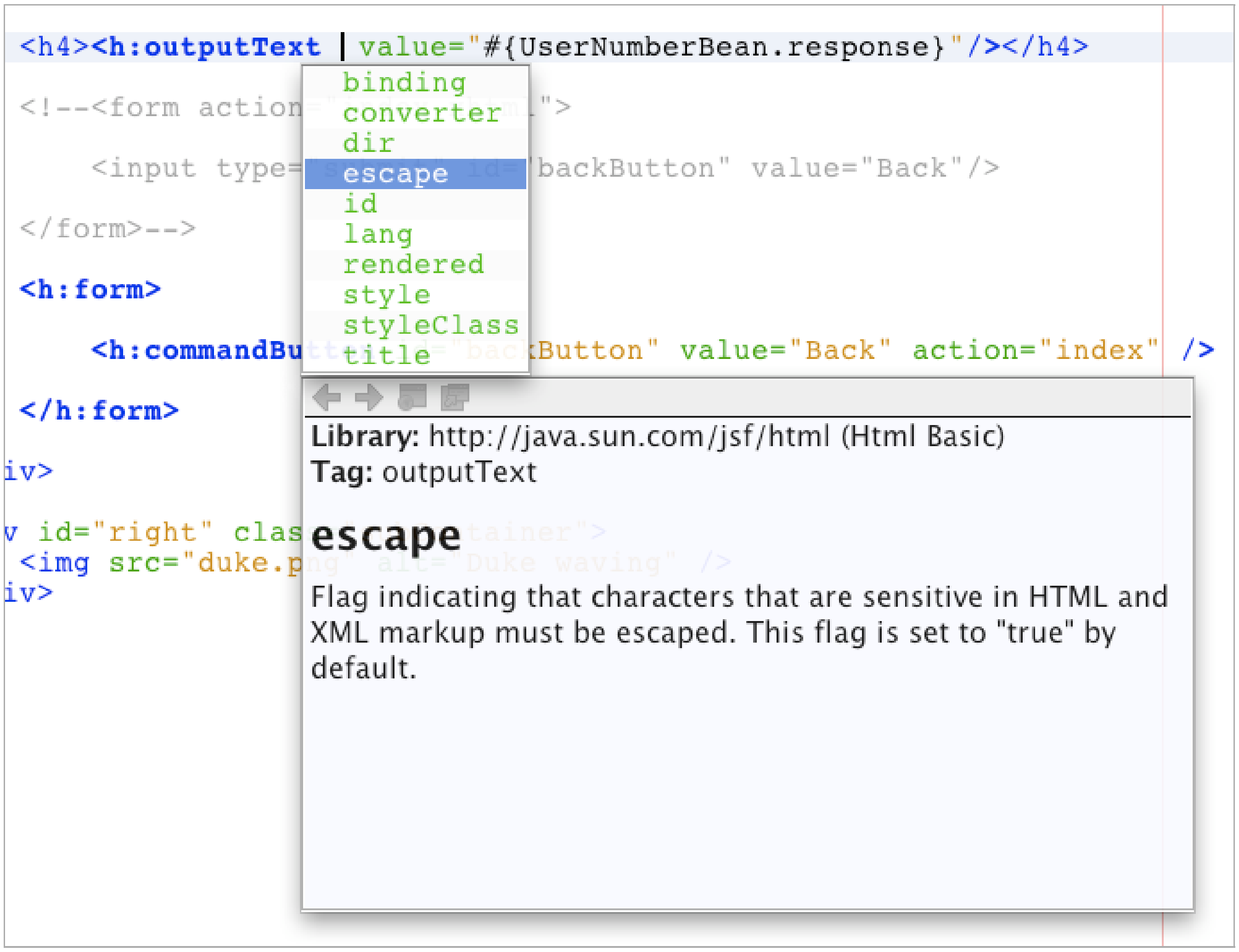
<p>Sorry, 3 isn't it.</p>
<p>Guess again...</p>

Two things are wrong with the current status of the response page:

1. The html `<p>` tags are displaying in the response message.
2. The Back button is not displaying in the correct location. (Compare it to the original version.)

The following two steps correct these points, respectively.

