JAVASERVER FACES

It is a server side component based user interface framework. It is used to develop web applications. It provides a well-defined programming model and consists of rich API and tag libraries. The latest version JSF 2 uses Facelets as its default templating system. It is written in Java.

The JSF API provides components (inputText, commandButton etc) and helps to manage their states. It also provides server-side validation, data conversion, defining page navigation, provides extensibility, supports for internationalization, accessibility etc.

The JSF Tag libraries are used to add components on the web pages and connect components with objects on the server. It also contains tag handlers that implements the component tag.

With the help of these features and tools, you can easily and effortlessly create server-side user interface.

Java Server Faces Versions History

Versions	Release date	Description
Jsf 2.3	Expected in 2017	It may includes major features: bean validation for complete classes, push communication using enhanced integration with cdi.
Jsf 2.2	21-05-2013	It has introduced new concepts like stateless views, page flow and the ability to create portable resource contracts.
Jsf 2.1	22-11-2010	It was a maintenance release 2 of jsf 2.0. only a very minor number of specification changes.
Jsf 2.0	01-07-2009	It was major release for ease of use, enhanced functionality, and performance. coincides with java ee 6.
Jsf 1.2	11-05-2006	It has many improvements to core systems and apis. coincides with Java ee 5. initial adoption into java ee.
Jsf 1.1	27-05-2004	It was a bug-fix release. no specification changes.
Jsf 1.0	11-03-2004	It was a initial specification released.

Benefits of JavaServer Faces

1) It provides clean and clear separation between behavior and presentation of web application. You can write business logic and user interface separately.
2) JavaServer Faces APIs are layered directly on top of the Servlet API. Which enables several various application use cases, such as using different presentation technologies, creating your own custom components directly from the component classes.
3) Including of Facelets technology in JavaServer Faces 2.0, provides massive advantages to it. Facelets is now the preferred presentation technology for building JavaServer Faces based web applications.

PREREQUISITES

Java: You must have Java 7 or higher version.

Java IDE: We can use NetBean IDE 8.2, Eclipse, Java IDEs.

Server: Tomcat Server, Glass Fish, etc.

JSF 2.2 Library: Latest JavaServer Faces libraries are automatically installed with the IDE. So, you don't need to install it manually.

JSF FEATURES

Latest version of JSF 2.2 provides the following features.

- **Component Based Framework:** JSF is a server side component-based framework. It provides inbuilt components to build web application. You can use HTML5, Facelets tags to create web pages.
- **Facelets Technology :** Facelets is an open source Web template system. It is a default view handler technology for JavaServer Faces (JSF). The language requires valid input XML documents to work. Facelets supports all of the JSF UI components and focuses completely on building the view for a JSF application.
- **Expression Language :** Expression Language provides an important mechanism for creating the user interface (web pages) to communicate with the application logic (managed beans). The EL represents a union of the expression languages offered by JavaServer Faces technology.
- **HTML 5:** HTML5 is the new standard for writing web pages. JavaServer Faces version 2.2 offers an easy way for including new attributes of HTML 5 to JSF components and provides HTML5 friendly markup.
- **Ease and Rapid web Development:** JSF provides rich set of inbuilt tools and libraries so that you can easily and rapidly develop we application.
- **Support Internationalization:** JSF supports internationalization for creating World Class web application. You can create applications in the different-different languages. With the help of JSF you can make the application adaptable to various languages and regions.
- **Bean Annotations:** JSF provides annotations facility in which you can perform validation related tasks in Managed Bean. It is good because you can validate your data in bean rather than in HTML validation.
- **Exception Handling:** JSF provide default Exception handling so you can develop exception and bug free web application.
- **Templating:** Introducing template in new version of JSF provides reusablity of components. In JSF application, you can create new template, reuse template and treat it as component for application.
- **AJAX Support:** JSF provides inbuilt AJAX support. So, you can render application request to server side without refreshing the web page. JSF also support partial rendering by using AJAX.
- **Security:** JSF provides implicit protection against this when state is saved on the server and no stateless views are used, since a post-back must then contain a valid javax.faces.ViewState hidden parameter. Contrary to earlier versions, this value seems sufficiently random in modern JSF implementations. Note that stateless views and saving state on the client does not have this implicit protection.

JSF - ARCHITECTURE

JSF technology is a framework for developing, building server-side User Interface Components and using them in a web application. JSF technology is based on the Model View Controller (MVC) architecture for separating logic from presentation.

What is MVC Design Pattern?

MVC design pattern designs an application using three separate modules –

- **Model:** Carries Data and login
- **View:** Shows User Interface
- **Controller:** Handles processing of an application.

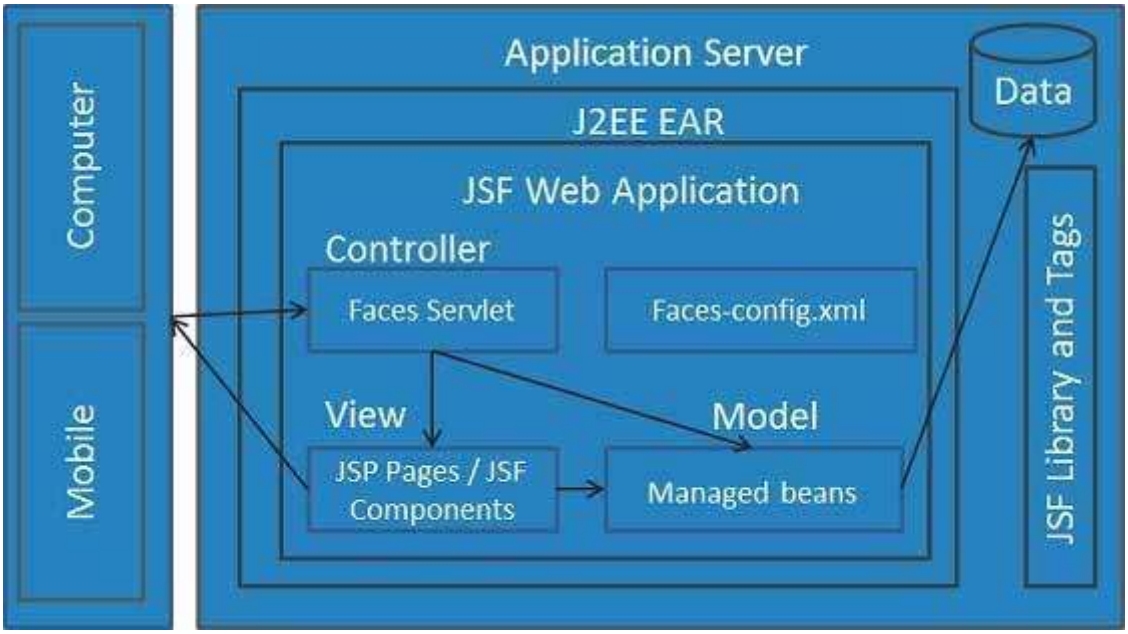
The purpose of MVC design pattern is to separate model and presentation enabling developers to focus on their core skills and collaborate more clearly.

Web designers have to concentrate only on view layer rather than model and controller layer. Developers can change the code for model and typically need not change view layer. Controllers are used to process user actions. In this process, layer model and views may be changed.

JSF ARCHITECTURE

JSF application is similar to any other Java technology-based web application; it runs in a Java servlet container, and contains -

- JavaBeans components as models containing application-specific functionality and data
- A custom tag library for representing event handlers and validators
- A custom tag library for rendering UI components
- UI components represented as stateful objects on the server
- Server-side helper classes
- Validators, event handlers, and navigation handlers
- Application configuration resource file for configuring application resources



There are controllers which can be used to perform user actions. UI can be created by web page authors and the business logic can be utilized by managed beans.

JSF provides several mechanisms for rendering an individual component. It is upto the web page designer to pick the desired representation, and the application developer doesn't need to know which mechanism was used to render a JSF UI component.

JAVASERVER FACES LIFECYCLE

JavaServer Faces application framework manages lifecycle phases automatically for simple applications and also allows you to manage that manually. The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page.

The JSF lifecycle is divided into two main phases:

1. Execute Phase
2. Render Phase

1) Execute Phase

In execute phase, when first request is made, application view is built or restored. For other subsequent requests other actions are performed like request parameter values are applied, conversions and validations are performed for component values, managed beans are updated with component values and application logic is invoked.

The execute phase is further divided into following subphases.

1. **Restore View Phase:** When a client requests for a JavaServer Faces page, the JavaServer Faces implementation begins the restore view phase. In this phase, JSF builds the view of the requested page, wires event handlers and validators to components in the view and saves the view in the FacesContext instance.
   - If the request for the page is a postback, a view corresponding to this page already exists in the FacesContext instance. During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.
2. **Apply Request Values Phase:** In this phase, component tree is restored during a postback request. Component tree is a collection of form elements.Each component in the tree extracts its new value from the request parameters by using its decode (processDecodes()) method. After that value is stored locally on each component.
   - If any decode methods or event listeners have called the renderResponse method on the current FacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.
   - If any events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.
   - If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete() method.
   - If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.
3. **Process Validations Phase :** In this phase, the JavaServer Faces processes all validators registered on the components by using its validate () method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. The JavaServer Faces also completes conversions for input components that do not have the immediate attribute set to true.
   - If any validate methods or event listeners have called the renderResponse method on the current FacesContext, the JavaServer Faces implementation skips to the Render Response phase.
   - If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.
   - If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.
   - If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.
4. **Update Model Values Phase:** After ensuring that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components' local values. The JavaServer Faces implementation updates only the bean properties pointed at by an input component's value attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed.
   - If any updateModels methods or any listeners have called the renderResponse() method on the current FacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.
   - If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete() method.

- If any events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.
- If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.

5. **Invoke Application Phase:** In this phase, JSF handles application-level events, such as submitting a form or linking to another page.
   - Now, if the application needs to redirect to a different web application resource or generate a response that does not contain any JSF components, it can call the FacesContext.responseComplete() method.
   - After that, the JavaServer Faces implementation transfers control to the Render Response phase.
6. **Render Response Phase:** This is last phase of JSF life cycle. In this phase, JSF builds the view and delegates authority to the appropriate resource for rendering the pages.
   - If this is an initial request, the components that are represented on the page will be added to the component tree.
   - If this is not an initial request, the components are already added to the tree and need not to be added again.
   - If the request is a postback and errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered again during this phase.
   - If the pages contain h:message or h:messages tags, any queued error messages are displayed on the page.
   - After rendering the content of the view, the state of the response is saved so that subsequent requests can access it. The saved state is available to the Restore View phase.

2) Render

In this phase, the requested view is rendered as a response to the client browser. View rendering is a process in which output is generated as HTML or XHTML. So, user can see it at the browser.

The following steps are taken during the render process.

- Application is compiled, when a client makes an initial request for the index.xhtml web page.
- Application executes after compilation and a new component tree is constructed for the application and placed in a FacesContext.
- The component tree is populated with the component and the managed bean property associated with it, represented by the EL expression.
- Based on the component tree. A new view is built.
- The view is rendered to the requesting client as a response.
- The component tree is destroyed automatically.
- On subsequent requests, the component tree is rebuilt, and the saved state is applied.

