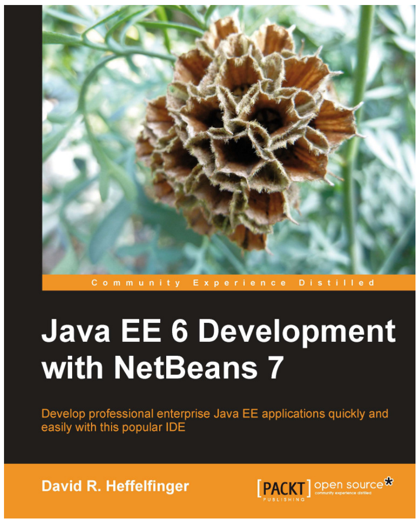
Java EE 6 Development with NetBeans 7

David R. Heffelfinger


Community Experience Distilled
Java EE 6 Development with NetBeans 7
Develop professional enterprise Java EE applications quickly and easily with this popular IDE
David R. Heffelfinger
PACKT open source

Chapter No. 4

"Developing Web Applications using JavaServer Faces 2.0"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Developing Web Applications using JavaServer Faces 2.0"

A synopsis of the book's content

Information on where to buy this book

About the Author

David R. Heffelfinger is the Chief Technology Officer of Ensode Technology, LLC, a software consulting firm based in the greater Washington DC area. He has been architecting, designing, and developing software professionally since 1995. He has been using Java as his primary programming language since 1996. He has worked on many large scale projects for several clients including IBM, Accenture, Lockheed Martin, Fannie Mae, Freddie Mac, the US Department of Homeland Security, and the US Department of Defense. He has a Masters degree in Software Engineering from Southern Methodist University. David is an editor in chief of `Ensode.net` (`http://www.ensode.net`), a web site about Java, Linux, and other technology topics.

I would like to thank everyone whose help made this book a reality. I would like to thank the Development Editors, Kartikey Pandey and Tariq Rakhange; and the Project Coordinator, Shubhanjan Chatterjee.

I would also like to thank the technical reviewers, Allan Bond, Arun Gupta, and Bruno Vernay for their insightful comments and suggestions.

Additionally, I would like to thank the NetBeans team at Oracle for developing such an outstanding IDE.

Finally, I would like to thank my wife and daughter, for putting up with the long hours of work that kept me away from the family.

Java EE 6 Development with NetBeans 7

Java EE 6, the latest version of the Java EE specification, adds several new features to simplify enterprise application development. New versions of existing Java EE APIs have been included in this latest version of Java EE. JSF 2.0 greatly simplifies web application development. JPA 2.0 features a new criteria API and several other enhancements. EJB session beans have been enhanced to support asynchronous method calls as well as a few other enhancements. Servlet 3.0 adds several new features such as additional method calls and making the `web.xml` deployment descriptor optional. Additionally, few new APIs have been added to Java EE, including JAX-RS, which simplifies RESTful web service development, and CDI, which helps integrate the different layers in a typical enterprise application.

NetBeans has been updated to support all features of Java EE 6, making development of Java EE 6 compliant application even quicker and simpler. This book will guide you through all the NetBeans features that make development of enterprise Java EE 6 applications a breeze.

What This Book Covers

Chapter 1, Getting Started with NetBeans provides an introduction to NetBeans, giving time saving tips and tricks that will result in more efficient development of Java applications.

Chapter 2, Developing Web Applications with Servlets and JSPs covers how NetBeans aids in the development of web applications using the servlet API and JavaServer Pages.

Chapter 3, Enhancing JSP Functionality with JSTL and Custom Tags shows how NetBeans can help us create maintainable web applications by taking advantage of **JavaServer Pages Standard Tag Library (JSTL)**, and it also covers how to write our own custom JSP tags.

Chapter 4, Developing Web Applications using JavaServer Faces 2.0 explains how NetBeans can help us easily develop web applications that take advantage of the JavaServer Faces 2.0 framework.

Chapter 5, Elegant Web Applications with PrimeFaces covers how to develop elegant web applications with full Ajax functionality by taking advantage of the PrimeFaces JSF component library bundled with NetBeans.

Chapter 6, Interacting with Databases through the Java Persistence API explains how NetBeans allows us to easily develop applications taking advantage of the **Java Persistence API (JPA)**, including how to automatically generate JPA entities from existing schemas. This chapter also covers how complete web-based applications can be generated with a few clicks from an existing database schema.

Chapter 7, Implementing the Business Tier with Session Beans discusses how NetBeans simplifies EJB 3.1 session bean development.

Chapter 8, Contexts and Dependency Injection (CDI) discusses how the new CDI API introduced in Java EE 6 can help us integrate the different layers of our application.

Chapter 9, Messaging with JMS and Message Driven Beans addresses Java EE messaging technologies such as **the Java Messaging Service (JMS)** and **Message Driven Beans (MDB)**, covering NetBeans features that simplify application development taking advantage of these APIs.

Chapter 10, SOAP Web Services with JAX-WS explains how NetBeans can help us easily develop SOAP web services based on the Java API for XML Web Services (JAX-WS) API.

Chapter 11, RESTful Web Services with JAX-RS covers JAX-RS, a new addition to the Java EE specification that simplifies development of RESTful web services.

Appendix A, Debugging Enterprise Applications with the NetBeans Debugger provides an introduction to the NetBeans debugger, and how it can be used to discover defects in our application.

Appendix B, Identifying Performance Issues with the NetBeans Profiler covers the NetBeans profiler, explaining how it can be used to analyze performance issues in our applications.

4

Developing Web Applications using JavaServer Faces 2.0

In the previous two chapters we covered how to develop web applications in Java using Servlets and JSPs. Although a lot of legacy applications have been written using these APIs, most modern Java web applications are written using some kind of web application framework. The standard framework for building web applications is **Java Server Faces** (**JSF**). In this chapter we will see how using JSF can simplify web application development.

The following topics will be covered in this chapter:

- Creating a JSF project with NetBeans
- Laying out JSF tags by taking advantage of the JSF `<h:panelGrid>` tag
- Using static and dynamic navigation to define navigation between pages
- Using the NetBeans **New JSF Managed Bean** wizard to create a JSF managed bean
- Implementing custom JSF validators
- How to easily generate JSF 2.0 templates via NetBeans wizards
- How to easily create JSF 2.0 composite components with NetBeans

Introduction to JavaServer faces

Before JSF existed, most Java web applications were typically developed using non-standard web application frameworks such as Apache Struts, Tapestry, Spring Web MVC, or many others. These frameworks are built on top of the Servlet and JSP standards, and automate a lot of functionality that needs to be manually coded when using these APIs directly.

Having a wide variety of web application frameworks available (at the time of writing, Wikipedia lists 31 Java web application frameworks, and this list is far from exhaustive!), often resulted in "analysis paralysis", that is, developers often spend an inordinate amount of time evaluating frameworks for their applications.

The introduction of JSF to the Java EE specification resulted in having a standard web application framework available in any Java EE compliant application server.

> We don't mean to imply that other web application frameworks are obsolete or that they shouldn't be used at all. However, a lot of organizations consider JSF the "safe" choice since it is part of the standard and should be well supported for the foreseeable future. Additionally, NetBeans offers excellent JSF support, making JSF a very attractive choice.

Strictly speaking, JSF is not a web application framework per se, but a component framework. In theory, JSF can be used to write applications that are not web-based, however, in practice JSF is almost always used for this purpose.

In addition to being the standard Java EE component framework, one benefit of JSF is that it provides good support for tools vendors, allowing tools such as NetBeans to take advantage of the JSF component model with drag and drop support for components.

Developing our first JSF application

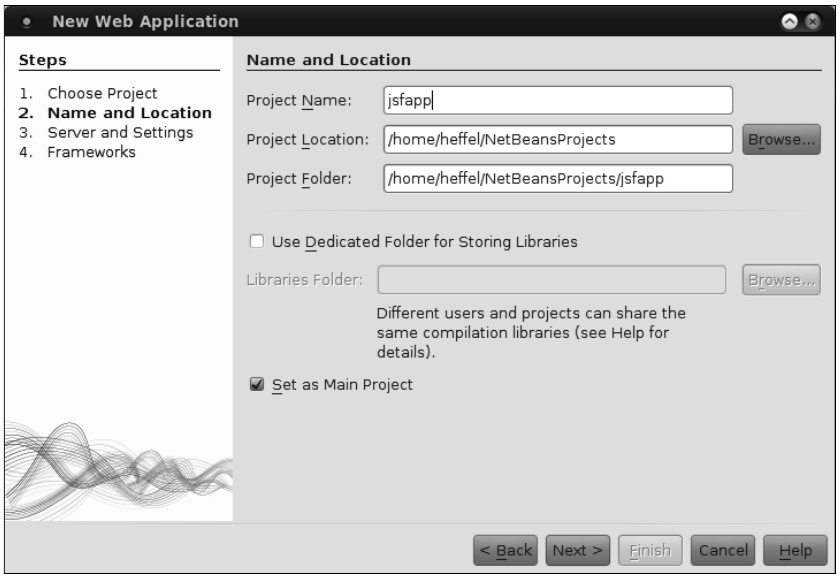
From an application developer's point of view, a JSF application consists of a series of XHTML pages containing custom JSF tags, one or more **JSF managed beans**, and an optional configuration file named `faces-config.xml`.

> `faces-config.xml` used to be required in JSF 1.x, however, in JSF 2.0, some conventions were introduced that reduce the need for configuration. Additonally, a lot of JSF configuration can be specified using annotations, reducing, and in some cases, eliminating the need for this XML configuration file.

Creating a new JSF project

To create a new JSF project, we need to go to **File | New Project**, select the **Java Web** project category, and **Web Application** as the project type.

After clicking **Next>**, we need to enter a project name, and optionally change other information for our project, although NetBeans provides sensible defaults.


New Web Application

Steps
1. Choose Project
2. **Name and Location**
3. Server and Settings
4. Frameworks

Name and Location

Project Name: jsfapp

Project Location: /home/heffel/NetBeansProjects

Project Folder: /home/heffel/NetBeansProjects/jsfapp

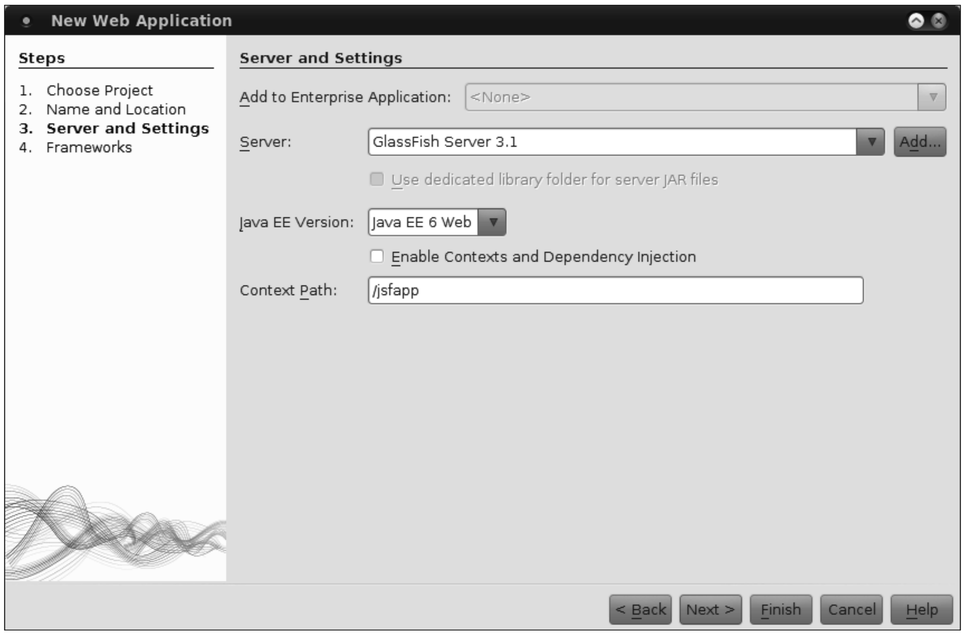
☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Set as Main Project

On the next page in the wizard, we can select the server, Java EE version, and context path of our application. In our example we will simply pick the default values.