8. Set the `<h:outputText>` tag's `escape` attribute to `false`. Place your cursor between `outputText` and `value`, insert a space, then press Ctrl-Space to invoke code-completion. Scroll down to choose the `escape` attribute and inspect the documentation.



```
<h4><h:outputText | value="#{UserNumberBean.response}"/></h4>
<!--<form action=                   ">
    <input type=              'backButton" value="Back"/>
</form>-->
<h:form>
    <h:commandBu              Button" value="Back" action="index" />
</h:form>
iv>
v id="right" clas
 <img src="duke.png" alt="Duke waving" />
iv>
```

**Library:** http://java.sun.com/jsf/html (Html Basic)
**Tag:** outputText

**escape**

Flag indicating that characters that are sensitive in HTML and XML markup must be escaped. This flag is set to "true" by default.

As indicated by the documentation, the `escape` value is set to `true` by default. This means that any characters that would normally be parsed as html are included in the string, as shown above. Setting the value to `false` enables any characters that can be parsed as html to be rendered as such.

Click Enter, then type `false` as the value.

```
<h4><h:outputText escape="false" value="#{UserNumberBean.response}"/></h4>
```

9. Set the `<h:form>` tag's `prependId` attribute to `false`. Place your cursor just after 'm' in `<h:form>` and insert a space, then press Ctrl-Space to invoke code-completion. Scroll down to choose the `prependId` attribute and inspect the documentation. Then click Enter, and type `false` as the value.

```
<h:form prependId="false">
```

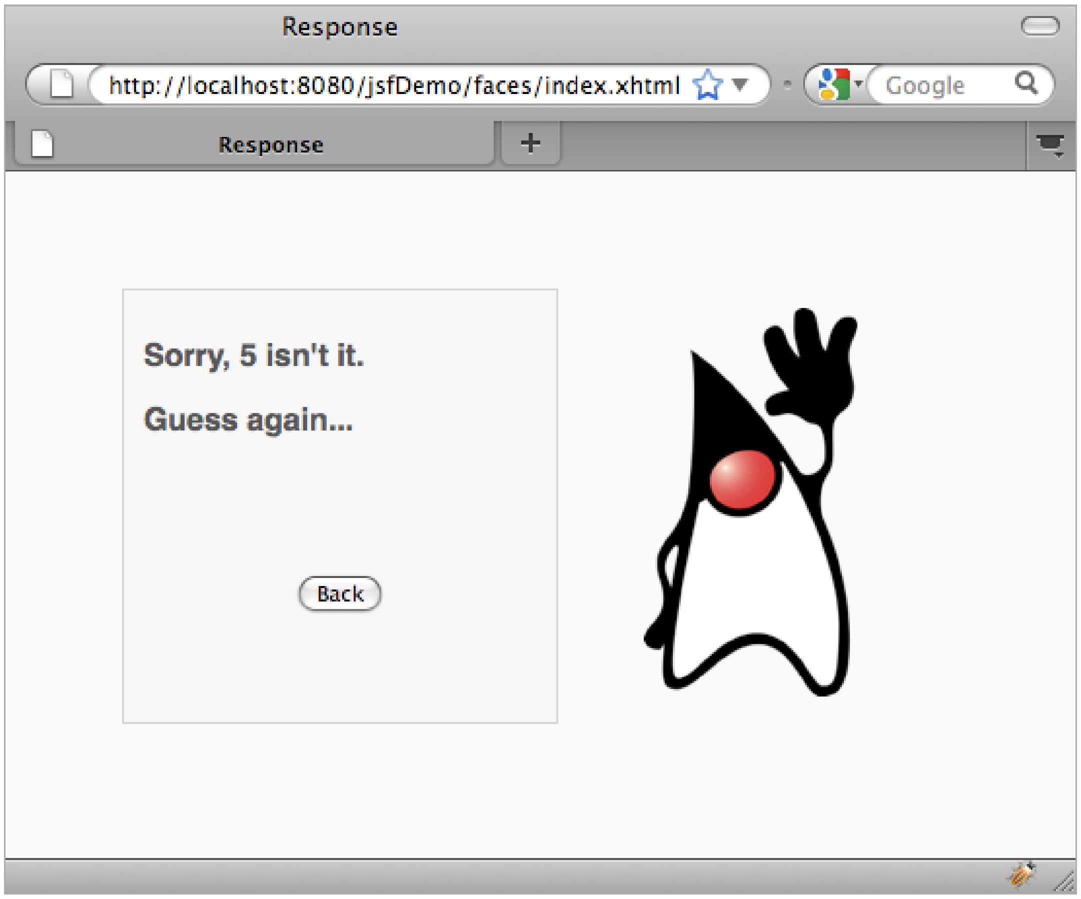
JSF applies internal id's to keep track of UI components. In the current example, if you inspect the source code of the rendered page, you will see something like the following:

```
<form id="j_idt5" name="j_idt5" method="post" action="/jsfDemo/faces/response.xhtml"
```

```
enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt5" value="j_idt5" />
    <input id="j_idt5:backButton" type="submit" name="j_idt5:backButton" value="Back"
/>
    <input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="7464469350430442643:-8628336969383888926" autocomplete="off" />
</form>
```