TRACING PHASES

For better understandability, we can use a phase listener to trace the JSF life-cycle phases and to see what happens in which phase. A phase listener class must implement PhaseListener interface which has three methods. Here is a simple phase listener class:

```
import javax.faces.event.*;
public class MyListener implements PhaseListener
{
public void beforePhase(PhaseEvent pe)
```

```
{
System.out.println("Start phase : "+pe.getPhaseId());
}
public void afterPhase(PhaseEvent pe)
{
System.out.println("End phase : "+pe.getPhaseId());
}
public PhaseId getPhaseId()
{
return PhaseId.ANY_PHASE;
}
}
```

This phase listener is interested in all the phases and hence the **getPhaseId**() method returns the identifier **ANY_PHASE**. The **beforePhase**() and **afterPhase**() methods are called just before and after a phase is started and completed. To active the phase listener to a JSF application, it must be registered in faces-**config.xml** as follows:

```
<lifecycle>
<phase-listener>phaseListeners.MyListener</phase-listener>
</lifecycle>
```

For an initial request, it produces the following in the system output:

```
Start phase : RESTORE_VIEW 1
End phase : RESTORE_VIEW 1
Start phase : RENDER_RESPONSE 6
End phase : RENDER_RESPONSE 6
```

For a postback request, it produces the following output:

```
Start phase : RESTORE_VIEW 1
End phase : RESTORE_VIEW 1
Start phase : APPLY_REQUEST_VALUES 2
End phase : APPLY_REQUEST_VALUES 2
Start phase : PROCESS_VALIDATIONS 3
End phase : PROCESS_VALIDATIONS 3
Start phase : UPDATE_MODEL_VALUES 4
End phase : UPDATE_MODEL_VALUES 4
Start phase : INVOKE_APPLICATION 5
End phase : INVOKE_APPLICATION 5
Start phase : RENDER_RESPONSE 6
End phase : RENDER_RESPONSE 6
```

JSF MANAGED BEAN

It is a pure Java class which contains set of properties and set of getter, setter methods.

Following are the common functions that managed bean methods perform:

- Validating a component's data
- Handling an event fired by a component
- Performing processing to determine the next page to which the application must navigate

It also works as model for JFS Framework.

JSF Managed Bean Example

```
1. public class User {
2. private String name;
3. public String getName() {
4. return name;
5. }
6. public void setName(String name) {
7. this.name = name;
8. }
9. }
```

You can use this bean by the following ways.

1. By configuring into XML file.
2. By using annotations.

Configuring Managed Bean into XML file

```
1. <managed-bean>
2. <managed-bean-name>user</managed-bean-name>
3. <managed-bean-class>User</managed-bean-class>
4. <managed-bean-scope>session</managed-bean-scope>
5. </managed-bean>
```

This is an older approach to configure bean into xml file. In this approach, we have to create a xml file named faces-config.xml. JSF provides <managec-bean> tag to configure the bean.

In the above example, we are listing bean-name, bean-class and bean-scope. So, it can be accessible in the project.

Configuring Managed Bean using Annotations

```
1. import javax.faces.bean.ManagedBean;
2. import javax.faces.bean.RequestScoped;
3. @ManagedBean    // Using ManagedBean annotation
4. @RequestScoped  // Using Scope annotation
5. public class User {
6.     private String name;
7.     public String getName() {
8.         return name;
9.     }
10.    public void setName(String name) {
11.     this.name = name;
12. }
13. }
```

The @ManagedBean annotation in a class automatically registers that class as a resource with the JavaServer Faces. Such a registered managed bean does not need managed-bean configuration entries in the application configuration resource file.

This is an alternative to the application configuration resource file approach and reduce the task of configuring managed beans.

The @RequestScoped annotation is used to provide scope for ManagedBean. You can use annotations to define the scope in which the bean will be stored.

You can use following scopes for a bean class:

- **Application (@ApplicationScoped):** Application scope persists across all users? interactions with a web application.
- **Session (@SessionScoped):** Session scope persists across multiple HTTP requests in a web application.
- **View (@ViewScoped):** View scope persists during a user?s interaction with a single page (view) of a web application.
- **Request (@RequestScoped):** Request scope persists during a single HTTP request in a web application.
- **None (@NoneScoped):** Indicates a scope is not defined for the application.
- **Custom (@CustomScoped):** A user-defined, nonstandard scope. Its value must be configured as a java.util.Map. Custom scopes are used infrequently.

Eager Managed Bean

Managed bean is lazy by default. It means, bean is instantiated only when a request is made from the application.

You can force a bean to be instantiated and placed in the application scope as soon as the application is started. You need to set eager attribute of the managed bean to true as shown in the following example:

1. @ManagedBean(eager=true)

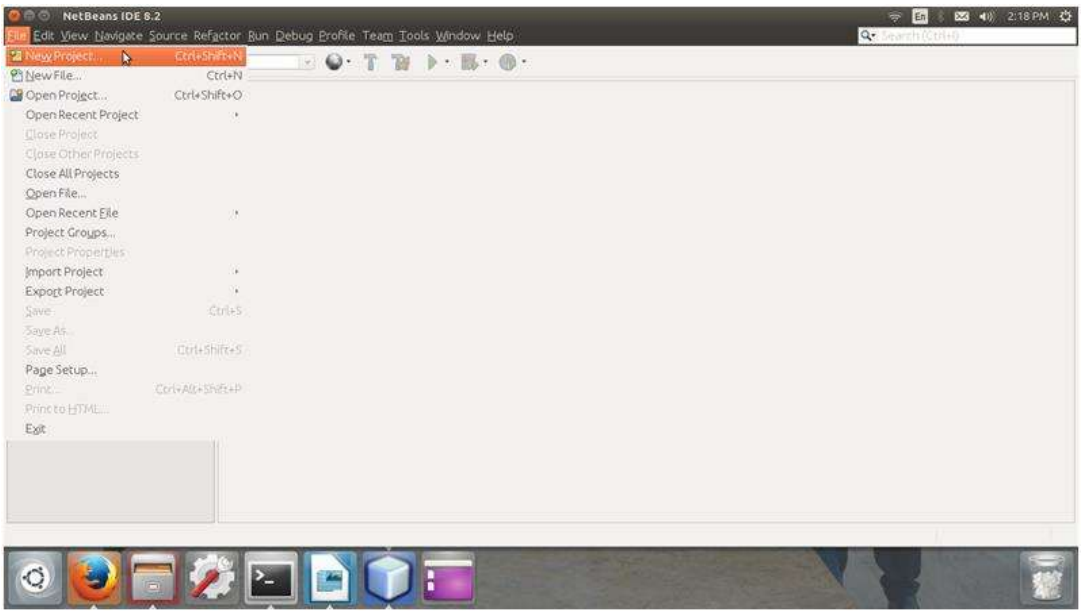
A SIMPLE JAVASERVER FACES APPLICATION

To create a JSF application, we are using NetBeans IDE 8.2. You can also refer to other Java IDEs.

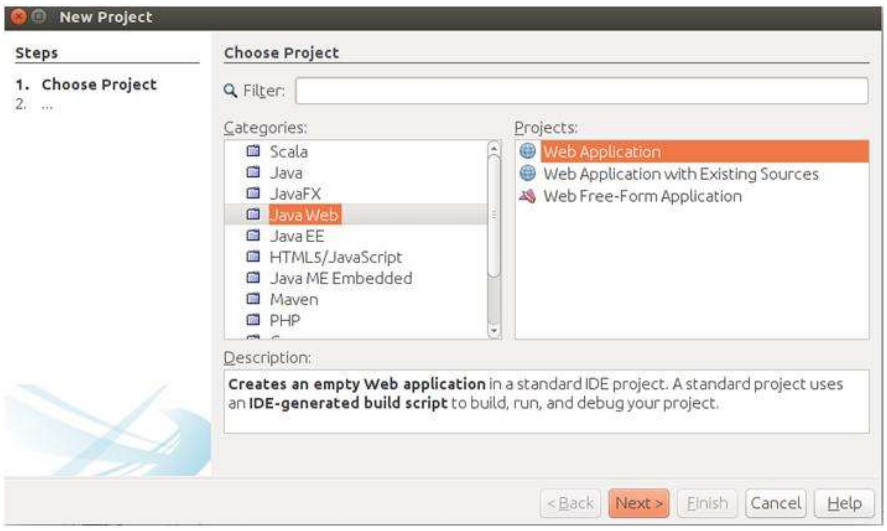
Here, we are creating a project after that we will run to test it's configuration settings. So, let's create a new project fist.

Step 1: Create a New Project

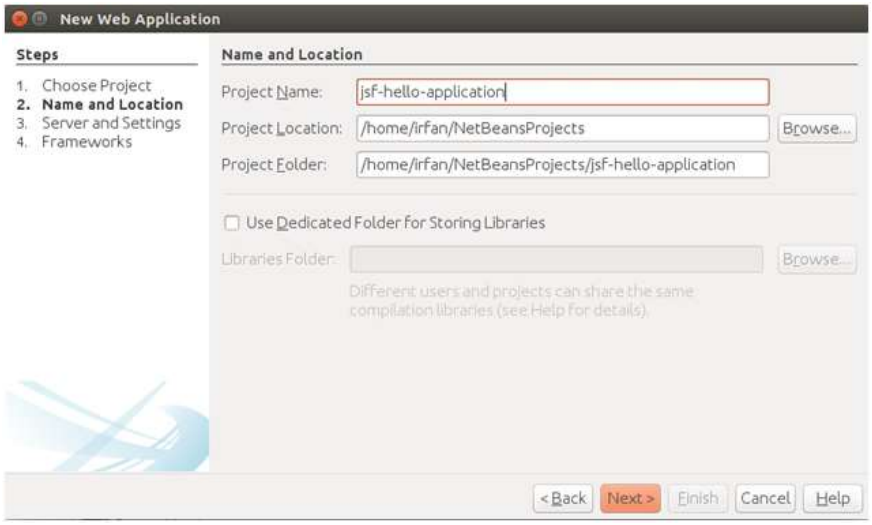
Go to file menu and select new Project.



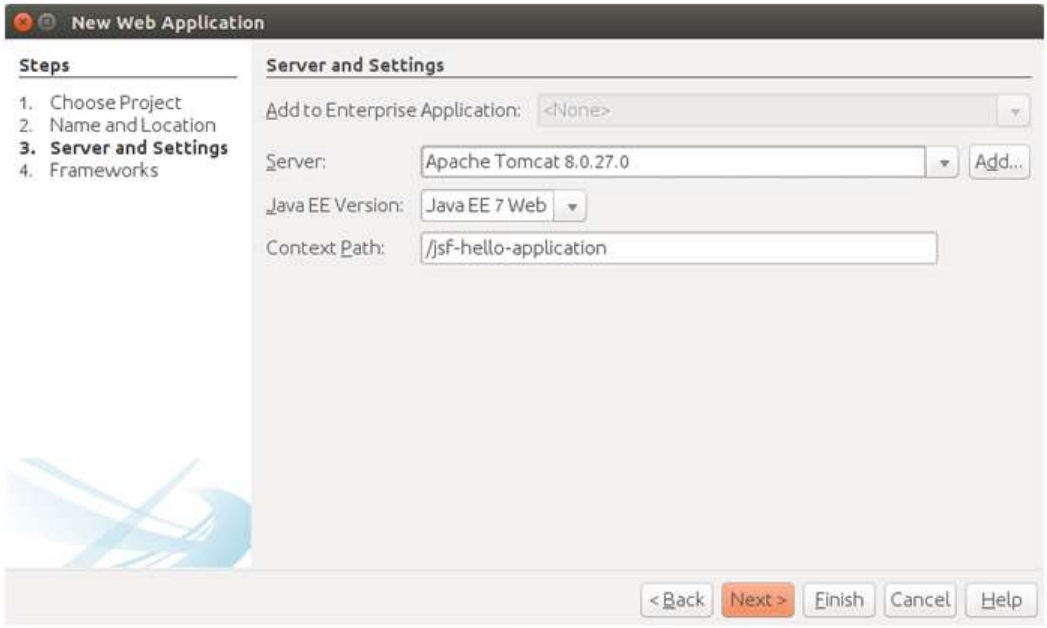
Select Category Java Web and Project Web Application.