New Web Application

Steps
1. Choose Project
2. Name and Location
3. **Server and Settings**
4. Frameworks

Server and Settings

Add to Enterprise Application: <None>

Server: GlassFish Server 3.1

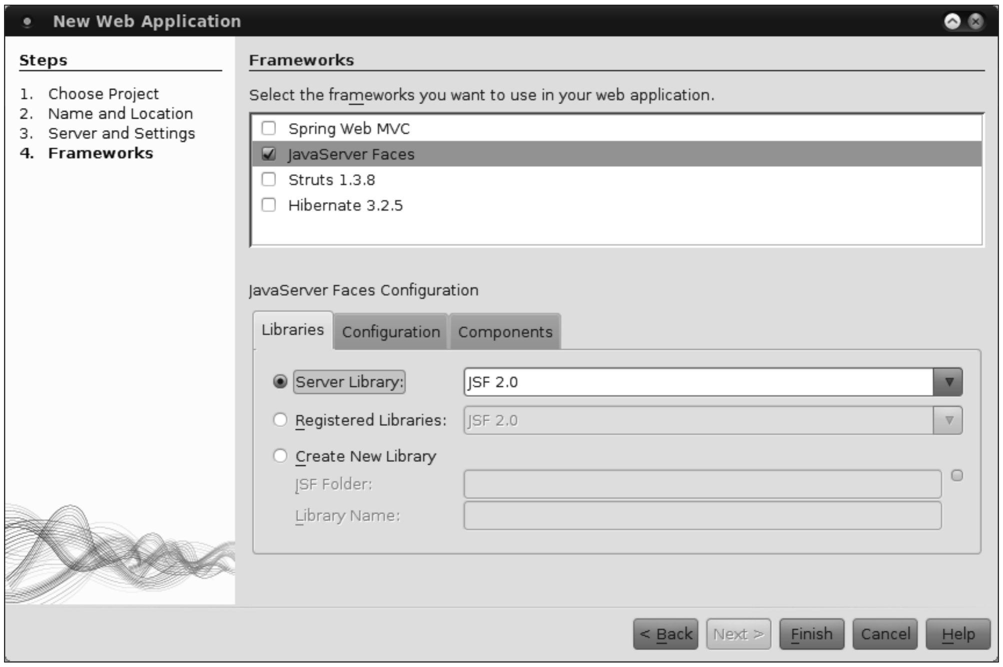
☐ Use dedicated library folder for server JAR files

Java EE Version: Java EE 6 Web

☐ Enable Contexts and Dependency Injection

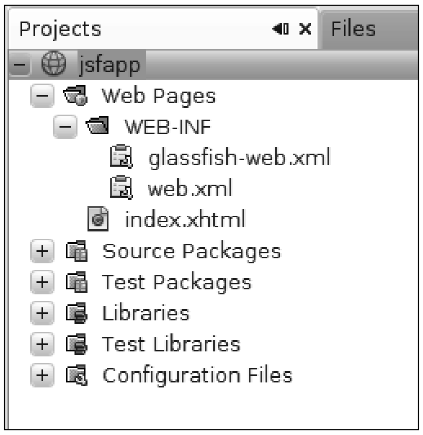
Context Path: /jsfapp

On the next page of the new project wizard, we can select what frameworks our web application will use.



Unsurprisingly, for JSF applications we need to select the JavaServer Faces framework.

When clicking on **Finish**, the wizard generates a skeleton JSF project for us, consisting of a single facelet file called `index.xhtml`, a `web.xml` configuration file.



web.xml is the standard, optional configuration file needed for Java web applications, this file became optional in version 3.0 of the Servlet API, which was introduced with Java EE 6. In many cases, web.xml is not needed anymore, since most of the configuration options can now be specified via annotations. For JSF applications, however, it is a good idea to add one, since it allows us to specify the **JSF project stage**.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
```

As we can see, NetBeans automatically sets the JSF project stage to Development, setting the project stage to development configures JSF to provide additional debugging help not present in other stages. For example, one common problem when developing a page is that while a page is being developed, validation for one or more of the fields on the page fails, but the developer has not added an <h:message> or <h:messages> tag to the page (more on this later). When this happens and the form is submitted, the page seems to do nothing, or page navigation doesn't seem to be working. When setting the project stage to Development, these validation errors will automatically be added to the page, without the developer having to explicitly add one of these tags to the page (we should, of course, add the tags before releasing our code to production, since our users will not see the automatically generated validation errors).

The following are the valid values for the `javax.faces.PROJECT_STAGE` context parameter for the faces servlet:

- Development
- Production
- SystemTest
- UnitTest

As we previously mentioned, the `Development` project stage adds additional debugging information to ease development. The `Production` project stage focuses on performance. The other two valid values for the project stage (`SystemTest` and `UnitTest`), allow us to implement our own custom behavior for these two phases. The `javax.faces.application.Application` class has a `getProjectStage()` method that allows us to obtain the current project stage. Based on the value of this method, we can implement the code that will only be executed in the appropriate stage. The following code snippet illustrates this:

```
public void someMethod() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        Application application = facesContext.getApplication();
        ProjectStage projectStage = application.getProjectStage();

        if (projectStage.equals(ProjectStage.Development)) {
            //do development stuff
        } else if (projectStage.equals(ProjectStage.Production)) {
            //do production stuff
        } else if (projectStage.equals(ProjectStage.SystemTest)) {
            // do system test stuff
        } else if (projectStage.equals(ProjectStage.UnitTest)) {
            //do unit test stuff
        }
    }
```

As illustrated in the snippet above, we can implement the code to be executed in any valid project stage, based on the return value of the `getProjectStage()` method of the `Application` class.

When creating a Java Web project using JSF, a facelet is automatically generated.

The generated facelet file looks like this:

```
<?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"
```

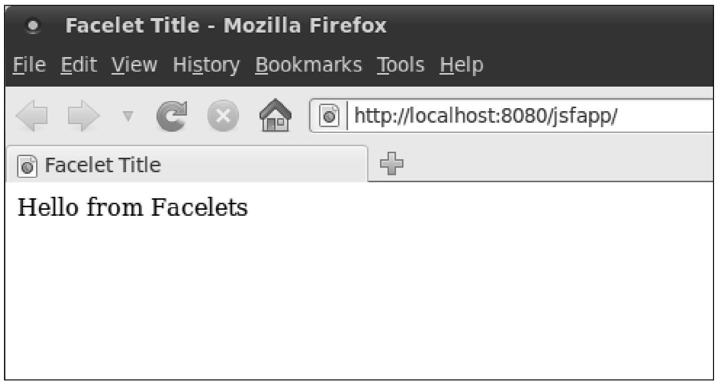
```
         xmlns:h="http://java.sun.com/jsf/html">
       <h:head>
           <title>Facelet Title</title>
       </h:head>
       <h:body>
           Hello from Facelets
       </h:body>
   </html>
```

As we can see, a facelet is nothing but an XHTML file using some facelets-specific XML name spaces. In the automatically generated page above, the following namespace definition allows us to use the "h" (for HTML) JSF component library:

```
xmlns:h="http://java.sun.com/jsf/html"
```

The above namespace declaration allows us to use JSF specific tags such as `<h:head>` and `<h:body>` which are a drop in replacement for the standard HTML/XHTML `<head>` and `<body>` tags, respectively.

The application generated by the new project wizard is a simple, but complete JSF web application. We can see it in action by right-clicking on our project in the project window and selecting **Run**. At this point the application server is started (if it wasn't already running), the application is deployed and the default system browser opens, displaying our application's default page.

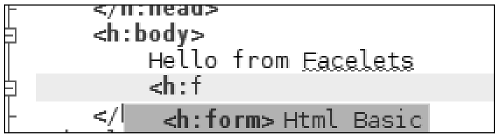

Facelet Title - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/
Facelet Title

Hello from Facelets

Modifying our page to capture user data

The generated application, of course, is nothing but a starting point for us to create a new application. We will now modify the generated `index.xhtml` file to collect some data from the user.

The first thing we need to do is add an `<h:form>` tag to our page. The `<h:form>` tag is equivalent to the `<form>` tag in standard HTML pages. After typing the first few characters of the `<h:form>` tag into the page, and hitting *Ctrl+Space*, we can take advantage of NetBeans' excellent code completion.

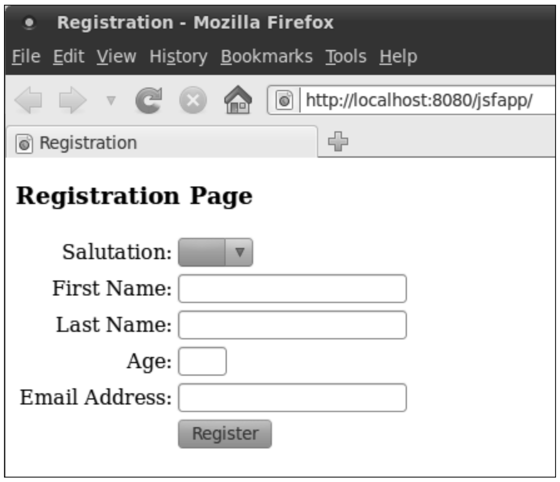


```
</h:head>
<h:body>
    Hello from Facelets
    <h:f
</    <h:form> Html Basic
```

After adding the `<h:form>` tag and a number of additional JSF tags, our page now looks like this:

```
<?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:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <h:head>
        <title>Registration</title>
        <h:outputStylesheet library="css" name="styles.css"/>
    </h:head>
    <h:body>
        <h3>Registration Page</h3>
        <h:form>
            <h:panelGrid columns="3"
                         columnClasses="rightalign,leftalign,leftalign">
                <h:outputLabel value="Salutation: " for="salutation"/>
                <h:selectOneMenu id="salutation" label="Salutation"
                                 value="#{registrationBean.salutation}" >
                    <f:selectItem itemLabel="" itemValue=""/>
                    <f:selectItem itemLabel="Mr." itemValue="MR"/>
                    <f:selectItem itemLabel="Mrs." itemValue="MRS"/>
                    <f:selectItem itemLabel="Miss" itemValue="MISS"/>
                    <f:selectItem itemLabel="Ms" itemValue="MS"/>
                    <f:selectItem itemLabel="Dr." itemValue="DR"/>
                </h:selectOneMenu>
                <h:message for="salutation"/>
                <h:outputLabel value="First Name:" for="firstName"/>
                <h:inputText id="firstName" label="First Name"
                             required="true"
                             value="#{registrationBean.firstName}" />
                <h:message for="firstName" />
                <h:outputLabel value="Last Name:" for="lastName"/>
                <h:inputText id="lastName" label="Last Name"
                             required="true"
                             value="#{registrationBean.lastName}" />
                <h:message for="lastName" />
                <h:outputLabel for="age" value="Age:"/>
                <h:inputText id="age" label="Age" size="2"
                             value="#{registrationBean.age}"/>
                <h:message for="age"/>
                <h:outputLabel value="Email Address:" for="email"/>
                <h:inputText id="email" label="Email Address"
                             required="true"
                             value="#{registrationBean.email}">
                </h:inputText>
                <h:message for="email" />
                <h:panelGroup/>
                <h:commandButton id="register" value="Register"
                                 action="confirmation" />
            </h:panelGrid>
        </h:form>
    </h:body>
</html>
```

The following screenshot illustrates how our page will be rendered at runtime:


Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/
Registration
Registration Page
Salutation:
First Name:
Last Name:
Age:
Email Address:
Register

All JSF input fields must be inside an `<h:form>` tag. The `<h:panelGrid>` helps us to easily lay out JSF tags on our page. It can be thought of as a grid where other JSF tags will be placed. The `columns` attribute of the `<h:panelGrid>` tag indicates how many columns the grid will have, each JSF component inside the `<h:panelGrid>` component will be placed in an individual cell of the grid. When the number of components matching the value of the `columns` attribute (three in our example) has been placed inside `<h:panelGrid>`, a new row is automatically started.

The following table illustrates how tags will be laid out inside an `<h:panelGrid>` tag:

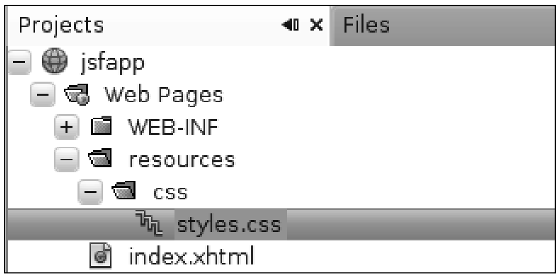
First Tag	Second Tag	Third Tag
Fourth Tag	Fifth Tag	Sixth Tag
Seventh Tag	Eighth Tag	Ninth Tag

Each row in our `<h:panelGrid>` consists of an `<h:outputLabel>` tag, an input field, and an `<h:message>` tag.

The `columnClasses` attribute of `<h:panelGrid>` allows us to assign CSS styles to each column inside the panel grid, its `value` attribute must consist of a comma separated list of CSS styles (defined in a CSS stylesheet). The first style will be applied to the first column, the second style will be applied to the second column, the third style will be applied to the third column, so on and so forth. Had our panel grid had more than three columns, then the fourth column would have been styled using the first style in the `columnClasses` attribute, the fifth column would have been styled using the second style in the `columnClasses` attribute, so on and so forth.

If we wish to style rows in an `<h:panelGrid>`, we can do so with its `rowClasses` attribute, which works the same way that the `columnClasses` works for columns.

Notice the `<h:outputStylesheet>` tag inside `<h:head>` near the top of the page, this is a new tag that was introduced in JSF 2.0. One new feature that JSF 2.0 brings to the table is standard resource directories. Resources such as CSS stylesheets, JavaScript files, images, and so on, can be placed under a top level directory named `resources`, and JSF tags will have access to those resources automatically. In our NetBeans project, we need to place the `resources` directory under the **Web Pages** folder.



We then need to create a subdirectory to hold our CSS stylesheet (by convention, this directory should be named `css`), then we place our CSS stylesheet(s) on this subdirectory.

The CSS stylesheet for our example is very simple, therefore it is not shown. However, it is part of the code download for this chapter.

The value of the library attribute in `<h:outputStylesheet>` must match the directory where our CSS file is located, and the value of its `name` attribute must match the CSS file name.

In addition to CSS files, we should place any JavaScript files in a subdirectory called `javascript` under the `resources` directory. The file can then be accessed by the `<h:outputScript>` tag using `"javascript"` as the value of its library attribute and the file name as the value of its name attribute.

Similarly, images should be placed in a directory called `images` under the `resources` directory. These images can then be accessed by the JSF `<h:graphicImage>` tag, where the value of its library attribute would be `"images"` and the value of its `name` attribute would be the corresponding file name.

Now that we have discussed how to lay out elements on the page and how to access resources, let's focus our attention on the input and output elements on the page.

The `<h:outputLabel>` tag generates a label for an input field in the form, the value of its `for` attribute must match the value of the `id` attribute of the corresponding input field.

`<h:message>` generates an error message for an input field, the value of its `for` field must match the value of the `id` attribute for the corresponding input field.

The first row in our grid contains an `<h:selectOneMenu>`. This tag generates an HTML `<select>` tag on the rendered page.

Every JSF tag has an `id` attribute, the value for this attribute must be a string containing a unique identifier for the tag. If we don't specify a value for this attribute, one will be generated automatically. It is a good idea to explicitly state the ID of every component, since this ID is used in runtime error messages. Affected components are a lot easier to identify if we explicitly set their IDs.

When using `<h:label>` tags to generate labels for input fields, or when using `<h:message>` tags to generate validation errors, we need to explicitly set the value of the `id` tag, since we need to specify it as the value of the `for` attribute of the corresponding `<h:label>` and `<h:message>` tags.

Every JSF input tag has a `label` attribute. This attribute is used to generate validation error messages on the rendered page. If we don't specify a value for the `label` attribute, then the field will be identified in the error message by its ID.

Each JSF input field has a `value` attribute, in the case of `<h:selectOneMenu>`, this attribute indicates which of the options in the rendered `<select>` tag will be selected. The value of this attribute must match the value of the `itemValue` attribute of one of the nested `<f:selectItem>` tags. The value of this attribute is usually a **value binding expression**, that means that the value is read at runtime from a JSF managed bean. In our example, the value binding expression `#{registrationBean.salutation}` is used. What will happen is at runtime JSF will look for a managed bean named `registrationBean`, and look for an attribute named `salutation` on this bean, the getter method for this attribute will be invoked, and its return value will be used to determine the selected value of the rendered HTML `<select>` tag.

Nested inside the `<h:selectOneMenu>` there are a number of `<f:selectItem>` tags. These tags generate HTML `<option>` tags inside the HTML `<select>` tag generated by `<h:selectOneMenu>`. The value of the `itemLabel` attribute is the value that the user will see while the value of the `itemValue` attribute will be the value that will be sent to the server when the form is submitted.

All other rows in our grid contain `<h:inputText>` tags, this tag generates an HTML `input` field of type `text`, which accept a single line of typed text as input. We explicitly set the `id` attribute of all of our `<h:inputText>` fields, this allows us to refer to them from the corresponding `<h:outputLabel>` and `<h:message>` fields. We also set the `label` attribute for all of our `<h:inputText>` tags, this results in more user-friendly error messages.

Some of our `<h:inputText>` fields require a value, these fields have their `required` attribute set to `true`, each JSF input field has a `required` attribute, if we need to require the user to enter a value for this attribute, then we need to set this attribute to `true`. This attribute is optional, if we don't explicitly set a value for it, then it defaults to `false`.

In the last row of our grid, we added an empty `<h:panelGroup>` tag. The purpose of this tag is to allow adding several tags into a single cell of an `<h:panelGrid>`. Any tags placed inside this tag are placed inside the same cell of the grid where `<h:panelGrid>` is placed. In this particular case, all we want to do is to have an "empty" cell in the grid so that the next tag, `<h:commandButton>`, is aligned with the input fields in the rendered page.

`<h:commandButton>` is used to submit a form to the server. The value of its `value` attribute is used to generate the text of the rendered button. The value of its `action` attribute is used to determine what page to display after the button is pressed.

In our example, we are using **static navigation**. When using JSF static navigation, the value of the `action` attribute of a command button is hard-coded in the markup.

When using static navigation, the value of the action attribute of `<h:commandButton>` corresponds to the name of the page we want to navigate to, minus its `.xhtml` extension. In our example, when the user clicks on the button, we want to navigate to a file named `confirmation.xhtml`, therefore we used a value of `"confirmation"` for its `action` attribute.

An alternative to static navigation is **dynamic navigation**. When using dynamic navigation, the value of the `action` attribute of the command button is a value binding expression resolving to a method returning a `String` in a managed bean. The method may then return different values based on certain conditions. Navigation would then proceed to a different page depending on the value of the method.

As long as it returns a `String`, the managed bean method executed when using dynamic navigation can contain any logic inside it, and is frequently used to save data in a managed bean into a database.

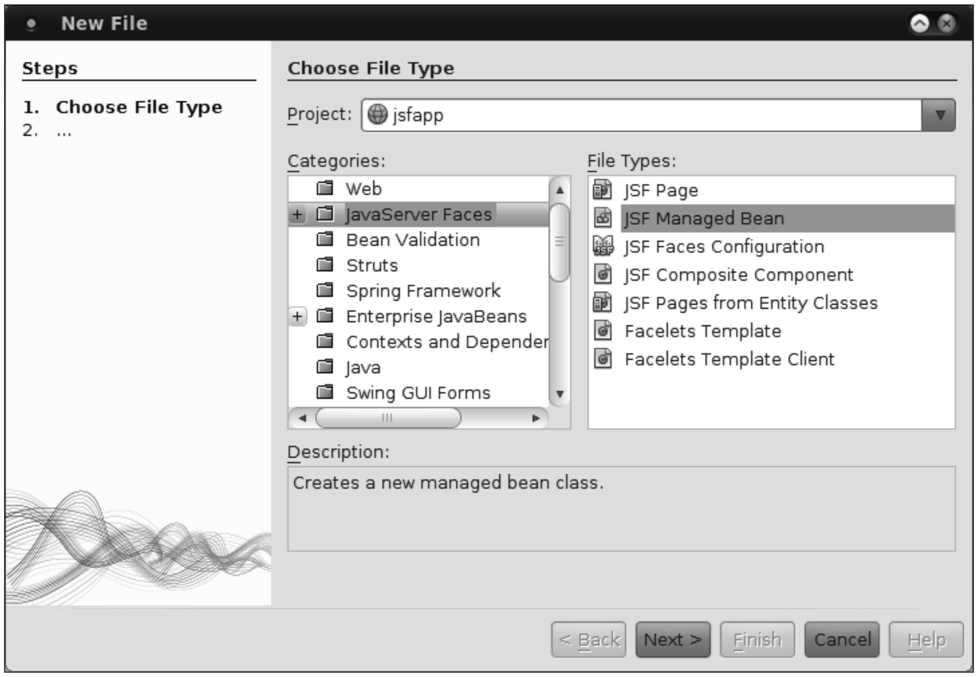
When using dynamic navigation, the return value of the method executed when clicking the button must match the name of the page we want to navigate to (again, minus the file extension).

In earlier versions of JSF, it was necessary to specify navigation rules in `faces-config.xml`, with the introduction of the conventions introduced in the previous paragraphs, this is no longer necessary.

Creating our managed bean

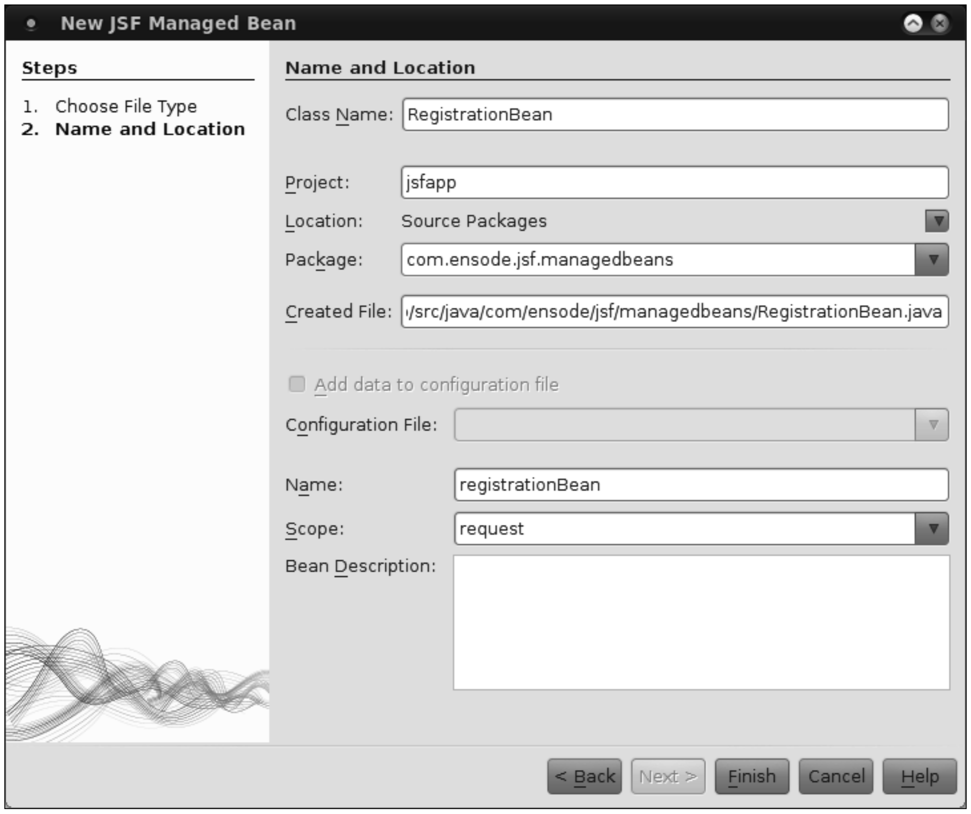
JSF managed beans are standard JavaBeans that are used to hold user-entered data in JSF applications.

In order to create a new managed bean, we need to go to **File | New File...**, select **JavaServer Faces** from the category list, and **JSF Managed Bean** from the file type list.



Description:
Creates a new managed bean class.

On the next screen in the wizard, we need to enter a name for our managed bean, as well as a package:


New JSF Managed Bean

Steps
1. Choose File Type
2. Name and Location

Name and Location

Class Name: RegistrationBean

Project: jsfapp

Location: Source Packages

Package: com.ensode.jsf.managedbeans

Created File: ı/src/java/com/ensode/jsf/managedbeans/RegistrationBean.java

Add data to configuration file

Configuration File:

Name: registrationBean

Scope: request

Bean Description:

Most default values are sensible and in most cases can be accepted. The only one we should change if necessary is the **Scope** field.

Managed beans can have different scopes. A scope of request means that the bean is only available in a single HTTP request. Managed beans can also have session scope, in which case they are available in a single user's HTTP session. A scope of application means that the bean is accessible to all users in the application, across user sessions. Managed beans can also have a scope of none, which means that the managed bean is not stored at any scope, but is created on demand as needed. Additionally, managed beans can have a scope of view, in which case the bean is available until the user navigates to another page. View scoped managed beans are available across Ajax requests.

We should select the appropriate scope for our managed bean, in our particular example, the default request scope will meet our needs.

After finishing the wizard, a boilerplate version of our managed bean is created in the specified package.

The generated managed bean source simply consists of the annotated managed bean class containing a single public no argument constructor.