The id for the form element is `j_idt5`, and this id is *prepended* to the id for the Back button included in the form (shown in **bold** above). Because the Back button relies on the `#backButton` style rule (defined in `stylesheet.css`), this rule becomes obstructed when the JSF id is prepended. This can be avoided by setting `prependId` to `false`.

10. Run the project again (click the Run Project ( ) button, or press F6; fn-F6 on Mac). Enter a number in the welcome page, then click Submit. The response page now displays the response message without the `<p>` tags, and the Back button is positioned correctly.

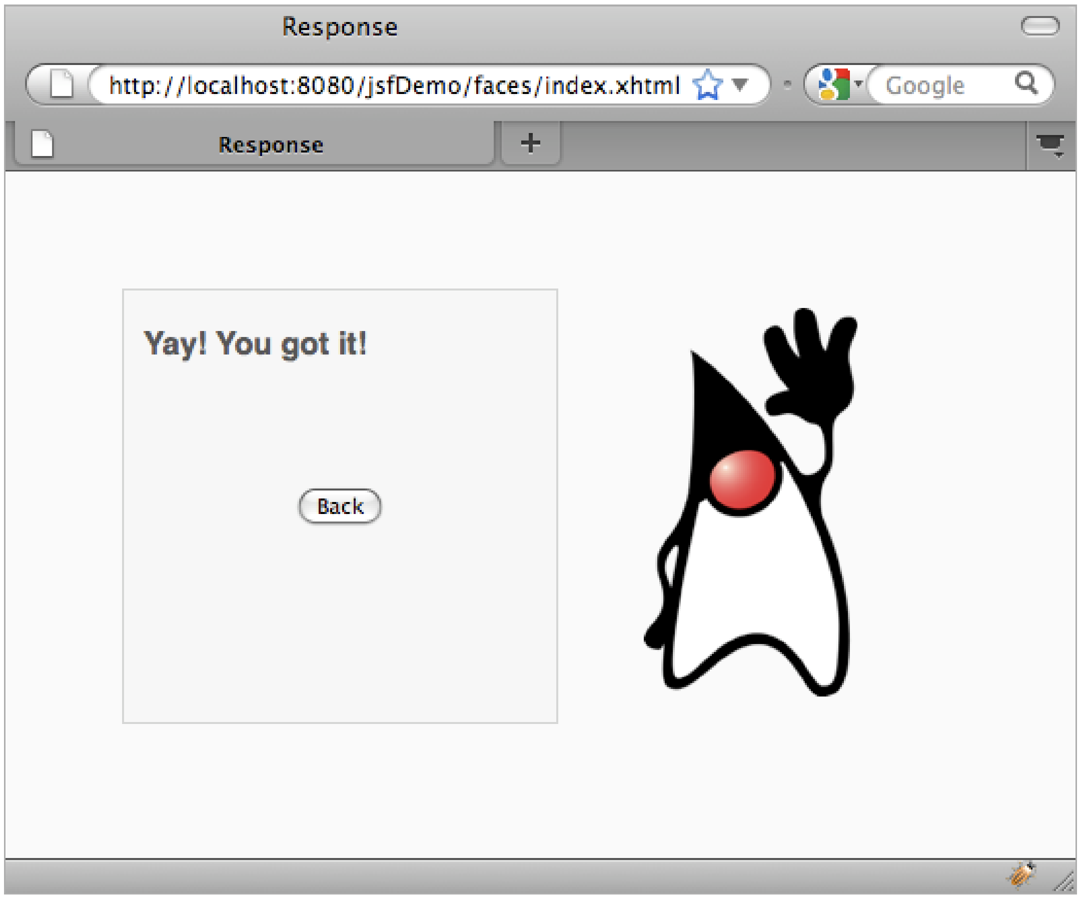


11. Click the Back button. Because the current value of `UserNumberBean`'s `userNumber` property is bound to the JSF `inputText` component, the number you previously entered is now displayed in the text field.