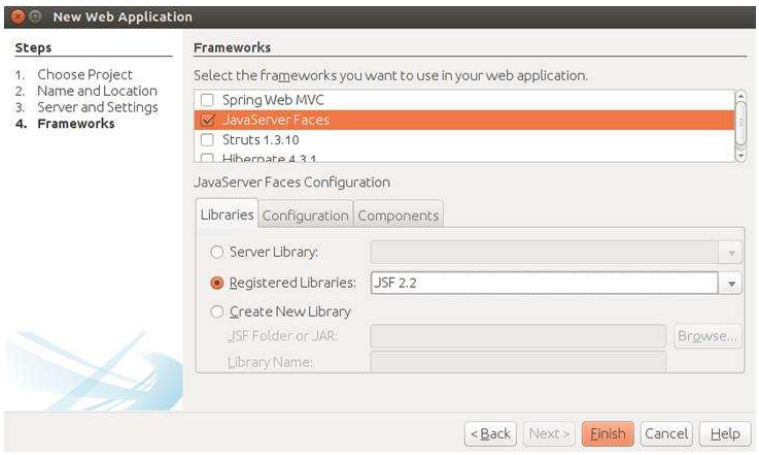
Enter project name.



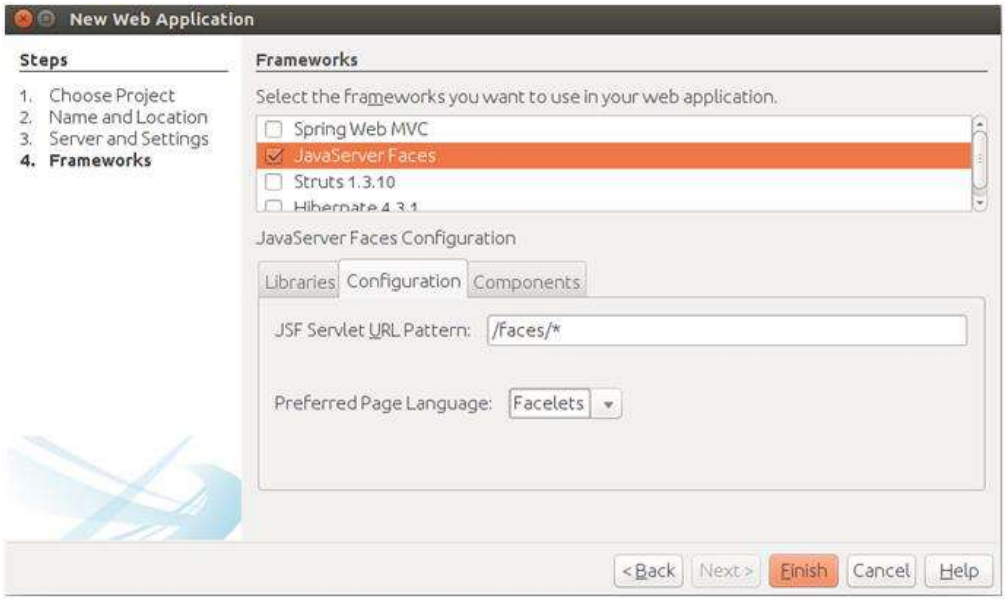
Select Server and Java EE Version.



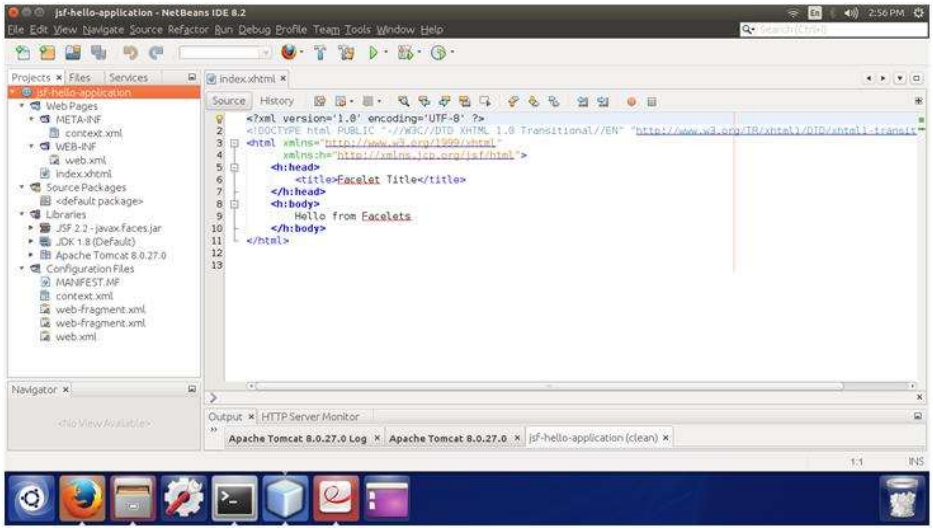
Select JSF Framework



Select Preferred Page Language: Earlier versions of JSF framework are default to JSP for presentation pages. Now, in latest version 2.0 and later JSF has included powerful tool "Facelets". So, here we have selected page language as facelets. We will talk about facelets in more details in next chapter.



Index.xhtml Page: After finishing, IDE creates a JSF project for you with a default index.xhtml file. Xhtml is a extension of html and used to create facelets page.



Run: Now, you can run your application by selecting run option after right click on the project. It will produce a default message "Hello from Facelets".

We have created JSF project successfully. This project includes following files:

1. **index.xhtml:** inside the Web Pages directory
2. **web.xml:** inside the WEB-INF directory

Whenever we run the project, it renders index.xhtml as output. Now, we will create an application which contains two web pages, one bean class and a configuration file.

It requires the following steps in order to develop new application:

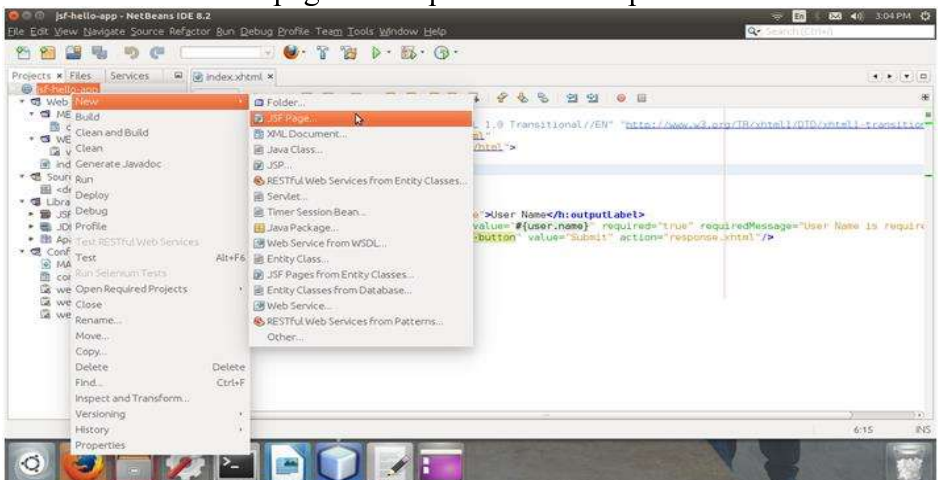
1. Creating user interface
2. Creating managed beans
3. Configuring and managing FacesServlet

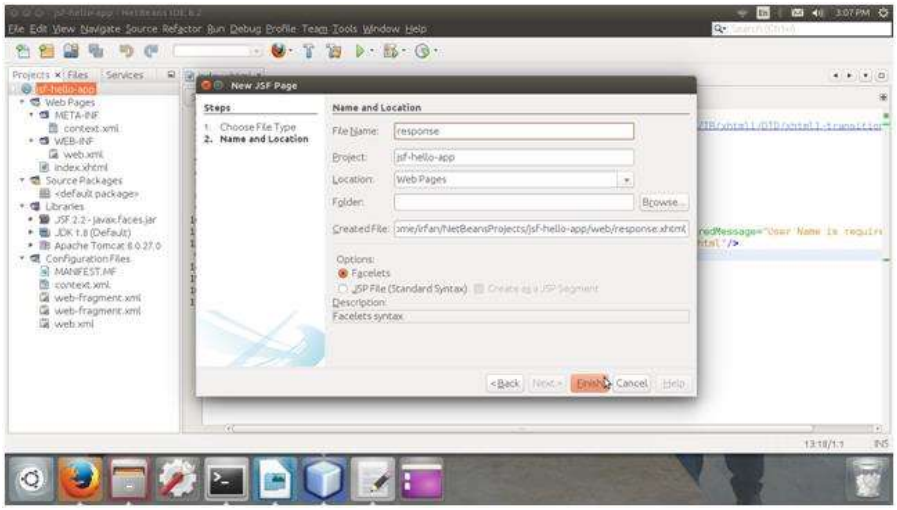
1) Create User Interface

We will use default page index.xhtml to render input web page. Modify your index.xhtml source code as the given below.

1. // index.xhtml
2. <?xml version='1.0' encoding='UTF-8' ?>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5. xmlns:h="http://xmlns.jcp.org/jsf/html">
6. <h:head>
7. <title>User Form</title>
8. </h:head>
9. <h:body>
10. <h:form>
11. <h:outputLabel for="username">User Name</h:outputLabel>
12. <h:inputText id="username" value="#{user.name}" required="true" requiredMessage="User Name i s required" />
13. <h:commandButton id="submit-button" value="Submit" action="response.xhtml"/>
14. </h:form>
15. </h:body>
16. </html>

Create a second web page which produce the output.