```
package com.ensode.jsf.managedbeans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class RegistrationBean {

    /** Creates a new instance of RegistrationBean */
    public RegistrationBean() {
    }
}
```

The `@ManagedBean` annotation marks the class as a JSF managed bean. By default, the managed bean name defaults to the class name (`RegistrationBean`, in our case) with its first character switched to lower case (`registrationBean`, in our case). If we want to override the default name, we can do it by specifying a different name in the NetBeans **New JSF Managed Bean** wizard, or by simply setting the name attribute of `@ManagedBean` to the desired value. In general, sticking to the defaults allows for more readable and maintainable code therefore we shouldn't deviate from them unless we have a good reason.

With the addition of any Java class annotated with `@ManagedBean` in our project, we no longer need to register FacesServlet in `web.xml` as the JSF runtime in the application server will automatically register the servlet.

The `@RequestScoped` annotation designates that our managed bean will have a scope of request. Had we selected a different scope when creating the managed bean with the NetBeans wizard, it would have been annotated with the appropriate annotation corresponding to the selected scope. Session scoped managed beans are annotated with the `@SessionScoped` annotation. Application scoped managed beans are annotated with the `@ApplicationScoped` annotation. Managed beans with a scope of "none", are annotated with the `@NoneScoped` annotation. View scoped managed beans are annotated with the `@ViewScoped` annotation.

At this point, we need to modify our managed bean by adding properties that will hold the user-entered values.

Automatic Generation of Getter and Setter Methods

Netbeans can automatically generate getter and setter methods for our properties. We simply need to click the keyboard shortcut for "insert code", which defaults to *Alt+Insert* in Windows and Linux, then select **Getters and Setters**.

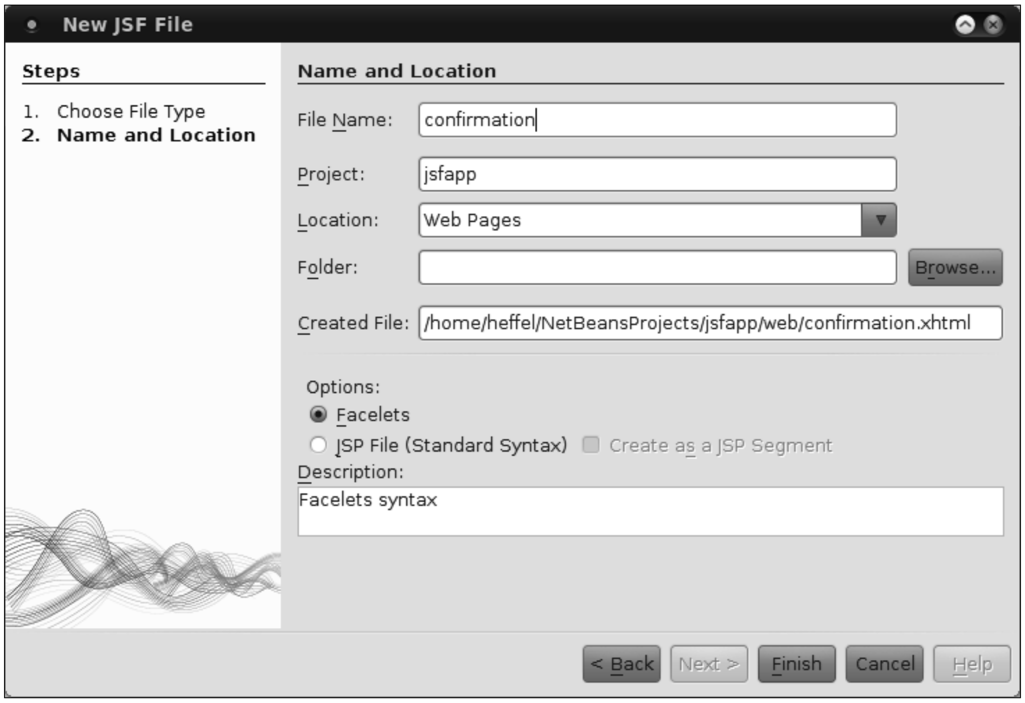
```
package com.ensode.jsf.managedbeans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class RegistrationBean {

    /** Creates a new instance of RegistrationBean */
    public RegistrationBean() {
    }
    private String salutation;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;

    //getters and setters omitted for brevity
}
```

Notice that the names of all of the bean's properties (instance variables) match the names we used in the page's value binding expressions. These names must match so that JSF knows how to map the bean's properties to the value binding expressions.

Implementing the confirmation page

Once our user fills out the data on the input page and submits the form, we want to show a confirmation page displaying the values that the user entered. Since we used value binding expressions on every input field on the input page, the corresponding fields on the managed bean will be populated with user-entered data. Therefore all we need to do in our confirmation page is display the data on the managed bean via a series of `<h:outputText>` JSF tags.

We can create the confirmation page via the **New JSF File** wizard.



File Name: confirmation

Project: jsfapp

Location: Web Pages

Folder:

Created File: /home/heffel/NetBeansProjects/jsfapp/web/confirmation.xhtml

Options:

[✓ Facelets]

JSP File (Standard Syntax) Create as a JSP Segment

Description:

Facelets syntax

We need to make sure the name of the new file matches the value of the action attribute in the command button of the input page (`confirmation.xhtml`) so that static navigation works properly.

After modifying the generated page to meet the requirements, it should look like this:

```
<?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:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Confirmation Page</title>
        <h:outputStylesheet library="css" name="styles.css"/>
    </h:head>
    <h:body>
        <h2>Confirmation Page</h2>
        <h:panelGrid columns="2"
                     columnClasses="rightalign-bold,normal">
```