12. Inspect the server log in the IDE's Output window (Ctrl-4; ⌘-4 on Mac) to determine what the correct guess number is.

    If you can't see the server log for any reason, you can open it by switching to the Services window (Ctrl-5; ⌘-5 on Mac) and expanding the Servers node. Then right-click the GlassFish server on which the project is deployed and choose View Server Log. If you cannot see the number in the server log, try rebuilding the application by right-clicking the project node and choosing Clean and Build.

13. Type in the correct number and click Submit. The application compares your input with the currently saved number and displays the appropriate message.



14. Click the Back button again. Notice that the previously entered number is no longer displayed in the text field. Recall that `UserNumberBean`'s `getResponse()` method invalidates the current user session upon guessing the correct number.

## Applying a Facelets Template

Facelets has become the standard display technology for JSF 2.x. Facelets is a light-weight templating framework that supports all of the JSF UI components and is used to build and render the JSF component tree for application views. It also provides development support when EL errors occur by enabling you to inspect the stack trace, component tree, and scoped variables.

Although you may not have realized it, the `index.xhtml` and `response.xhtml` files you have been working with so far in the tutorial are Facelets pages. Facelets pages use the `.xhtml` extension and since you are working in a JSF 2.x project (The JSF 2.x libraries include the Facelets JAR files.), the views were able to appropriately render the JSF component tree.

The purpose of this section is to familiarize you with Facelets templating. For projects containing many views, it is often advantageous to apply a template file that defines the structure and appearance for multiple views. When servicing requests, the application inserts dynamically prepared content into the template file and sends the result back to the client. Although this project only contains two views (the welcome page and the response page), it is easy to see that they contain a lot of duplicated content. You can factor out this duplicated content into a Facelets template, and create template client files to handle content that is specific to the welcome and response pages.

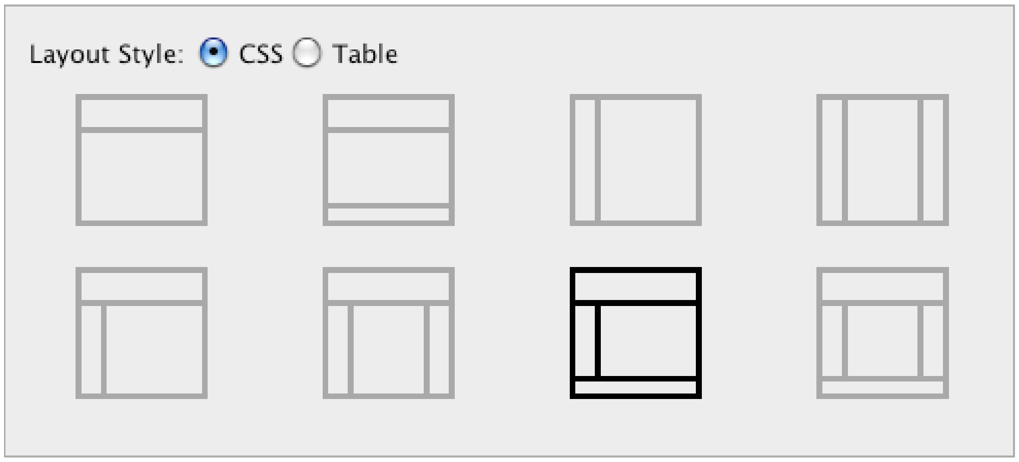
The IDE provides a Facelets Template wizard for creating Facelets templates, and a Facelets Template Client wizard for creating files that rely on a template. This section makes use of these wizards.

**Note:** The IDE also provides a JSF Page wizard that enables you to create individual Facelets pages for your project. For more information, see JSF 2.x Support in NetBeans IDE.

- Creating the Facelets Template File
- Creating Template Client Files

## Creating the Facelets Template File

1. Create a Facelets template file. Press Ctrl-N (⌘-N on Mac) to open the File wizard. Select the JavaServer Faces category, then Facelets Template. Click Next.
2. Type in `template` for the file name.
3. Choose from any of the eight layout styles and click Finish. (You will be using the existing stylesheet, so it does not matter which layout style you choose.)