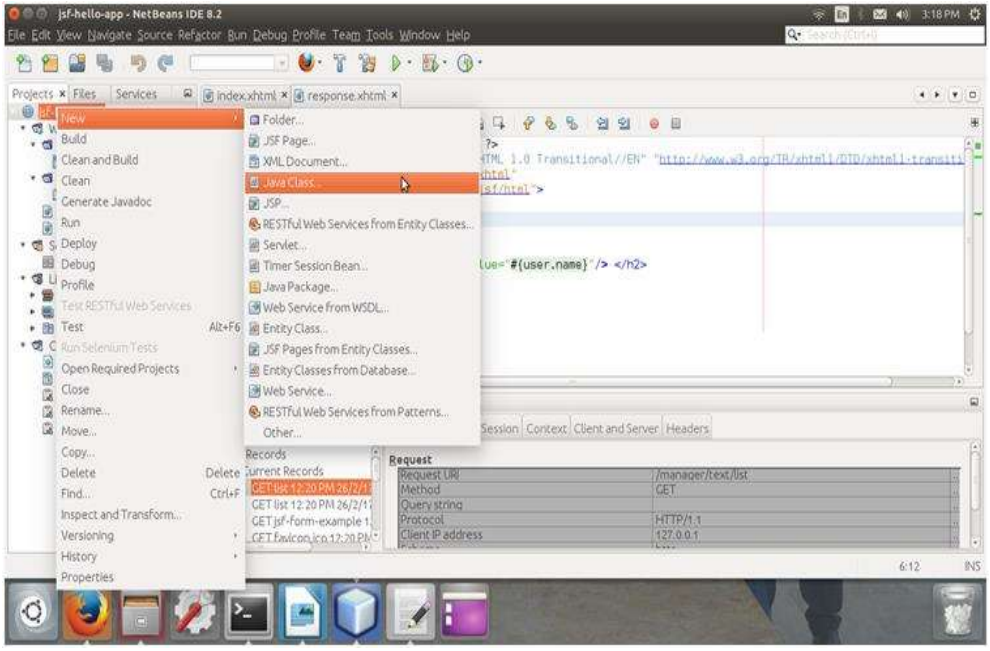
After creating response.xhtml page. Now, modify it's source code as the given below.

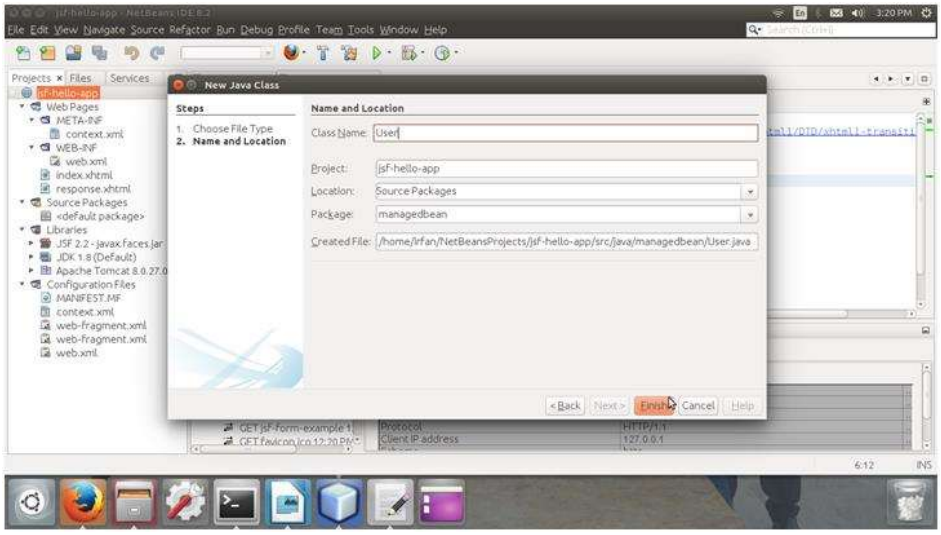
// response.xhtml

```
1. <?xml version='1.0' encoding='UTF-8' ?>
2. <!DOCTYPE html PUBLIC "-
   //W3C//DTD XHTML 1.0 Transitional//EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1-
   transitional.dtd">
3. <html xmlns="http://www.w3.org/1999/xhtml"
4. xmlns:h="http://xmlns.jcp.org/jsf/html">
5. <h:head>
6. <title>Welcome Page</title>
7. </h:head>
8. <h:body>
9. <h2>Hello, <h:outputText value="#{user.name}"></h:outputText></h2>
10. </h:body>
11. </html>
```

2) Create a Managed Bean

It is a Java class which contains properties and getter setter methods. JSF uses it as a Model. So, you can use it to write your business logic also.





After creating a Java class put the below code into your User.java file.

// User.java

```
1. import javax.faces.bean.ManagedBean;
2. import javax.faces.bean.RequestScoped;
3. @ManagedBean
4. @RequestScoped
5. public class User {
6. String name;
7.  public String getName() {
8. return name;
9. }
10. public void setName(String name) {
11. this.name = name;
12. }
13. }
```

3) Configure Application

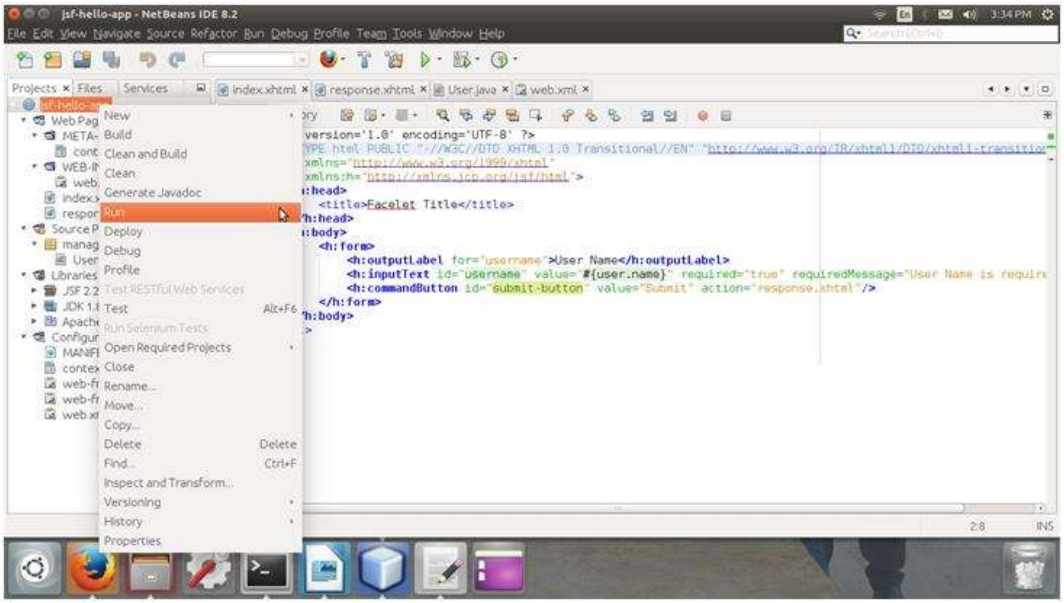
To configure application, project contains a web.xml file which helps to set FacesServlet instances. You can also set your application welcome page and any more.

Below is the code of web.xml code for this application.

// web.xml

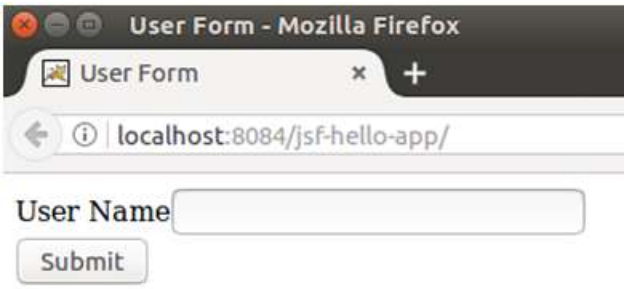
```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
4. http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
5. <context-param>
6. <param-name>javax.faces.PROJECT_STAGE</param-name>
7. <param-value>Development</param-value>
8. </context-param>
9. <servlet>
10. <servlet-name>Faces Servlet</servlet-name>
11. <servlet-class>javax.faces.webapFacelet Titlep.FacesServlet</servlet-class>
12. <load-on-startup>1</load-on-startup>
13. </servlet>
14. <servlet-mapping>
15. <servlet-name>Faces Servlet</servlet-name>
16. <url-pattern>/faces/*</url-pattern>
17. </servlet-mapping>
18. <session-config>
19. <session-timeout>
20. 30
21. </session-timeout>
22. </session-config>
23. <welcome-file-list>
24. <welcome-file>faces/index.xhtml</welcome-file>
25. </welcome-file-list>
26. </web-app>
```

Well! All set. Now run the application.

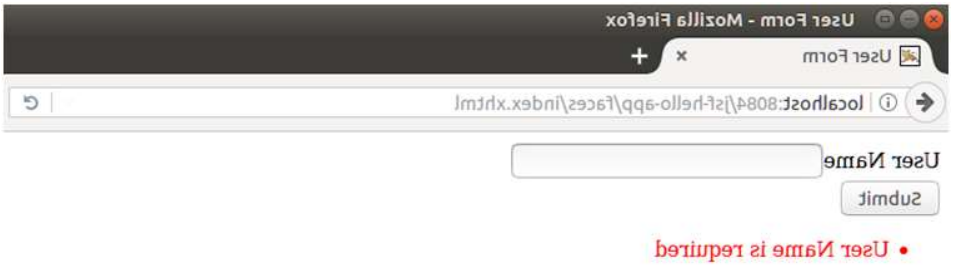


Output:

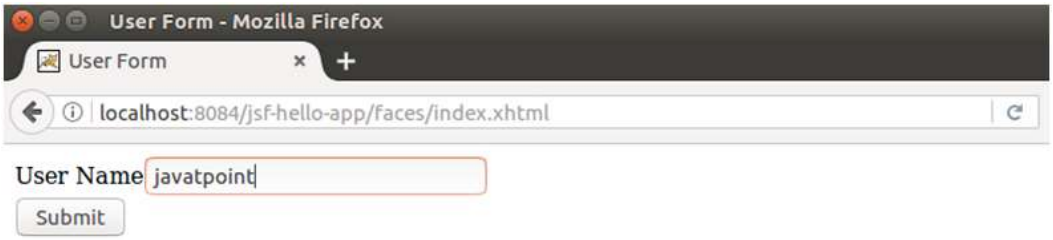
This is index page of the application.



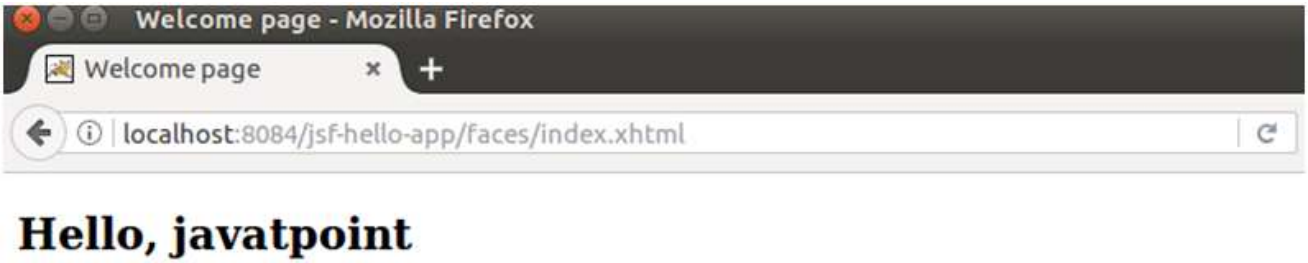
Form validation for empty field.



Providing input to input text.



This is response page after submitting input (index) page.



JSF User Interface Component Model

JavaServer Faces provides rich set of components library to define the architecture of application.

It includes the following:

Rich set of classes for specifying the state and behavior of user interface components.