```
                <h:outputText value="Salutation: "/>
                <h:outputText
                    value="#{registrationBean.salutation}" />
                <h:outputText value="First Name:"/>
                <h:outputText value="#{registrationBean.firstName}" />
                <h:outputText value="Last Name:"/>
                <h:outputText value="#{registrationBean.lastName}" />
                <h:outputText value="Age:"/>
                <h:outputText value="#{registrationBean.age}"/>
                <h:outputText value="Email Address:"/>
                <h:outputText value="#{registrationBean.email}" />
            </h:panelGrid>
        </h:body>
    </html>
```

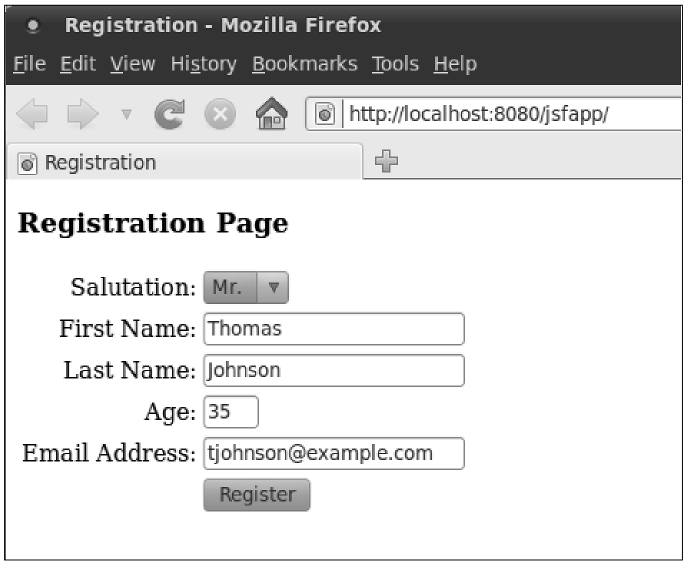
As we can see, our confirmation page is very simple. It consists of a series of `<h:outputText>` tags containing labels and value binding expressions bound to our managed bean's properties. The JSF `<h:outputText>` tag simply displays the value of the expression of its value attribute on the rendered page.

Executing our application

We are now ready to execute our JSF application. The easiest way to do so is to right-click on our project and click on **Run** in the resulting pop up menu

At this point GlassFish (or whatever application server we are using for our project) will start automatically, if it hadn't been started already, the default browser will open and it will automatically be directed to our page's URL.

After entering some data on the page, it should look something like the following screenshot:

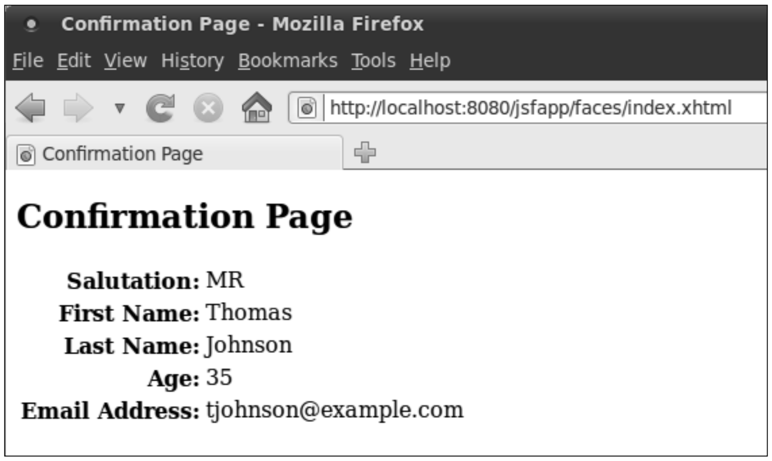

Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/
Registration
Register

Registration Page

Salutation: Mr.
First Name: Thomas
Last Name: Johnson
Age: 35
Email Address: tjohnson@example.com

When we click on the **Register** button, our `RegistrationBean` managed bean is populated with the values we entered into the page. Each property in the field will be populated according to the value binding expression in each input field.

At this point JSF navigation "kicks in", and we are taken to the confirmation page.


Confirmation Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/faces/index.xhtml
Confirmation Page

Confirmation Page

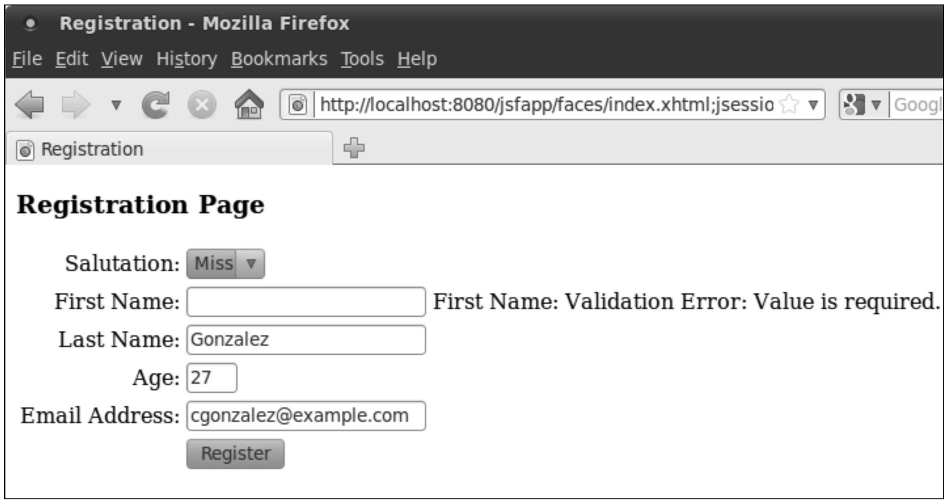
Salutation: MR
First Name: Thomas
Last Name: Johnson
Age: 35
Email Address: tjohnson@example.com

The values displayed in the confirmation page are taken from our managed bean, confirming that the bean's properties were populated correctly.

JSF validation

Earlier in this chapter we discussed how the `required` attribute for JSF input fields allows us to easily make input fields mandatory.

If a user attempts to submit a form with one or more required fields missing, an error message is automatically generated.

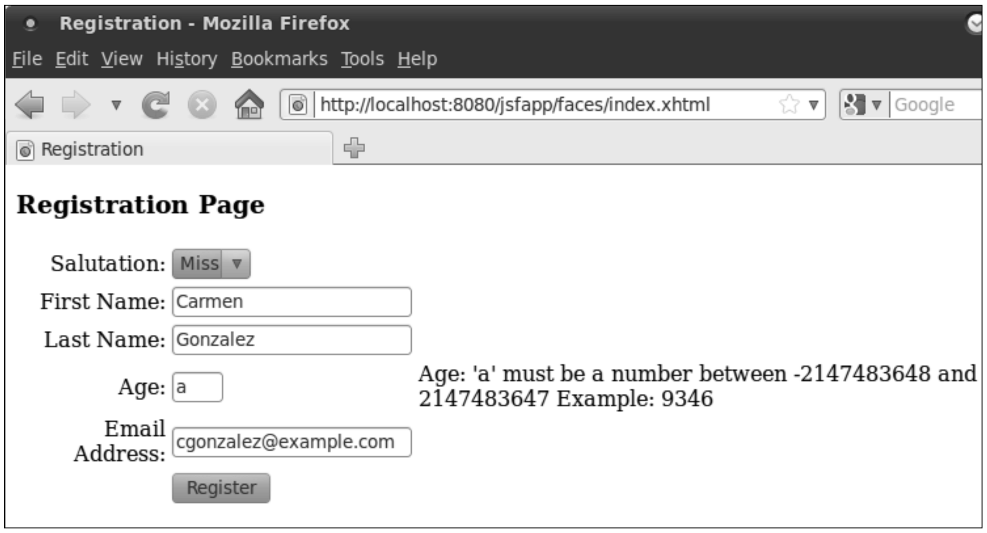

Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/faces/index.xhtml;jsessio
Registration
Register

Registration Page

Salutation:	Miss	
First Name:		First Name: Validation Error: Value is required.
Last Name:	Gonzalez	
Age:	27	
Email Address:	cgonzalez@example.com	

The error message is generated by the `<h:message>` tag corresponding to the invalid field. The string "First Name" in the error message corresponds to the value of the `label` attribute for the field, had we omitted the label attribute, the value of the field's `id` attribute would have been shown instead. As we can see, the `required` attribute makes it very easy to implement mandatory field functionality in our application.

Recall that the `age` field is bound to a property of type `Integer` in our managed bean. If a user enters a value that is not a valid integer into this field, a validation error is automatically generated.


Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/faces/index.xhtml
Google
Registration

Registration Page

Salutation: Miss
First Name: Carmen
Last Name: Gonzalez
Age: a
Email Address: cgonzalez@example.com

Age: 'a' must be a number between -2147483648 and 2147483647 Example: 9346

Of course, a negative age wouldn't make much sense. However, our application validates that user input is a valid Integer with essentially no effort on our part.

The email address input field of our page is bound to a property of type `String` in our managed bean. As such, there is no built in validation to make sure that the user enters a valid email address. In cases like this, we need to write our own custom JSF validator.

Custom JSF validators must implement the `javax.faces.validator.Validator` interface. This interface contains a single method named `validate()`, this method takes three parameters, an instance of `javax.faces.context.FacesContext`, an instance of `javax.faces.component.UIComponent` containing the JSF component we are validating, and an instance of `java.lang.Object` containing the user-entered value for the component. The following example illustrates a typical custom validator:

```
package com.ensode.jsf.validators;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator("emailValidator")
public class EmailValidator implements Validator {

    public void validate(FacesContext facesContext,
            UIComponent uIComponent, Object value) t
ValidatorException {
        Pattern pattern = Pattern.compile("\\w+@\\w+\\.\\w+");
        Matcher matcher = pattern.matcher(
               (CharSequence) value);
        HtmlInputText htmlInputText =
               (HtmlInputText) uIComponent;
        String label;

        if (htmlInputText.getLabel() == null ||
                htmlInputText.getLabel().trim().equals("")) {
            label = htmlInputText.getId();
        } else {
            label = htmlInputText.getLabel();
        }

        if (!matcher.matches()) {
            FacesMessage facesMessage =
                   new FacesMessage(label +
                   ": not a valid email address");

            throw new ValidatorException(facesMessage);
        }
    }
}
```

In our example, the `validate()` method does a regular expression match against the value of the JSF component we are validating. If the value matches the expression, validation succeeds, otherwise, validation fails and an instance of `javax.faces.validator.ValidatorException` is thrown.

The primary purpose of our custom validator is to illustrate how to write custom JSF validations, and not to create a foolproof email address validator. There may be a valid email address that doesn't validate using our validator.

The constructor of `ValidatorException` takes an instance of `javax.faces.application.FacesMessage` as a parameter. This object is used to display the error message on the page when validation fails. The message to display is passed as a `String` to the constructor of `FacesMessage`. In our example, if the `label` attribute of the component is not `null` nor empty, we use it as part of the error message, otherwise we use the value of the component's `id` attribute. This behavior follows the pattern established by standard JSF validators.

Our validator needs to be annotated with the `@FacesValidator` annotation. The value of its `value` attribute is the ID that will be used to reference our validator in our JSF pages.

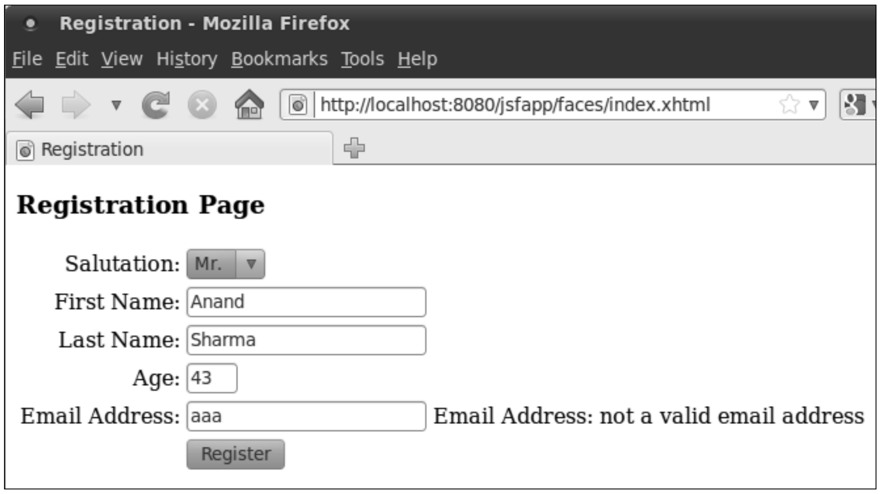
Once we are done implementing our validator, we are ready to use it in our pages.

In our particular case, we need to modify the email field to use our custom validator.