The wizard generates the `template.xhtml` file and accompanying stylesheets based on your selection, and places these in a `resources` > `css` folder within the project's webroot.

After completing the wizard, the template file opens in the editor. To view the template in a browser, right-click in the editor and choose View.

4. Examine the template file markup. Note the following points:
   - The `facelets` tag library is declared in the page's `<html>` tag. The tag library has the `ui` prefix.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

   - The Facelets page uses the `<h:head>` and `<h:body>` tags instead of the html `<head>` and `<body>` tags. By using these tags, Facelets is able to construct a component tree that encompasses the entire page.
   - The page references the stylesheets that were also created when you completed the wizard.

```
<h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link href="./resources/css/default.css" rel="stylesheet" type="text/css"
/>
    <link href="./resources/css/cssLayout.css" rel="stylesheet" type="text/css"
/>
    <title>Facelets Template</title>
</h:head>
```

- `<ui:insert>` tags are used in the page's body for every compartment associated with the layout style you chose. Each `<ui:insert>` tag has a `name` attribute that identifies the compartment. For example:

```
<div id="top">
    <ui:insert name="top">Top</ui:insert>
</div>
```

5. Reexamine the welcome and response pages. The only content that changes between the two pages is the title and the text contained in the grey square. The template, therefore, can provide all remaining content.
6. Replace the entire content of your template file with the content below.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <link href="css/stylesheet.css" rel="stylesheet" type="text/css" />

        <title><ui:insert name="title">Facelets Template</ui:insert></title>
    </h:head>

    <h:body>

        <div id="left">
            <ui:insert name="box">Box Content Here</ui:insert>
        </div>

    </h:body>

</html>
```

The above code implements the following changes:

- The project's `stylesheet.css` file replaces the template stylesheet references created by the wizard.

- All `<ui:insert>` tags (and their containing `<div>` tags) have been removed, except for one named `box`.
- An `<ui:insert>` tag pair has been placed around the page title, and named `title`.

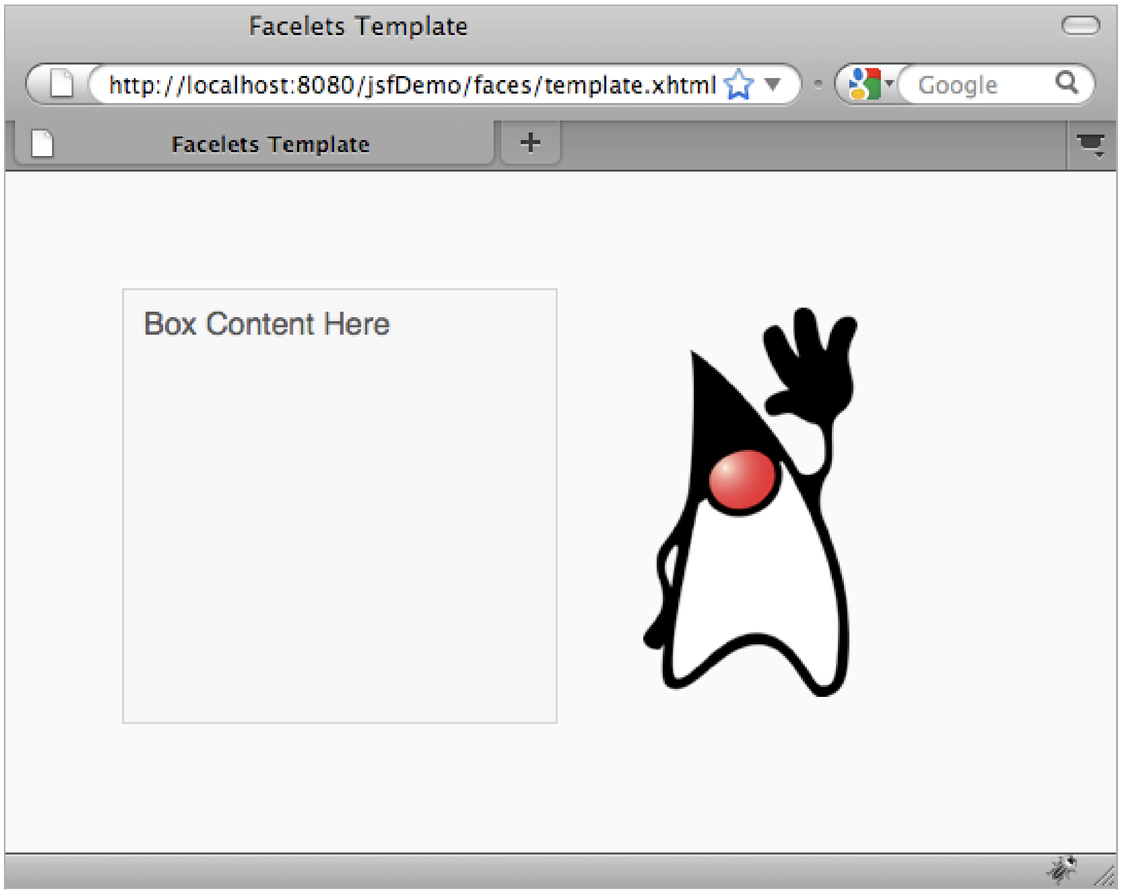
7. Copy relevant code from either the `index.xhtml` or `response.xhtml` file into the template. Add the content shown in **bold** below to the template file's `<h:body>` tags.