- A rendering model that defines how to render the components in various ways.
- A conversion model that defines how to register data converters onto a component.
- An event and listener model that defines how to handle component events.
- A validation model that defines how to register validators onto a component.

JSF User Interface Components

JavaServer Faces HTML tag library represents HTML form components and other basic HTML elements, which are used to display or accept data from the user. A JSF form send this data to the server after submitting the form.

The following table contains the user interface components.

Tag	Functions	Rendered As	Appearance
h:inputText	It allows a user to input a string.	An HTML <input type="text"> element	A field
h:outputText	It displays a line of text.	Plain text	Plain text
h:form	It represents an input form.	An HTML <form> element	No appearance
h:commandButton	It submits a form to the application.	An HTML <input type=value> element for which the type value can be "submit", "reset", or "image"	A button
h:inputSecret	It allows a user to input a string without the actual string appearing in the field.	An HTML <input type="password"> element	A field that displays a row of characters instead of the actual string entered.
h:inputTextarea	It allows a user to enter a multiline string.	An HTML <textarea> element	A multirow field
h:commandLink	It links to another page or location on a page.	An HTML <a href> element	A link
h:inputSecret	It allows a user to input a string without	An HTML <input type="password"> element	A field that displays a row of characters

	the actual string appearing in the field.		instead of the actual string entered.
h:inputHidden	It allows a page author to include a hidden variable in a page.	An HTML <input type="hidden"> element	No appearance
h:inputFile	It allows a user to upload a file.	An HTML <input type="file"> element	A field with a Browse button
h:graphicImage	It displays an image.	An HTML <img> element	An image
h:dataTable	It represents a data wrapper.	An HTML <table> element	A table that can be updated dynamically.
h:message	It displays a localized message.	An HTML tag if styles are used	A text string
h:messages	It displays localized messages.	A set of HTML tags if styles are used	A text string
h:outputFormat	It displays a formatted message.	Plain text	Plain text
h:outputLabel	It displays a nested component as a label for a specified input field.	An HTML <label> element	Plain text
h:outputLink	It links to another page or location on a page without generating an action event.	An HTML <a> element	A link
h:panelGrid	It displays a table.	An HTML <table> element with <tr> and <td> elements	A table
h:panelGroup	It groups a set of components under one parent.	A HTML or element	A row in a table
h:selectBooleanCheckbox	It allows a user to change the value of a Boolean choice.	An HTML <input type="checkbox"> element	A check box
h:selectManyCheckbox	It displays a set of check boxes from which the user can select multiple values.	A set of HTML <input> elements of type checkbox	A group of check boxes
h:selectManyListbox	It allows a user to select multiple items from a set of items all displayed at once.	An HTML <select> element	A box
h:selectManyMenu	It allows a user to select multiple items from a set of items.	An HTML <select> element	A menu
h:selectOneListbox	It allows a user to select one item from a set of items all displayed at once.	An HTML <select> element	A box
h:selectOneMenu	It allows a user to select one item from a set of items	An HTML <select> element	A menu

h:selectOneRadio	It allows a user to select one item from a set of items.	An HTML <input type="radio"> element	A group of options
h:column	It represents a column of data in a data component.	A column of data in an HTML table	A column in a table

JSF UI Components Example

JSF provides inbuilt components to create web pages. Here, we are creating a user registration for with the help of JSF components. Follow the following steps to create the form.

1) Creating a User Registration Form

// index.xhtml

```
<!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://xmlns.jcp.org/jsf/html"
xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
<title>User Registration Form</title>
</h:head>
<h:body>
<h:form id="form">
<table>
<tr>
<td><h:outputLabel for="username">User Name</h:outputLabel></td>
<td><h:inputText id="name-id" value="#{user.name}"/></td>
</tr>
<tr>
<td><h:outputLabel for="email">Your Email</h:outputLabel></td>
<td><h:inputText id="email-id" value="#{user.email}"/></td>
</tr>
<tr>
<td><h:outputLabel for="password">Password</h:outputLabel></td>
<td><h:inputSecret id="password-id" value="#{user.password}"/></td>
</tr>

<tr>
<td><h:outputLabel for="gender">Gender</h:outputLabel></td>
<td><h:selectOneRadio value="#{user.gender}">
<f:selectItem itemValue="Male" itemLabel="Male" />
```

```
28. <f:selectItem itemValue="Female" itemLabel="Female" />
29. </h:selectOneRadio></td>
30. </tr>
31. <tr><td><h:outputLabel for="address">Address</h:outputLabel></td>
32. <td><h:inputTextarea value="#{user.address}" cols="50" rows="5"/></td></tr>
33. </table>
34. <h:commandButton value="Submit" action="response.xhtml"></h:commandButton>
35. </h:form>
36. </h:body>
37. </html>
```

2) Creating a Managed Bean

// User.java

```
1. import javax.faces.bean.ManagedBean;
2. import javax.faces.bean.RequestScoped;
3. @ManagedBean
4. @RequestScoped
5. public class User{
6. String name;
7. String email;
8. String password;
9. String gender;
10. String address;
11. public String getName() {
12. return name;
13. }
14. public void setName(String name) {
15. this.name = name;
16. }
17. public String getEmail() {
18. return email;
19. }
20.
21. public void setEmail(String email) {
22. this.email = email;
23. }
24. public String getPassword() {
25. return password;
26. }
```

28. this.password = password;
29. }
30. public String getGender() {
31. return gender;
32. }
33. public void setGender(String gender) {
34. this.gender = gender;
35. }
36. public String getAddress() {
37. return address;
38. }
39. public void setAddress(String address) {
40. this.address = address;
41. }
42. }

---

3) Creating an Output Page

// response.xhtml

1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml"
3. xmlns:h="http://xmlns.jcp.org/jsf/html"
4. xmlns:f="http://xmlns.jcp.org/jsf/core">
5. <h:head>
6. <title>User Details</title>
7. </h:head>
8. <h:body>
9. <h2><h:outputText value="Hello #{user.name}"/></h2>
10. <h4><h:outputText value="You have Registered with us Successfully, Your Details are The Following."/></h4>
11. <table>
12. <tr>
13. <td>Email: </td>
14. <td><h:outputText value="#{user.email}"/>
</td>
15. </tr>
16. <tr>
17. <td>Password:</td>
18. <td><h:outputText value="#{user.password}"/>
</td>

20. <tr>
21. <td>Gender:</td>
22. <td><h:outputText value="#{user.gender}"/>
</td>
23. </tr>
24. <tr>
25. <td>Address: </td>
26. <td><h:outputText value="#{user.address}"/></td>
27. </tr>
28. </table>
29. </h:body>
30. </html>

4) Run the Application

Output:

User Registration Form (index page).

After submitting form, JSF renders response.xhtml file as a result web page.

// response page

JSF <h:inputText>Tag

The JSF <h: inputText> tag is used to render an input field on the web page.

It is used within a <h: form> tag to declare input field that allows user to input data.

The value attribute refers to the name property of a managed bean named User. This property holds the data for the name component. After the user submits the form, the value of the name property in User will be set to the text entered in the field corresponding to this tag.

JSF <h:InputText> Tag Example:

In the following example, we are using a label tag for providing label to inputText tag, a inputText with attributes, a commandButton to represent a submit button. All are enclosed in a <h:form> tag.

```
1. <h:inputText id="username" value="#{user.name}" label="username" maxlength="10"
2. size="15" alt="username" autocomplete="off" readonly="false" required="true"
3. requiredMessage="Username is required" style="color:red" accesskey="q">
4.     </h:inputText>
```

JSF renders <h:inputText> tag as below:

```
1. <input id="userform:username" name="userform:username" autocomplete="off" accesskey="q"
2. alt="username" maxlength="10" size="15" style="color:red" type="text">
```

Output:

JSF <h:inputText> Tag Attributes

Attribute name	Description
id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
value	It is used to collect present value of the inputText.
class	It gives class name to the component. It is used to access component from CSS and JS file.
maxlength	The maximum number of characters that may be entered in this field.
alt	Alternate textual description of the element rendered by this component.
accesskey	Access key that, when pressed, transfers focus to this element. It varies browser to browser.
size	The number of characters used to determine the width of this field .
required	It indicates that the user is required to provide a submitted value for this input component.
requiredMessage	If required attribute is set to true, the message description provided in the requiredMessage is display to the web page.
style	It is used to apply CSS for the component.
rendered	It is used to render the component. The default value for this property is true.
convertor	It is used to converter instance registered with this component.
readonly	It indicates that this component prohibits changes by the user. You can make component readonly by passing readonly as a value of this attribute. eg. readonly = "readonly"

JSF <h:outputText> Tag

It is used to render a plain text. If the "styleClass", "style", "dir" or "lang" attributes are present, render a "span" element. If the "styleClass" attribute is present, render its value as the value of the "class" attribute.

JSF <h:outputText> Tag Example 1:

```
1. <h:outputText value="hello"></h:outputText>
```

JSF renders it as plain text

hello

JSF <h:outputText> Tag Example 2:

```
1. <h:outputText value="hello" lang="en" style="color: red"></h:outputText>
```

JSF renders it as a HTML span tag

```
1. <span style="color: red" lang="en">hello</span>
```

HTML Output:

JSF <h:outputText> Tag attributes

Attribute	Description
value	It holds current value of this component.
id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
style	It is used to apply CSS for the component.
class	It gives class name to the component. It is used to access component from CSS and JS file.
lang	It is used to specify language. It helps to make web page localized.

JSF <h:form> Tag

The <h:form> tag represents an input form. It includes child components that can contain data which is either presented to the user or submitted with the form. It can also include HTML markup to lay out the components on the page.

Note: The h:form tag itself does not perform any layout, its purpose is to collect data and to declare attributes that can be used by other components in the form.

JSF <h:form> Tag Declaration

```
1. <h:form>
2. <!-- form elements -->
3. </h:form>
```

JSF <h:form> Tag Example:

```
1. <h:form id="user-form">
2. <h:outputLabel for="username">User Name</h:outputLabel>
3. <h:inputText id="username" value="#{user.name}" required="true" requiredMessage="Username is required"/><br/>
4. <h:commandButton id="submit-button" value="Submit" action="response.xhtml"/>
```

Output:

JSF <h:form> Tag attributes:

Attribute	Description
accept	List of content types that a server processing this form will handle correctly.
class	CSS class name for this component.
enctype	It is used to submit content to the server. If not specified, the default value is "application/x-www-form-urlencoded".
id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
lang	It describes the language used in the generated markup for this component.
rendered	It is used to render the component. The default value for this property is true.
onclick	It executes Javascript code when a pointer button is clicked over this element.
onsubmit	It executes Javascript code when form is submitted.

JSF <h:commandButton> Tag

It creates a submit button and used to submit a application form. You can create it by using the following syntax.

1. <h:commandButton></h:commandButton>

JSF <h:commandButton> Tag Example:

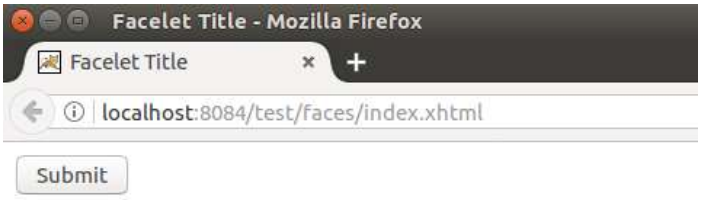
In the following example, we have created a form which is submitted by using a <h:commandButton>.