```
<h:inputText id="email" label="Email Address"
        required="true" value="#{registrationBean.email}">
    <f:validator validatorId="emailValidator"/>
</h:inputText>
```

All we need to do is nest a `<f:validator>` tag inside the input field we wish to have validated using our custom validator. The value of the `validatorId` attribute of `<f:validator>` must match the value of the value attribute in the `@FacesValidator` annotation in our validator.

At this point we are ready to test our custom validator.


Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jsfapp/faces/index.xhtml
Registration
Register

Registration Page

Salutation:	Mr.
First Name:	Anand
Last Name:	Sharma
Age:	43
Email Address:	aaa

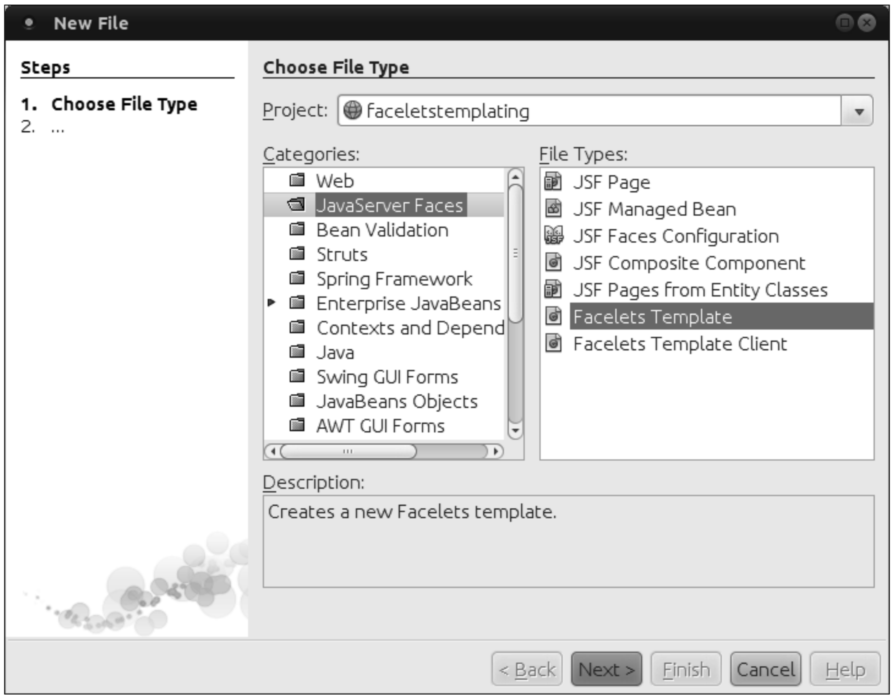
Email Address: not a valid email address

When entering an invalid email address into the email address input field and submitting the form, our custom validator logic was executed and the `String` we passed as a parameter to `FacesMessage` in our `validator()` method is shown as the error text by the `<h:message>` tag for the field.

Facelets templating

One advantage that Facelets has over JSP is its templating mechanism. Templates allow us to specify page layout in one place, then we can have template clients that use the layout defined in the template. Since most web applications have consistent layout across pages, using templates makes our applications much more maintainable, since changes to the layout need to be made in a single place. If at one point we need to change the layout for our pages (add a footer, or move a column from the left side of the page to the right side of the page, for example), we only need to change the template, and the change is reflected in all template clients.

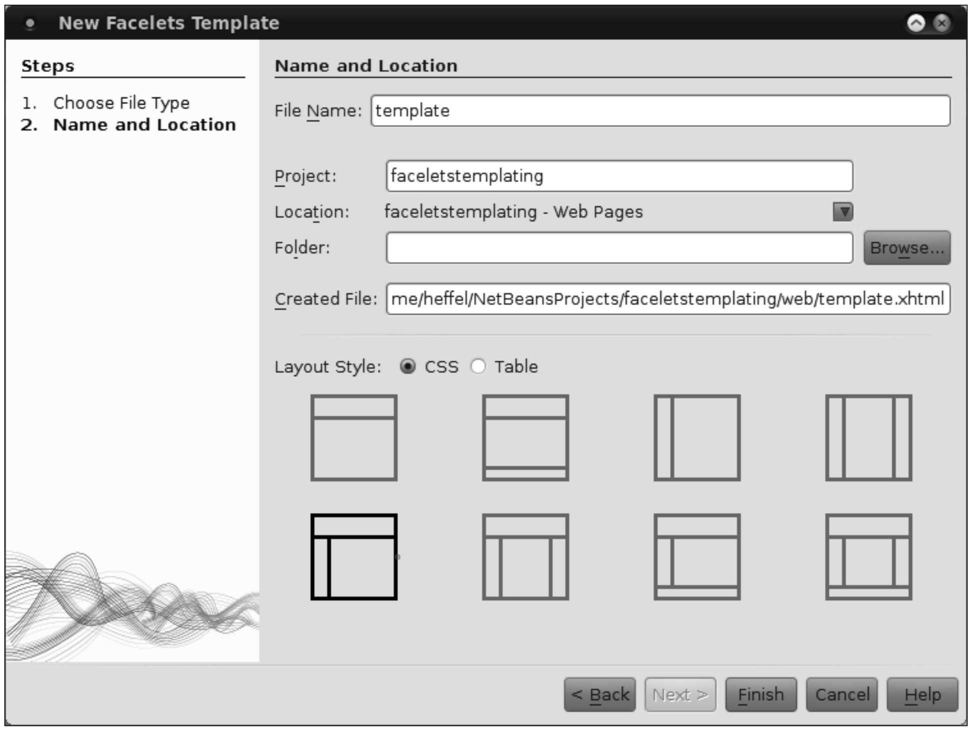
NetBeans provides very good support for facelets templating. It provides several templates "out of the box", using common web page layouts.



Project: faceletstemplating

Description: Creates a new Facelets template.

We can then select from one of several predefined templates to use as a base for our template or simply to use it "out of the box".



File Name: template
Project: faceletstemplating
Location: faceletstemplating - Web Pages
Folder:
Created File: me/heffel/NetBeansProjects/faceletstemplating/web/template.xhtml
Layout Style: [✓ CSS] Table

NetBeans gives us the option of using HTML tables or CSS for layout. For most modern web applications, CSS is the preferred approach. For our example we will pick a layout containing a header area, a single left column, and a main area.

After clicking on **Finish**, NetBeans automatically generates our template, along with the necessary CSS files.

The automatically generated template looks like this:

```
<?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://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

    <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>

    <h:body>
        <div id="top" class="top">
            <ui:insert name="top">Top</ui:insert>
        </div>
        <div>
            <div id="left">
                <ui:insert name="left">Left</ui:insert>
            </div>
            <div id="content" class="left_content">
                <ui:insert name="content">Content</ui:insert>
            </div>
        </div>
    </h:body>
</html>
```

As we can see, the template doesn't look much different from a regular Facelets file.

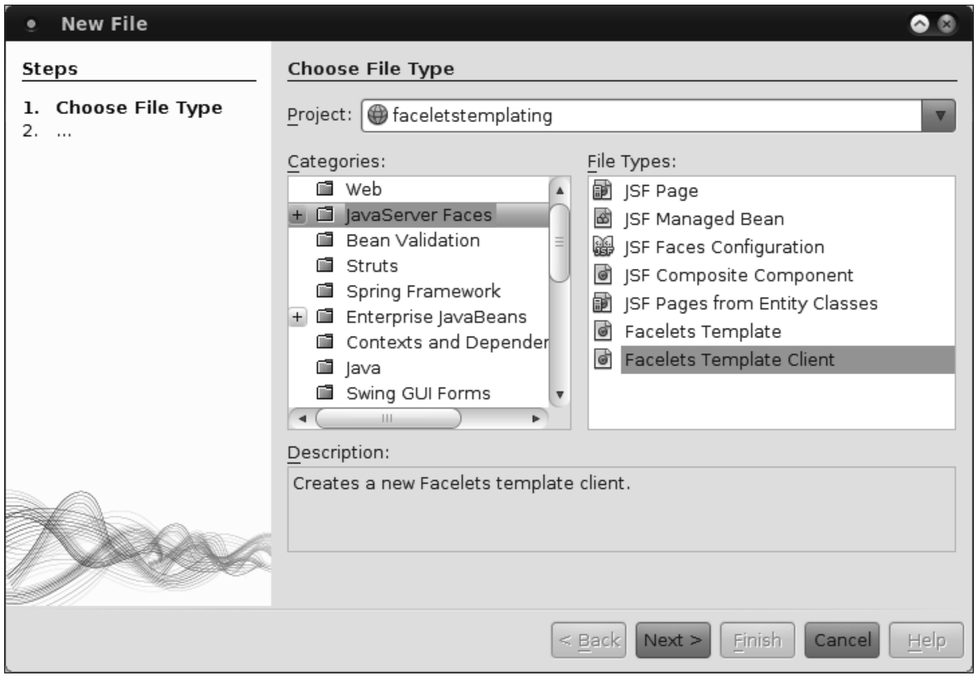
Adding a Facelets template to our project

We can add a Facelets template to our project simply by clicking on **File | New File**, then selecting the **JavaServer Faces** category and the **Facelets Template** file type.

Notice that the template uses the following namespace: `xmlns:ui="http://java.sun.com/jsf/facelets"`. This namespace allows us to use the `<ui:insert>` tag, the contents of this tag will be replaced by the content in a corresponding `<ui:define>` tag in template clients.

Using the template

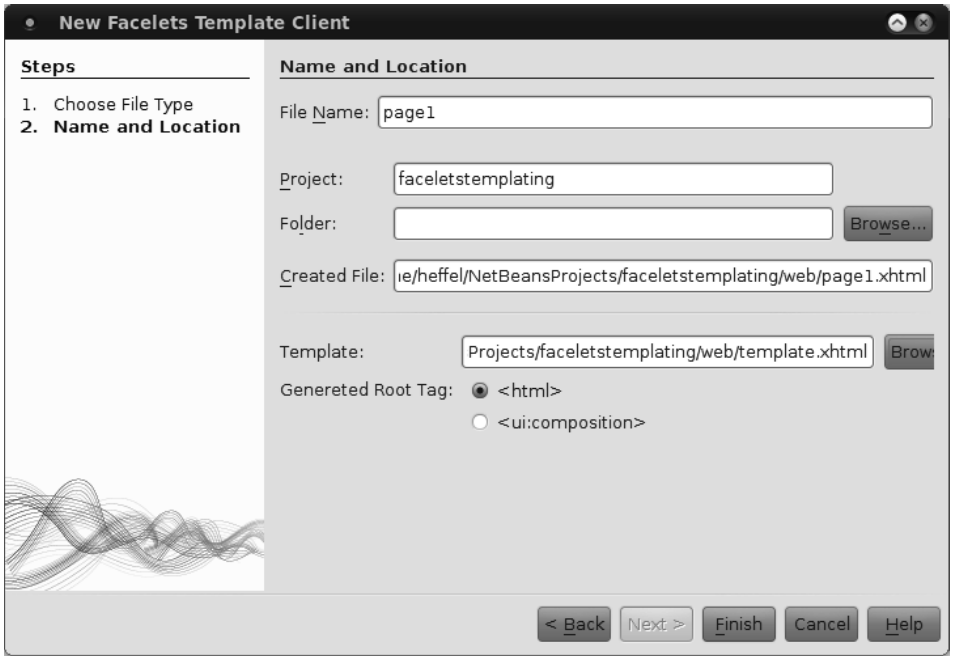
To use our template, we simply need to create a Facelets template client, which can be done by clicking on **File | New File**, selecting the **JavaServer Faces** category and the **Facelets Template Client** file type.



Project: faceletstemplating

Description: Creates a new Facelets template client.

After clicking on **Next >**, we need to enter a file name (or accept the default), and select the template that we will use for our template client.



File Name: page1

Project: faceletstemplating

Folder:

Created File: ıe/heffel/NetBeansProjects/faceletstemplating/web/page1.xhtml

Template: Projects/faceletstemplating/web/template.xhtml

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

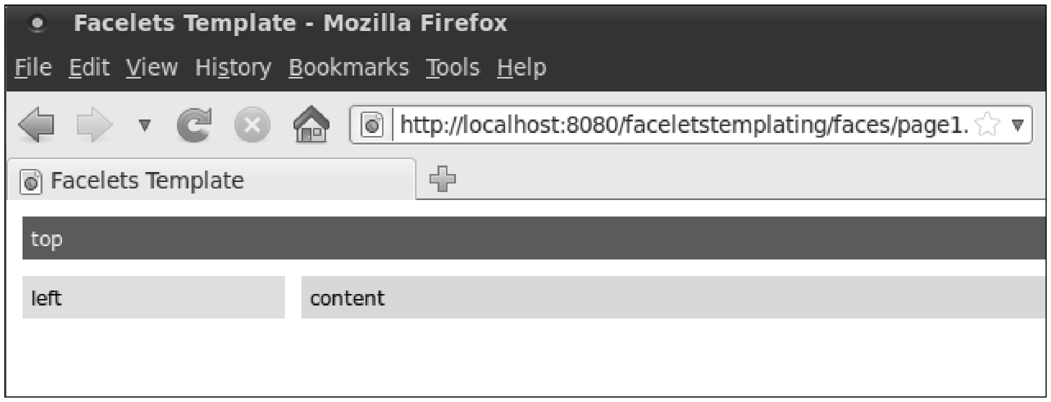
After clicking on **Finish**, our template client is created.