```
<h:body>
    <div id="mainContainer">
        <div id="left" class="subContainer greyBox">
            <ui:insert name="box">Box Content Here</ui:insert>
        </div>
        <div id="right" class="subContainer">
            <img src="duke.png" alt="Duke waving" />
        </div>
    </div>
</h:body>
```

8. Run the project. When the welcome page opens in the browser, modify the URL to the following:

```
http://localhost:8080/jsfDemo/faces/template.xhtml
```

The template file displays as follows:



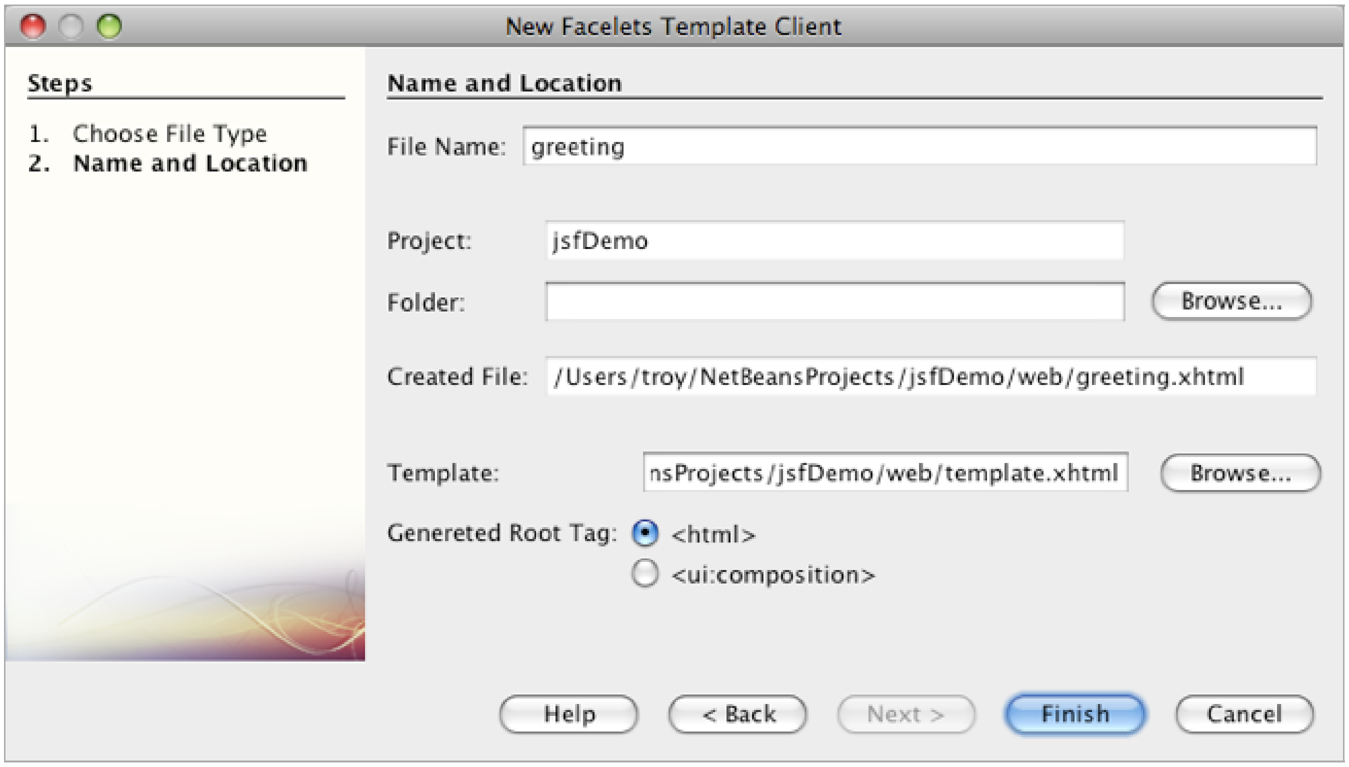
The project now contains a template file that provides the appearance and structure for all views. You can now create client files that invoke the template.

## Creating Template Client Files

Create template client files for the welcome and response pages. Name the template client file for the welcome page `greeting.xhtml`. For the response page, the file will be `response.xhtml`.

### greeting.xhtml

1. Press Ctrl-N (⌘-N on Mac) to open the New File wizard. Select the JavaServer Faces category, then select Facelets Template Client. Click Next.
2. Type in `greeting` for the file name.
3. Click the Browse button next to the Template field, then use the dialog that displays to navigate to the `template.xhtml` file you created in the previous section.


New Facelets Template Client

Steps
1. Choose File Type
2. **Name and Location**

Name and Location

File Name: greeting

Project: jsfDemo

Folder:

Created File: /Users/troy/NetBeansProjects/jsfDemo/web/greeting.xhtml

Template: nsProjects/jsfDemo/web/template.xhtml

Genereted Root Tag: [✓ <html>] <ui:composition>

4. Click Finish. The new `greeting.xhtml` template client file is generated and displays in the editor.
5. Examine the markup. Note the content hightlighted in **bold**.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

    <body>

        <ui:composition template="./template.xhtml">
```

```
            <ui:define name="title">
                title
            </ui:define>

            <ui:define name="box">
                box
            </ui:define>

        </ui:composition>

    </body>
</html>
```