1. <h:form id="user-form">
2. <h:outputLabel for="username">User Name</h:outputLabel>
3. <h:inputText id="username" value="#{user.name}" required="true" requiredMessage="Username is required"/>
4. <h:commandButton id="submit-button" value="Submit" action="response.xhtml"/>
5. </h:form>

JSF renders <h:commandButton> tag as below:

1. <input id="user-form:submit-button"
2. name="user-form:submit-button" value="Submit" type="submit">

Output:



JSF <h:commandButton> Tag Attributes:

Attribute	Description

	element in CSS and JS file.
value	It holds current value for the commandbutton and display it as name of submit button.
action	It is used to specify action for the form. The commandButton submits the form to the server at the specified action. If you don't provide action, page redirect to the same page after submitting.
disabled	It is used to make a commandButton disabled. You can't click on the button after applying this attribute.
image	It is used to set an image on the commandButton. In this case, your image will work as submit button.
label	It is used to make localized name for your comandButton.
rendered	It is used to render the component. The default value for this property is true.
type	It is used to specify type of button. You can set "reset", "submit" or "button". If don't specify, it is submit by default.
style	It is used to specify CSS for the component.
onclick	It is used to execute JavaScript code when commandButton is clicked.
accesskey	It is used to access submit-button by using specified key.

JSF <h:inputTextarea> Tag

JSF renders this as an HTML "textarea" element. It allows a user to enter multiline string.

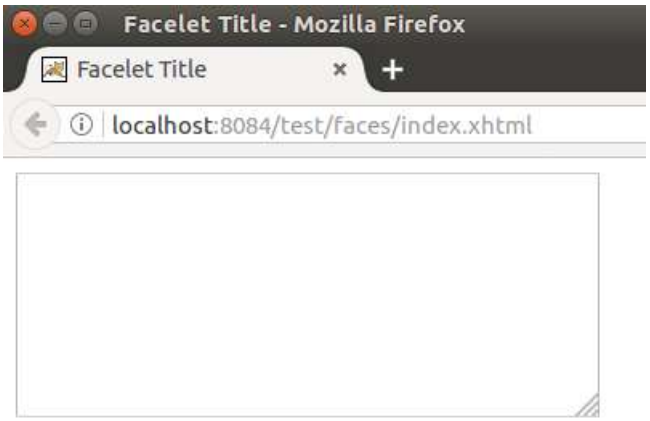
Example

```
1. <h:inputTextarea id="text-area-id" value="#{user.address}" required="true"
2. requiredMessage="Address is required" cols="50" rows="10"></h:inputTextarea>
```

JSF renders <h:inputTextarea> tag as below:

```
1. <textarea id="user-form:text-area-id"
2. name="user-form:text-area-id" cols="50" rows="10"></textarea>
```

Output:



Attribute

Attribute	Description
Id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
value	It holds current value of this component.
cols	It is used to set number of cols for the textarea.
rows	It is used to set number of rows for the textarea.
required	It indicates that the user is required to provide a submitted value for this input component.
requiredMessage	If required attribute is set to true, the message description provided in the requiredMessage

	is display to the web page.
disabled	It is used to disabled component. You can disabled it by assigning true value.
onclick	It is method which invokes JavaScript code after onclick on the textarea.
onselect	It is a method which invokes JavaScript code when a user select this component.
readonly	It indicates that this component prohibits changes by the user. You can make component readonly by passing readonly as a value of this attribute. eg. readonly = "readonly"
rendered	It is used to render this component. You can set it's value true or false. Default value is true.
label	It is used to set a localized name for this component.
lang	It is used to set language for this component.
style	It is used to set CSS style code to provide better user interface of this component.
accesskey	Access key that, when pressed, transfers focus to this element. It varies browser to browser.

JSF <h:commandLink> Tag

JSF renders it as an HTML "a" anchor element that acts like a form submit button when clicked. So, you can create anchor tag by using this tag. An h:commandLink tag must include a nested h:outputText tag, which represents the text that the user clicks to generate the event. It's also required to be placed inside a <h:form> tag.

JSF <h:commandLink> Tag Example

```
1. <h:commandLink id="image-link-id" action="response.xhtml">
2. <h:outputText value="Click here"></h:outputText>
3. </h:commandLink>
```

JSF renders <h:commandLink> tag as below:

```
1. <a id="user-form:image-link-id" href="#"
2. onclick="mojarra.jsfcljs(document.getElementById('user-form'),{'user-form:image-link-id':'user-
   form:image-link-id'},'');return
3. false">Click here</a>
```

Output:

Note: The h:commandLink tag renders JavaScript scripting language code. If you use it, make sure that your browser is JavaScript enabled.

JSF <h:commandLink> Tag Attributes

Attribute	Description
Id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
value	It is used to display label for this component.
type	The content type of the resource designated by this hyperlink.
action	It is used to provide action for this commandlink.
accesskey	Access key that, when pressed, transfers focus to this element. It varies browser to browser.
actionListener	It represents an action listener method that will be notified when this component is activated

	by the user.
charset	The character encoding of the resource designated by this hyperlink.
coords	It is used to set position and shape of the hot spot on the screen (for use in client-side image maps).
disabled	It is used to disabled component. You can disabled it by assigning true value.
hreflang	It is used to set the language code of the resource designated by this hyperlink.
Rel	The relationship from the current document to the anchor specified by this hyperlink. The value of this attribute is a space-separated list of link types.
rendered	It is used to render this component. You can set it's value true or false. Default value is true.
Rev	It is a reverse link from the anchor specified by this hyperlink to the current document. The value of this attribute is a space-separated list of link types.
shape	It represents the shape of the hot spot on the screen. The valid values are: default (entire region); rect (rectangular region); circle (circular region); and poly (polygonal region).
style	It is used to set CSS style code to provide better user interface of this component.

JSF <h:inputSecret> Tag

It is a standard password field which accepts one line of text with no spaces and displays it as a set of asterisks as it is entered. In other words, we say, it is used to create a HTML password field which allows a user to input a string without the actual string appearing in the field.

Example:

```
1. <h:inputSecret value="#{user.password}" maxlength="10" size="15"
2. required="true" requiredMessage="Password is required"></h:inputSecret>
```

JSF renders it as below HTML

```
1. <input name="user-password" value="" maxlength="10" size="15" type="password">
```

Output:

JSF <h:inputSecret> Tag Attributes

Attribute	Description
Id	
value	It holds the current value of inputSecret tag.
maxlength	It represents the maximum number of characters that may be entered in this field.
readonly	It indicates that this component prohibits changes by the user. You can make component readonly by passing readonly as a value of this attribute. eg. readonly = "readonly"
rendered	It is used to render the component. The default value for this property is true.
required	It indicates that the user is required to provide a submitted value for this input component.
requiredMessage	If required attribute is set to true, the message description provided in the requiredMessage is display to the web page.
Size	The number of characters used to determine the width of this field.
style	It is used to specify CSS for the component.
class	CSS class name for this component.

label	It is used to make localized name for your comandButton.
Lang	It is used to specify language. It helps to make web page localized.
accesskey	It is used to access component by using specified key.

JSF <h:inputHidden> Tag

It renders an HTML "input" element of type hidden. It does not appear in web page, so you can pass hidden information while submitting form.

Example

```
1. <h:inputHidden value="#{user.id}" id="userId" class="userId-class"></h:inputHidden>
```

Output:

This is hidden type input text. It dosen't display on web page.

Attributes

Attribute	Description
Id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
Value	It holds the current value of this field.
Class	CSS class name for this component.
Required	It indicates that the user is required to provide a submitted value for this input component.
requiredMessage	If required attribute is set to true, the message description provided in the requiredMessage is display to the web page.

JSF <h:inputFile> Tag

JSF renders it as an HTML element of type file. It is used to get file as input. In HTML form, it allows a user to upload a file.

Example:

```
1. <h:inputFile  id="file-id"  value="#{user.fileName}" required="true"
2. requiredMessage="Please upload a file" alt="upload file"></h:inputFile>
```

Output:

Attributes

Attribute	Description
id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
value	It holds the current value of this component.
class	CSS class name for this component.
alt	It is used to set alternate name for the component.

required	It indicates that the user is required to provide a submitted value for this input component.
requiredMessage	If required attribute is set to true, the message description provided in the requiredMessage is display to the web page.
disabled	It is used to disabled component. You can disabled it by assigning true value.
label	It is used to set a localized name for this component.
lang	It is used to set language for this component.
onclick	It is a method which invokes JavaScript code when a user click on this component.
onselect	It is a method which invokes JavaScript code when a user select this component.
rendered	It is used to render this component. By default it's value is true.
style	It is used to set CSS style code to provide better user interface of this component.

JSF <h:graphicImage> Tag

JSF renders an HTML element "img" tag. This tag is used to render an image on the web page.

Example

```
1. <h:graphicImage id="image-id" name="user-image" url="#{user.fileLocation()}"
2. height="50px" width="50px" alt="Image not found"></h:graphicImage>
```

In the above example, url attribute specifies the path to the image. The URL of the example tag begins with a slash (/), which adds the relative context path of the web application to the beginning of the path to the image.

Attributes

Attribute	Description
id	It is an identifier for this component. This id must be unique. You can use it to access HTML element in CSS and JS file.
name	It is used to set resource-name for this resource.
url	It is a context-relative URL to retrieve the resource associated with this component. This is an alias for the "value" property.
value	It represents current value of this component.
height	It is used to set hight of the image.
width	It is used to set width of this image.
alt	It is used to set alternate name of the image.
class	It represents CSS class for this component.

dir	It is a direction indication for text that does not inherit directionality. Valid values are "LTR" (left-to-right) and "RTL" (right-to-left).
library	It is used to set library-name for this resource.
onclick	It is method which invokes JavaScript code after onclick on the image.
rendered	It is used to render this component. You can set it's value true or false. Default value is true.
ismap	It is a flag indicating that this image is to be used as a server side image map. Such an image must be enclosed within a hyperlink ("a"). A value of false causes no attribute to be rendered, while a value of true causes the attribute to be rendered as ismap="ismap".
usemap	The name of a client side image map (an HTML "map" element) for which this element provides the image.
style	It is used to set CSS style code to provide better user interface of this component.

JSF <h:message> Tag

It is used to display a single message for a particular component. You can display your custom message by passing id of that component into the for attribute.

Tag Attributes

Attribute	Description
for	It is mandatory tag which is used to assign component's id, for that message is composed.
errorClass	It is used to apply CSS style class to any message with a severity class of "ERROR".
errorStyle	It is used to apply CSS style to any message with a severity class of "ERROR".
fatalClass	It is used to apply CSS style class to any message with a severity class of "FATAL".
FatalStyle	It is used to apply CSS style to any message with a severity class of "FATAL".
infoClass	It is used to apply CSS style class to any message with a severity class of "INFO".
InfoStyle	It is used to apply CSS style to any message with a severity class of "INFO".
tooltip	It is used to display detail portion of the message as a tooltip.
warnClass	It is used to apply CSS style class to any message with a severity class of "WARN".
warnStyle	It is used to apply CSS style to any message with a severity class of "WARN".