```
<?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://java.sun.com/jsf/facelets">

    <body>
        <ui:composition template="./template.xhtml">
            <ui:define name="top">
                top
            </ui:define>

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

            <ui:define name="content">
                content
            </ui:define>
        </ui:composition>

    </body>
</html>
```

As we can see, the template client also uses the `xmlns:ui="http://java.sun.com/jsf/facelets"` namespace. In a template client, the `<ui:composition>` tag must be the parent tag of any other tag belonging to this namespace. Any markup outside this tag will not be rendered; the template markup will be rendered instead.

The `<ui:define>` tag is used to insert markup into a corresponding `<ui:insert>` tag in the template. The value of the `name` attribute in `<ui:define>` must match the corresponding `<ui:insert>` tag in the template.

After deploying our application, we can see templating in action by pointing the browser to our template client URL.


Facelets Template - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/faceletstemplating/faces/page1.
Facelets Template
top
left
content

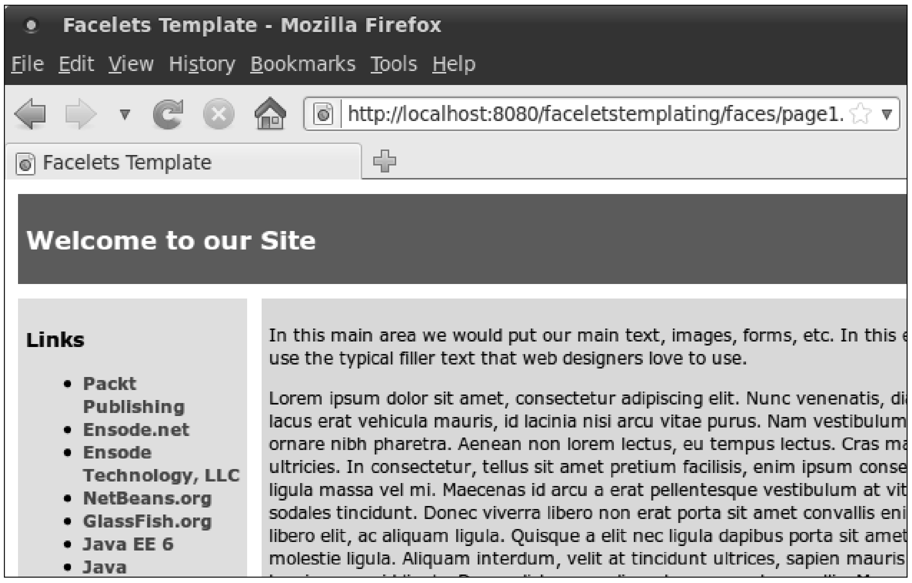
Notice that NetBeans generated a template that allows us to create a fairly elegant page with very little effort on our part. Of course, we should replace the markup in the `<ui:define>` tags to suit our needs.

Here is a modified version of our template, adding markup to be rendered in the corresponding places in the template:

```
<?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://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
    <body>
        <ui:composition template="./template.xhtml">
            <ui:define name="top">
                <h2>Welcome to our Site</h2>
            </ui:define>
            <ui:define name="left">
                <h3>Links</h3>
                <ul>
                    <li>
                        <h:outputLink value="http://www.packtpub.com">
                            <h:outputText value="Packt Publishing"/>
                        </h:outputLink>
                    </li>
                    <li>
                        <h:outputLink value="http://www.ensode.net">
                            <h:outputText value="Ensode.net"/>
                        </h:outputLink>
                    </li>
                    <li>
                        <h:outputLink value="http://www.ensode.com">
                           <h:outputText value="Ensode Technology,
                           LLC"/>
                        </h:outputLink>
                    </li>
                    <li>
                        <h:outputLink value="http://www.netbeans.org">
                            <h:outputText value="NetBeans.org"/>
                        </h:outputLink>
                    </li>
                    <li>
                        <h:outputLink value="http://www.glassfish.
                        org">
                            <h:outputText value="GlassFish.org"/>
                        </h:outputLink>
                    </li>
                    <li>
                        <h:outputLink
                            value="http://www.oracle.com/technetwork/
                            java/javaee/overview/index.html">
                            <h:outputText value="Java EE 6"/>
                        </h:outputLink>
                    </li>
                    <li><h:outputLink value="http://www.oracle.com/
                    technetwork/java/index.html">
                            <h:outputText value="Java"/>
                        </h:outputLink></li>
                </ul>
            </ui:define>
            <ui:define name="content">                <p>
                    In this main area we would put our main text,
                    images, forms, etc. In this example we will simply
                    use the typical filler text that web designers
                    love to use.
                </p>
                <p>
```