The template client file references a template using the `<ui:composition>` tag's `template` attribute. Because the template contains `<ui:insert>` tags for `title` and `box`, this template client contains `<ui:define>` tags for these two names. The content that you specify between the `<ui:define>` tags is what will be inserted into the template between the `<ui:insert>` tags of the corresponding name.

6. Specify `greeting` as the title for the file. Make the following change in **bold**.

```
<ui:define name="title">
    Greeting
</ui:define>
```

7. Switch to the `index.xhtml` file (press Ctrl-Tab) and copy the content that would normally appear in the grey square that displays in the rendered page. Then switch back to `greeting.xhtml` and paste it into the template client file. (Changes in **bold**.)

```
<ui:define name="box">
    <h4>Hi, my name is Duke!</h4>

    <h5>I'm thinking of a number

        <br/>
        between
        <span class="highlight">0</span> and
        <span class="highlight">10</span>.</h5>

    <h5>Can you guess it?</h5>

    <h:form>
        <h:inputText size="2" maxlength="2" value="#{UserNumberBean.userNumber}" />
        <h:commandButton id="submit" value="submit" action="response" />
    </h:form>
</ui:define>
```

8. Declare the JSF HTML tag library for the file. Place your cursor on any of the tags that are flagged with an error (any tag

using the 'h' prefix), and press Ctrl-Space. Then select the tag from the list of code completion suggestions. The tag library namespace is added to the file's `<html>` tag (shown in **bold** below), and the error indicators disappear.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

If you place your cursor after the 'm' in `<h:form>` and press Ctrl-Space, the namespace is automatically added to the file. If only one logical option is available when pressing Ctrl-Space, it is immediately applied to the file. JSF tag libraries are automatically declared when invoking code completion on tags.

### response.xhtml

Because the project already contains a file named `response.xhtml`, and since you know what the template client file should look like now, modify the existing `response.xhtml` to become the template client file. (For purposes of this tutorial, just copy and paste the provided code.)