Example

// index.xhtml

```
1. <h:form id="form">
2. <h:outputLabel for="username">User Name</h:outputLabel>
```

```
3. <h:inputText id="name-id" value="#{user.name}"/>
4. <h:message for="name-id" style="color: red"/>
5. <br/>
6. <h:outputLabel for="mobile">Mobile No.</h:outputLabel>
7. <h:inputText id="mobile-id" value="#{user.mobile}"/>
8. <h:message for="mobile-id" style="color: red"/>
9. <br/>
10. <h:commandButton value="OK" action="response.xhtml"></h:commandButton>
11. </h:form>
```

// User.java

```
1. import javax.faces.bean.ManagedBean;
2. import javax.faces.bean.RequestScoped;
3. import javax.validation.constraints.NotNull;
4. import javax.validation.constraints.Size;
5. @ManagedBean
6. @RequestScoped
7. public class User{
8. @NotNull(message = "Name can't be empty")
9. String name;
10. @NotNull(message = "Mobile can't be empty")
11. @Size(min = 10, max = 10, message = "Mobile must have 10 digits")
12. String mobile;
13. public String getName() {
14. return name;
15. }
16. public void setName(String name) {
17. this.name = name;
18. }
19. public String getMobile() {
20. return mobile;
21. }
22. public void setMobile(String mobile) {
23. this.mobile = mobile;
24. }
25. }
```

Output:

JSF <h:messages> Tag

It is used to displays all messages that were stored in the faces context during the course of the JSF life cycle.

Example:

// index.xhtml

```
1. <h:form id="form">
2. <h:outputLabel for="username">User Name</h:outputLabel>
3. <h:inputText id="name-id" value="#{user.name}"/><br/>
4. <h:outputLabel for="mobile">Mobile No.</h:outputLabel>
5. <h:inputText id="mobile-id" value="#{user.mobile}"/><br/>
6. <h:commandButton value="OK" action="response.xhtml"></h:commandButton>
7. <!-- Here, we are single tag to display all the errors. -->
8. <h:messages style="color: red"></h:messages>
9. </h:form>
```

// User.java

```
1. import javax.faces.bean.ManagedBean;
2. import javax.faces.bean.RequestScoped;
3. import javax.validation.constraints.NotNull;
4. import javax.validation.constraints.Size;
5. @ManagedBean
6. @RequestScoped
7. public class User{
8. @NotNull(message = "Name can't be empty")
9. String name;
10. @NotNull(message = "Mobile can't be empty")
11. @Size(min = 10, max = 10, message = "Mobile must have 10 digits")
12. String mobile;
13. public String getName() {
14. return name;
15. }
16. public void setName(String name) {
17. this.name = name;
18. }
19. public String getMobile() {
```

21. }
22. public void setMobile(String mobile) {
23. this.mobile = mobile;
24. }
25. }

Output

SF <h:dataTable> Tag

It is used to create a data table. A table that can be updated dynamically.

Tag Attributes

Attribute	Description
id	It is mandatory tag which is used to assign component?s id, for that message is composed.
class	It represents CSS class for this component.
lang	It is used to set language for this component.
bgcolor	It is used to set background color for this table.
binding	It is a ValueExpression linking this component to a property in a backing bean.
bodyrows	It is a comma separated list of row indices for which a new "tbody" element should be started.
border	It is used to set border width which is to be drawn around this table.
captionClass	It is used for Space-separated list of CSS style class(es) that will be applied to any caption generated for this table.
captionStyle	It is used to set CSS style for a caption of this component. This style is applied when component is rendered.
cellpadding	It is used to define that how much space the user should leave between the border of each cell and its contents.
cellspacing	It is used to define that how much space the user should leave between the left side of the table and the leftmost column, the top of the table and the top of the top side of the topmost row, and so on. It also specifies the amount of space to leave between the cells.
columnClasses	It is used to set comma-delimited list of CSS style classes that will be applied to the columns of this table.

dir	It is used for text that does not inherit directionality. Valid values are "LTR" (left-to-right) and "RTL" (right-to-left).
first	It is a zero-relative row number of the first row to be displayed. If this property is set to zero, rendering will begin with the first row of the underlying data.
footerClass	It contains space-separated list of CSS style class that will be applied to any footer generated for this table.
frame	It is used to set frame. Code specifying which sides of the frame surrounding this table will be visible. Valid values are: none (no sides, default value); above (top side only); below (bottom side only); hsides (top and bottom sides only); vsides (right and left sides only); lhs (left hand side only); rhs (right hand side only); box (all four sides); and border (all four sides).
headerClass	It is used to set header class. Space-separated list of CSS style class(es) that will be applied to any header generated for this table.
rowClasses	It is used to set CSS class for row. Comma-delimited list of CSS style classes that will be applied to the rows of this table.
rows	It is used to set number of rows to be displayed. Starting with the one identified by the "first" property. If this value is set to zero, all available rows in the underlying data model will be displayed.
summary	It is used to get summary of this table's purpose and structure, for user rendering to non-visual media such as speech and Braille.
title	It is used to set advisory title information about markup elements generated for this component.

JSF <h:dataTable> Tag Example

// index.xhtml

```
1. <h:body>
2. <h2><h:outputText value="Product List"></h:outputText></h2>
3. <h:dataTable value="#{product.productList}" var="p">
4. <h:column>
5. <f:facet name="header">ID</f:facet>
6. <h:outputText value="#{p.id}"/>
7. </h:column>
8. <h:column>
9.         <f:facet name="header">Name</f:facet>
10.        <h:outputText value="#{p.name}"/>
11. </h:column>
12. <h:column>
13.        <f:facet name="header">Price</f:facet>
```

```
15. </h:column>
16.     </h:dataTable>
17. </h:body>
```

// Product.Java

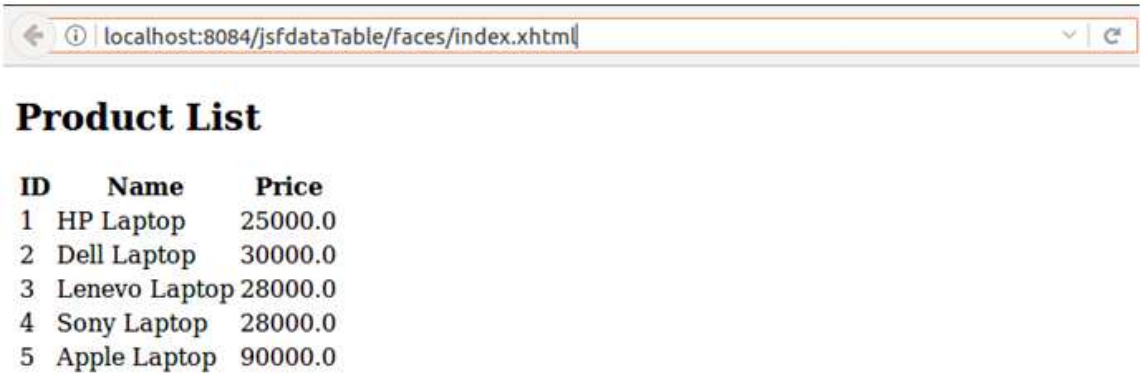
```
1.  public class Product {
2.  private final int id;
3.  private final String name;
4.  private final float price;
5.  public Product(int id, String name, float price) {
6.  this.id = id;
7.  this.name = name;
8.  this.price = price;
9.  }
10. public int getId() {
11. return id;
12. }
13. public String getName() {
14. return name;
15. }
16. public float getPrice() {
17. return price;
18. }
19. }
```

// DataTable.java

```
1.  import java.util.List;
2.  import java.util.ArrayList;
3.  import javax.faces.bean.ManagedBean;
4.  import javax.faces.bean.RequestScoped;
5.
6.  @ManagedBean(name = "product")
7.  @RequestScoped
8.  public class DataTable {
9.  public List<Product> productsList;
10. public List<Product> getProductList() {
11. productsList = new ArrayList<>();
12. productsList.add(new Product(1,"HP Laptop",25000f));
13. productsList.add(new Product(2,"Dell Laptop",30000f));
14. productsList.add(new Product(3,"Lenevo Laptop",28000f));
```

```
16. productsList.add(new Product(5,"Apple Laptop",90000f));
17. return productsList;
18. }
19. }
```

Output:



JSF - Expression Language

JSF provides a rich expression language. We can write normal operations using **#{operation-expression}** notation. Following are some of the advantages of JSF Expression languages.

- Can reference bean properties where bean can be an object stored in request, session or application scope or is a managed bean.
- Provides easy access to elements of a collection which can be a list, map or an array.
- Provides easy access to predefined objects such as a request.
- Arithmetic, logical and relational operations can be done using expression language.
- Automatic type conversion.
- Shows missing values as empty strings instead of NullPointerException.

Example Application

Let us create a test JSF application to test expression language.

Step	Description
1	Create a project with a name *helloworld* under a package *com.tutorialspoint.test* as explained in the *JSF - First Application* chapter.
2	Modify *UserData.java* under package *com.tutorialspoint.test* as explained below.
3	Modify *home.xhtml* as explained below. Keep the rest of the files unchanged.
4	Compile and run the application to make sure the business logic is working as per the requirements.
5	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.
6	Launch your web application using appropriate URL as explained below in the last step.

UserData.java

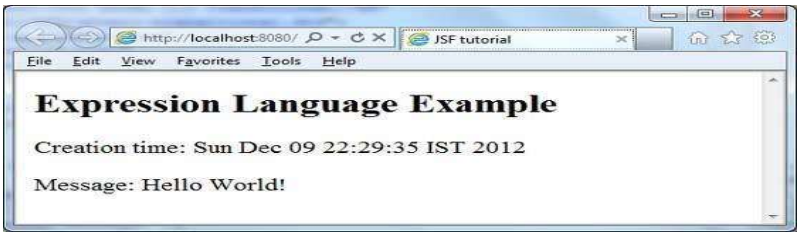
```
import java.io.Serializable;
import java.util.Date;

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

@ManagedBean(name = "userData", eager = true)
@SessionScoped
public class UserData implements Serializable {
   private static final long serialVersionUID = 1L;
   private Date createTime = new Date();
   private String message = "Hello World!";

   public Date getCreateTime() {
      return(createTime);
   }

   public String getMessage() {
      return(message);
   }
}
```

home.xhtml

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

   <h:head>
      <title>JSF Tutorial!</title>
   </h:head>

   <h:body>
      <h2>Expression Language Example</h2>
      Creation time:
      <h:outputText value = "#{userData.createTime}"/>
```