```
                    Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nunc venenatis, diam nec tempor dapibus, lacus erat
vehicula mauris, id lacinia nisi arcu vitae purus. Nam vestibulum
nisi non lacus luctus vel ornare nibh pharetra. Aenean non lorem
lectus, eu tempus lectus. Cras mattis nibh a mi pharetra ultricies.
In consectetur, tellus sit amet pretium facilisis, enim ipsum
consectetur magna, a mattis ligula massa vel mi. Maecenas id arcu a
erat pellentesque vestibulum at vitae nulla. Nullam eleifend sodales
tincidunt. Donec viverra libero non erat porta sit amet convallis enim
commodo. Cras eu libero elit, ac aliquam ligula. Quisque a elit nec
ligula dapibus porta sit amet a nulla. Nulla vitae molestie ligula.
Aliquam interdum, velit at tincidunt ultrices, sapien mauris sodales
mi, vel rutrum turpis neque id ligula. Donec dictum condimentum arcu
ut convallis. Maecenas blandit, ante eget tempor sollicitudin, ligula
eros venenatis justo, sed ullamcorper dui leo id nunc. Suspendisse
potenti. Ut vel mauris sem. Duis lacinia eros laoreet diam cursus nec
hendrerit tellus pellentesque.
                </p>
            </ui:define>
        </ui:composition>
    </body>
</html>
```

After making the above changes, our template client now renders as follows:


Facelets Template - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/faceletstemplating/faces/page1.
Facelets Template
Welcome to our Site
Links
Packt Publishing
Ensode.net
Ensode Technology, LLC
NetBeans.org
GlassFish.org
Java EE 6
Java

In this main area we would put our main text, images, forms, etc. In this
use the typical filler text that web designers love to use.

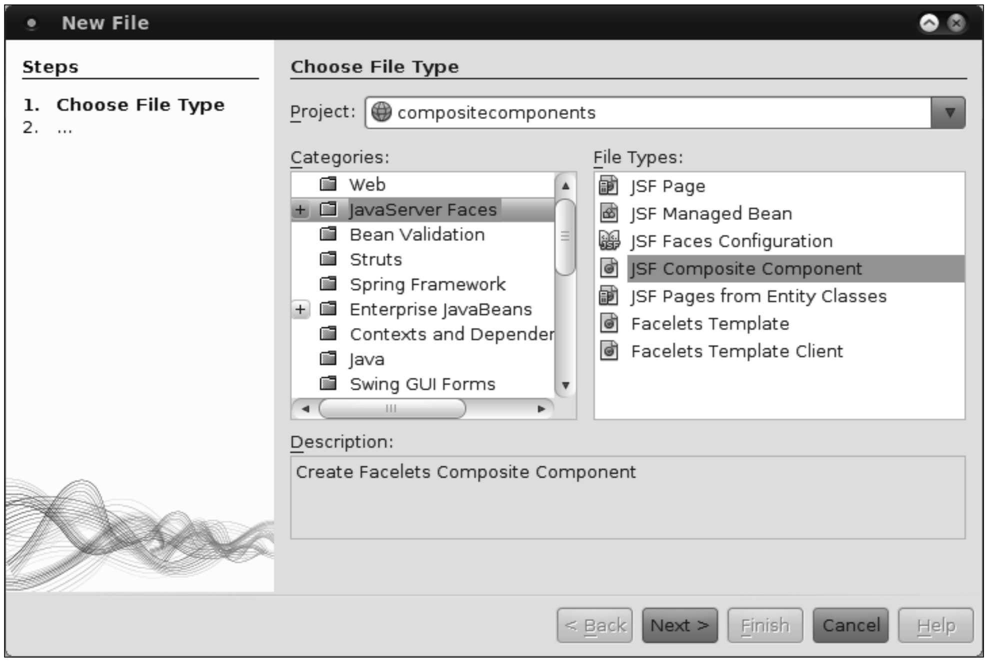
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc venenatis, di
lacus erat vehicula mauris, id lacinia nisi arcu vitae purus. Nam vestibulum
ornare nibh pharetra. Aenean non lorem lectus, eu tempus lectus. Cras ma
ultricies. In consectetur, tellus sit amet pretium facilisis, enim ipsum conse
ligula massa vel mi. Maecenas id arcu a erat pellentesque vestibulum at vit
sodales tincidunt. Donec viverra libero non erat porta sit amet convallis eni
libero elit, ac aliquam ligula. Quisque a elit nec ligula dapibus porta sit amet
molestie ligula. Aliquam interdum, velit at tincidunt ultrices, sapien mauris

As we can see, creating Facelets templates and template clients with NetBeans is a breeze.

Composite components

A very nice JSF 2.0 feature is the ability to easily write custom JSF components. With JSF 2, creating a custom component involves little more than creating the markup for it, with no Java code or configuration needed. Since custom components are typically composed of other JSF components, they are referred to as **composite components**.

We can generate a composite component by clicking on **File | New**, selecting the **JavaServer Faces** category and the **JSF Composite Component** file type.



Project: compositecomponents

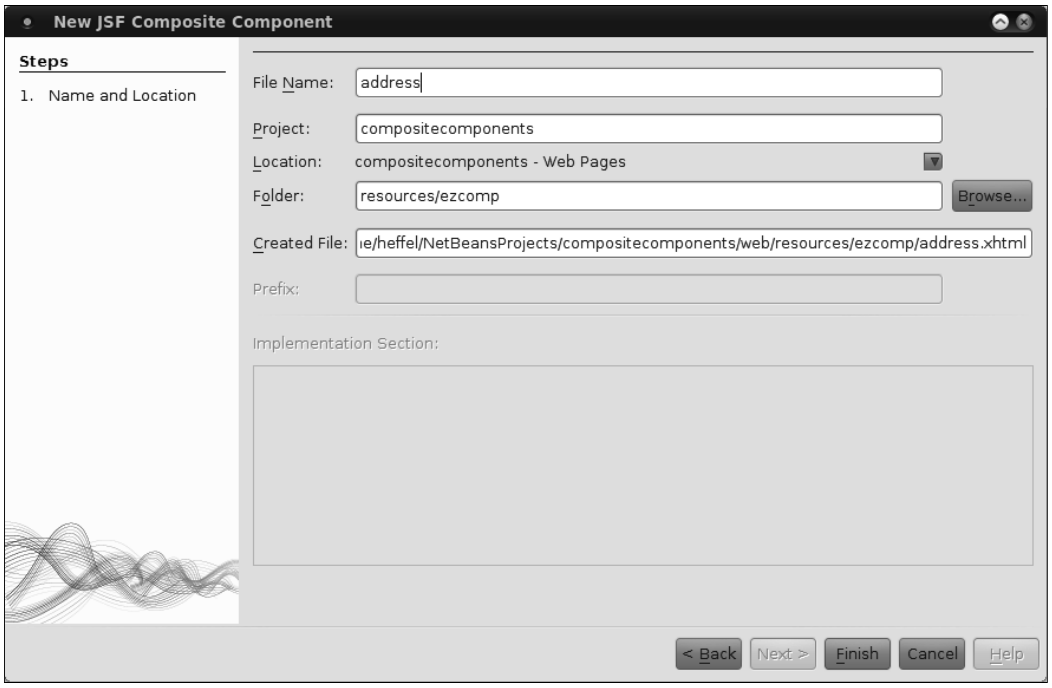
Categories:
- Web
- JavaServer Faces
- Bean Validation
- Struts
- Spring Framework
- Enterprise JavaBeans
- Contexts and Depender
- Java
- Swing GUI Forms

File Types:
- JSF Page
- JSF Managed Bean
- JSF Faces Configuration
- JSF Composite Component
- JSF Pages from Entity Classes
- Facelets Template
- Facelets Template Client

Description:

Create Facelets Composite Component

After clicking on **Next >**, we can specify the file name, project, and folder for our custom component.



File Name: address
Project: compositecomponents
Location: compositecomponents - Web Pages
Folder: resources/ezcomp
Created File: ıe/heffel/NetBeansProjects/compositecomponents/web/resources/ezcomp/address.xhtml
Prefix:
Implementation Section:

To take advantage of JSF 2.0's automatic resource handling and conventions, it is recommended that we don't change the folder where our custom component will be placed.

When we click on **Finish**, NetBeans generates an empty composite component that we can use as a base to create our own.

```
<?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:cc="http://java.sun.com/jsf/composite">
    <!-- INTERFACE -->
    <cc:interface>
    </cc:interface>
    <!-- IMPLEMENTATION -->
    <cc:implementation>
    </cc:implementation>
</html>
```

Every JSF 2.0 composite component contains two sections, an interface and an implementation.

The interface section must be enclosed inside a `<cc:interface>` tag. In the interface, we define any attributes that our component will have.

The implementation section contains the markup that will be rendered when we use our composite component.

In our example, we will develop a simple component which we can use to enter the addresses. That way, if we have to enter several addresses in an application, we can encapsulate the logic and/or display in our component. If later we need to change the address entry (to support international addresses, for example), we only need to change our component and all address entry forms in our application will be updated automatically.

After "filling in the blanks", our composite component now looks like this:

```
<?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:cc="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <!-- INTERFACE -->
    <cc:interface>
        <cc:attribute name="addrType"/>
        <cc:attribute name="managedBean" required="true"/>
    </cc:interface>

    <!-- IMPLEMENTATION -->
    <cc:implementation>
        <h:panelGrid columns="2">
            <f:facet name="header">
                <h:outputText value="#{cc.attrs.addrType} Address"/>
            </f:facet>
            <h:outputLabel for="line1" value="Line 1"/>
            <h:inputText id="line1" value="#{cc.attrs.managedBean.
            line1}"/>
            <h:outputLabel for="line2" value="Line 2"/>
            <h:inputText id="line2" value="#{cc.attrs.managedBean.
            line2}"/>
            <h:outputLabel for="city" value="City"/>
            <h:inputText id="city" value="#{cc.attrs.managedBean.
            city}"/>
```

```
                <h:outputLabel for="state" value="state"/>
                <h:inputText id="state" value="#{cc.attrs.managedBean.
                state}" size="2" maxlength="2"/>
                <h:outputLabel for="zip" value="Zip"/>
                <h:inputText id="zip" value="#{cc.attrs.managedBean.zip}"
                size="5" maxlength="5"/>
            </h:panelGrid>
        </cc:implementation>
    </html>
```

We specify attributes for our component via the `<cc:attribute>` tag. This tag has a `name` attribute used to specify the attribute name, and an optional `required` attribute that we can use to specify if the attribute is required.

The body of the `<cc:implementation>` tag looks almost like plain old JSF markup, with one exception, by convention, we can access the tag's attributes by using the `#{cc.attrs.ATTRIBUTE_NAME}` expression to access the attributes we defined in the component's interface section. Notice that the `managedBean` attribute of our component must resolve to a JSF managed bean. Pages using our component must use a JSF expression resolving to a managed bean as the value of this attribute. We can access the attributes of this managed bean by simply using the familiar `.property` notation we have used before, the only difference here is that instead of using a managed bean name in the expression, we must use the attribute name as defined in the interface section.

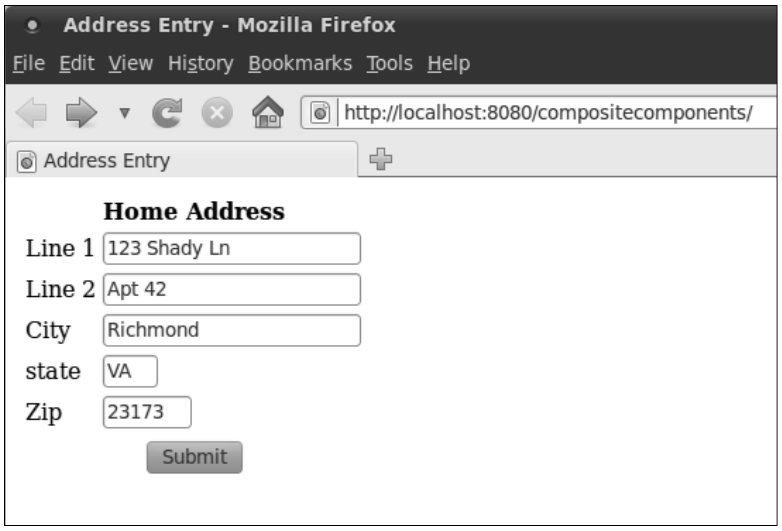
Now we have a simple but complete composite component, using it in our pages is very simple.

```
<?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:h="http://java.sun.com/jsf/html"
      xmlns:ezcomp="http://java.sun.com/jsf/composite/ezcomp">
    <h:head>
        <title>Address Entry</title>
    </h:head>
    <h:body>
        <h:form>
            <h:panelGrid columns="1">
                <ezcomp:address managedBean="#{addressBean}"
                                             addrType="Home"/>
                <h:commandButton value="Submit" action="confirmation"
                                 style="display: block; margin: 0
                                 auto;"/>
            </h:panelGrid>
        </h:form>
    </h:body>
</html>
```

By convention, the namespace for our custom components will always be `xmlns:ezcomp="http://java.sun.com/jsf/composite/ezcomp"`. This is why it is important not to override the default folder where our component will be placed, as doing so breaks this convention. NetBeans provides code completion for our custom composite components, just like it does for standard components.

In our application, we created a simple managed bean named addressBean. It is a simple managed bean with a few properties and corresponding getters and setters, therefore it is not shown (it is part of this chapter's code download). We use this bean as the value of the `managedBean` attribute of our component. We also used an `addressType` of "Home", this value will be rendered as a header for our address input component.

After deploying and running our application, we can see our component in action:


Address Entry - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/compositecomponents/
Address Entry
Submit

Home Address

Line 1 123 Shady Ln

Line 2 Apt 42

City Richmond

state VA

Zip 23173

As we can see, creating JSF 2.0 composite components with NetBeans is a breeze.

Summary

In this chapter we saw how NetBeans can help us easily create new JSF projects by automatically adding all required libraries.

We saw how we can quickly create JSF pages by taking advantage of NetBeans' code completion.

Additionally, we saw how we can significantly save time and effort by allowing NetBeans to generate JSF 2.0 templates, including generating the necessary CSS to easily create fairly elegant pages.

Finally, we saw how NetBeans can help us develop JSF 2.0 custom components.