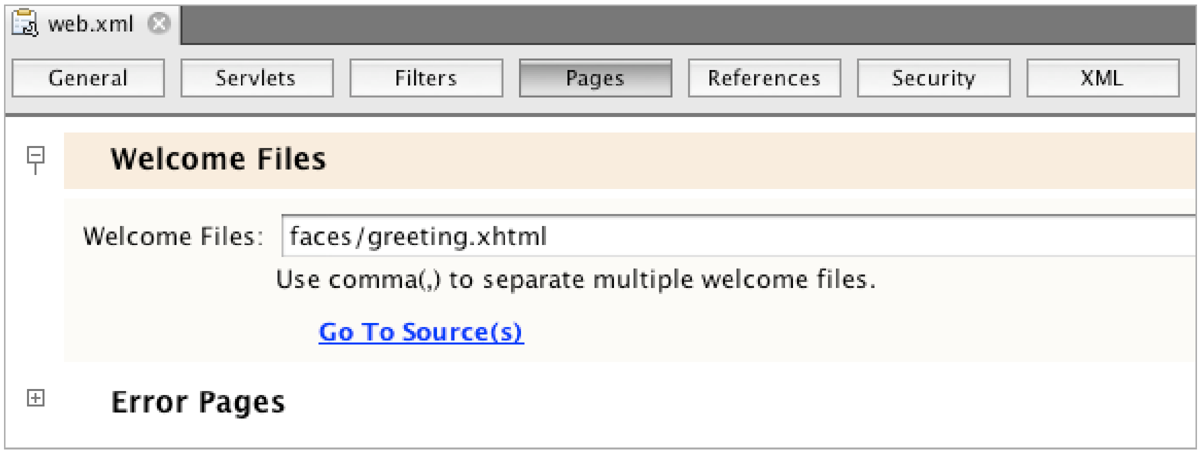
1. Open `response.xhtml` in the editor. (If it is already opened, press Ctrl-Tab and choose it.) Replace the contents of the entire file with the code below.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <body>

        <ui:composition template="./template.xhtml">

            <ui:define name="title">
                Response
            </ui:define>

            <ui:define name="box">
                <h4><h:outputText escape="false" value="#{UserNumberBean.response}"/>
</h4>

                <h:form prependId="false">

                    <h:commandButton id="backButton" value="Back" action="greeting"
/>

                </h:form>
            </ui:define>

        </ui:composition>
    </body>
</html>
```

Note that the file is identical to `greeting.xhtml`, except for the content specified between the `<ui:define>` tags for `title` and `box`.

2. In the project's `web.xml` deployment descriptor, modify the welcome file entry so that `greeting.xhtml` is the page that opens when the application is run.

   In the Projects window, double-click Configuration Files > `web.xml` to open it in the editor. Under the Pages tab, change the Welcome Files field to `faces/greeting.xhtml`.



Welcome Files: faces/greeting.xhtml

Use comma(,) to separate multiple welcome files.

Go To Source(s)

3. Run the project to see what it looks like in a browser. Press F6 (fn-F6 on Mac), or click the Run Project ( ▷ ) button in the main toolbar. The project is deployed to the GlassFish server, and opens in a browser.

Using the Facelets template and template client files, the application behaves in exactly the same way as it did previously. By factoring out duplicated code in the application's welcome and response pages, you succeeded in reducing the size of the application and eliminated the possibility of writing more duplicate code, should more pages be added at a later point. This can make development more efficient and easier to maintain when working in large projects.

*Send Feedback on This Tutorial*

## See Also

For more information about JSF 2.x, see the following resources.

### NetBeans Articles and Tutorials

- JSF 2.x Support in NetBeans IDE
- Generating a JavaServer Faces 2.x CRUD Application from a Database
- Scrum Toys - The JSF 2.0 Complete Sample Application
- Getting Started with Java EE Applications
- Java EE & Java Web Learning Trail

## External Resources

- JavaServer Faces Technology (Official homepage)
- JSR 314 Specification for JavaServer Faces 2.0
- The Java EE 7 Tutorial, Chapter 12: Developing with JavaServer Faces Technology
- GlassFish Project Mojarra (Official reference implementation for JSF 2.x)
- OTN Discussion Forums : JavaServer Faces
- JSF Central

## Blogs

- Ed Burns
- Jim Driscoll