```
    Message:
    <h:outputText value = "#{userData.message}"/>
  </h:body>
</html>
```

Once you are ready with all the changes done, let us compile and run the application as e did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.



JSF - EVENT HANDLING

When a user clicks a JSF button or link or changes any value in the text field, JSF UI component fires an event, which will be handled by the application code. To handle such an event, an event handler is to be registered in the application code or managed bean.

When a UI component checks that a user event has occured, it creates an instance of the corresponding event class and adds it to an event list. Then, Component fires the event, i.e., checks the list of listeners for that event and calls the event notification method on each listener or handler.

JSF also provide system level event handlers, which can be used to perform some tasks when the application starts or is stopping.

Following are some important *Event Handler* in JSF 2.0:

S.No	Event Handlers & Description
1	valueChangeListener : Value change events get fired when the user make changes in input components.
2	actionListener : Action events get fired when the user clicks a button or link component.
3	Application Events : Events firing during JSF lifecycle: PostConstructApplicationEvent, PreDestroyApplicationEvent , PreRenderViewEvent.

JSF – valueChangeListener: When the user interacts with input components, such as h:inputText or h:selectOneMenu, the JSF fires a valueChangeEvent, which can be handled in two ways.

S.No	Technique & Description
1	**Method Binding:** Pass the name of the managed bean method in *valueChangeListener* attribute of UI Component.
2	**ValueChangeListener:** Implement ValueChangeListener interface and pass the implementation class name to *valueChangeListener* attribute of UI Component.

Method Binding

Define a method

```
public void localeChanged(ValueChangeEvent e) {
```

```
    //assign new value to country
   selectedCountry = e.getNewValue().toString();
}
```

Use the above method

```
<h:selectOneMenu value = "#{userData.selectedCountry}"  onchange = "submit()"
   valueChangeListener = "#{userData.localeChanged}" >
   <f:selectItems value = "#{userData.countries}" />
</h:selectOneMenu>
```

ValueChangeListener

Implement ValueChangeListener

```
public class LocaleChangeListener implements ValueChangeListener {
   @Override
   public void processValueChange(ValueChangeEvent event)
      throws AbortProcessingException {

      //access country bean directly
      UserData userData = (UserData) FacesContext.getCurrentInstance().
      getExternalContext().getSessionMap().get("userData");
      userData.setSelectedCountry(event.getNewValue().toString());
   }
}
```

Use listener method

```
<h:selectOneMenu value = "#{userData.selectedCountry}" onchange = "submit()">
   <f:valueChangeListener type = "com.tutorialspoint.test.LocaleChangeListener" />
   <f:selectItems value = "#{userData.countries}" />
</h:selectOneMenu>
```

Example Application

Let us create a test JSF application to test the valueChangeListener in JSF.

Step	Description
1	Create a project with a name *helloworld* under a package *com.tutorialspoint.test* as explained in the *JSF - First Application* chapter.
2	Modify *UserData.java* file as explained below.
3	Create *LocaleChangeListener.java* file under a package *com.tutorialspoint.test*. Modify it as explained below.
4	Modify *home.xhtml* as explained below. Keep the rest of the files unchanged.
5	Compile and run the application to make sure the business logic is working as per the requirements.
6	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.

7	Launch your web application using appropriate URL as explained below in the last step.

UserData.java

```
import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Map;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.event.ValueChangeEvent;

@ManagedBean(name = "userData", eager = true)
@SessionScoped
public class UserData implements Serializable {
  private static final long serialVersionUID = 1L;
  private static Map<String,String> countryMap;
  private String selectedCountry = "United Kingdom"; //default value

  static {
    countryMap = new LinkedHashMap<String,String>();
    countryMap.put("en", "United Kingdom"); //locale, country name
    countryMap.put("fr", "French");
    countryMap.put("de", "German");
  }

  public void localeChanged(ValueChangeEvent e) {
    //assign new value to country
    selectedCountry = e.getNewValue().toString();
  }

  public Map<String, String> getCountries() {
    return countryMap;
  }

  public String getSelectedCountry() {
    return selectedCountry;
  }

  public void setSelectedCountry(String selectedCountry) {
    this.selectedCountry = selectedCountry;
  }
}
```

LocaleChangeListener.java

```
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

public class LocaleChangeListener implements ValueChangeListener {

  @Override
  public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {

    //access country bean directly
```

```
    UserData userData = (UserData) FacesContext.getCurrentInstance().
    getExternalContext().getSessionMap().get("userData");
    userData.setSelectedCountry(event.getNewValue().toString());
  }
}
```

home.xhtml

```
<?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>JSF tutorial</title>
  </h:head>

  <h:body>
    <h2>valueChangeListener Examples</h2>

    <h:form>
      <h2>Method Binding</h2>
      <hr/>
      <h:panelGrid columns = "2">
        Selected locale :
        <h:selectOneMenu value = "#{userData.selectedCountry}"
          onchange = "submit()"
          valueChangeListener = "#{userData.localeChanged}" >
          <f:selectItems value = "#{userData.countries}" />
        </h:selectOneMenu>
        Country Name:
        <h:outputText id = "country" value = "#{userData.selectedCountry}"
        size = "20" />
      </h:panelGrid>
    </h:form>

  </h:body>
</html>
```

Once you are ready with all the changes done, let us compile and run the application as we did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.

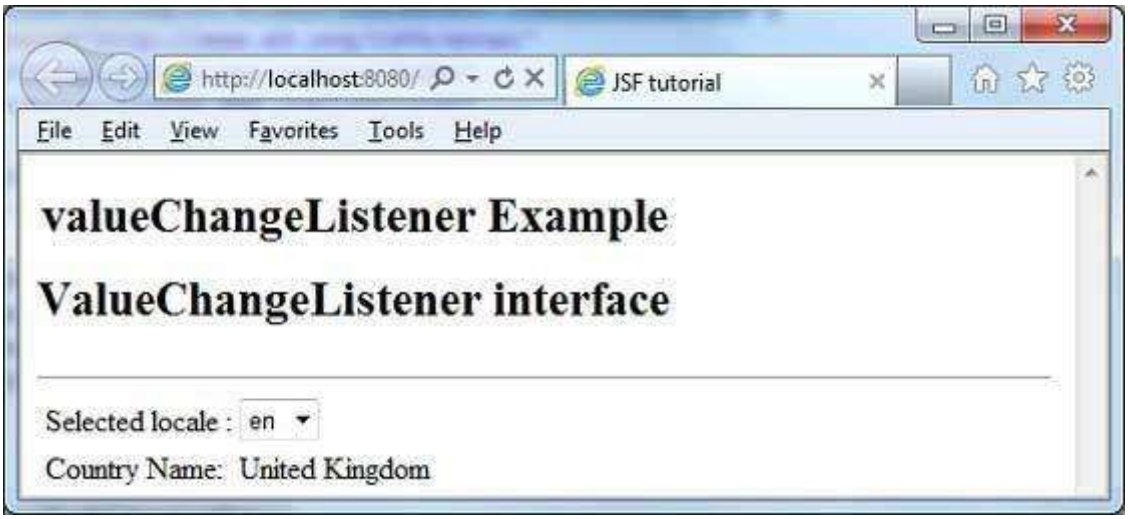
Select locale. You will see the following result.

Modify ***home.xhtml*** again in the deployed directory where you've deployed the application as explained below. Keep the rest of the files unchanged.

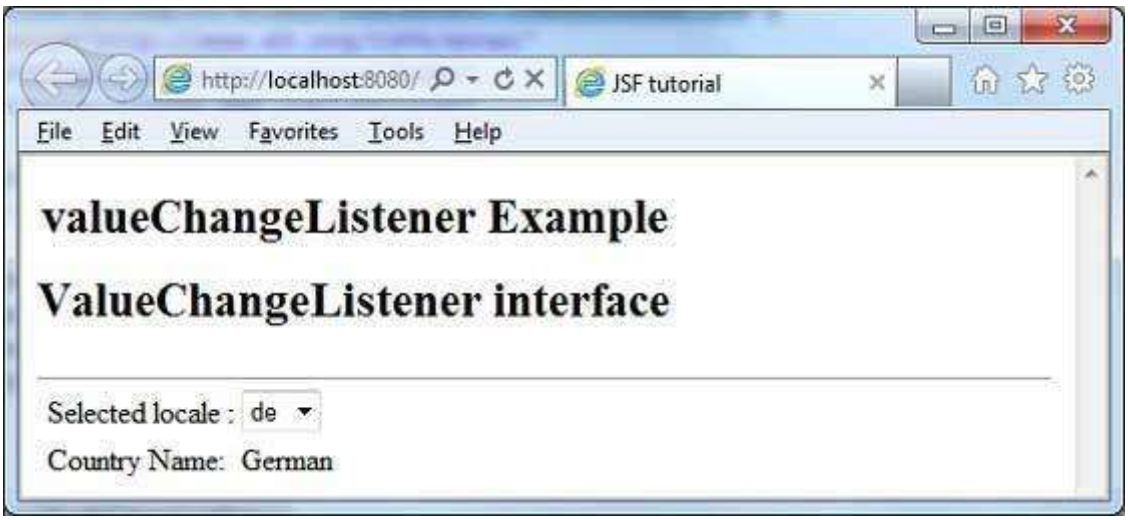
home.xhtml

```
<?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>JSF tutorial</title>
   </h:head>

   <h:body>
      <h2>valueChangeListener Examples</h2>

      <h:form>
         <h2>ValueChangeListener interface</h2>
         <hr/>
         <h:panelGrid columns = "2">
            Selected locale :
            <h:selectOneMenu value = "#{userData.selectedCountry}"
               onchange = "submit()">
               <f:valueChangeListener
                  type = "com.tutorialspoint.test.LocaleChangeListener" />
               <f:selectItems value = "#{userData.countries}" />
            </h:selectOneMenu>
            Country Name:
            <h:outputText id = "country1" value = "#{userData.selectedCountry}"
               size = "20" />
         </h:panelGrid>
      </h:form>

   </h:body>
</html>
```

Once you are ready with all the changes done, refresh the page in the browser. If everything is fine with your application, this will produce the following result.



Select locale. You will see the following result.



JSF – actionListener: When the user interacts with the components, such as h:commandButton or h:link, the JSF fires action events which can be handled in two ways.

S.No	Technique & Description
1	**Method Binding :** Pass the name of the managed bean method in *actionListener* attribute of UI Component.
2	**ActionListener:** Implement ActionListener interface and pass the implementation class name to *actionListener* attribute of UI Component.

Method Binding

Define a method

```
public void updateData(ActionEvent e) {
  data = "Hello World";
}
```

Use the above method

```
<h:commandButton id = "submitButton"   value = "Submit" action = "#{userData.showResult}"
  actionListener = "#{userData.updateData}" />
</h:commandButton>
```

ActionListener

Implement ActionListener

```
public class UserActionListener implements ActionListener {

   @Override
   public void processAction(ActionEvent arg0)
   throws AbortProcessingException {

      //access userData bean directly
      UserData userData = (UserData) FacesContext.getCurrentInstance().
      getExternalContext().getSessionMap().get("userData");
      userData.setData("Hello World");
   }
}
```

Use listener method

```
<h:commandButton id = "submitButton1"  value = "Submit" action = "#{userData.showResult}" >
   <f:actionListener type = "com.tutorialspoint.test.UserActionListener" />
</h:commandButton>
```

Example Application

Let us create a test JSF application to test the actionListener in JSF.

Step	Description
1	Create a project with a name *helloworld* under a *package com.tutorialspoint.test* as explained in the *JSF - First Application* chapter.
2	Modify *UserData.java* file as explained below.
3	Create *UserActionListener.java* file under a package *com.tutorialspoint.test*. Modify it as explained below.
4	Modify *home.xhtml* as explained below. Keep the rest of the files unchanged.
5	Modify *result.xhtml* as explained below. Keep the rest of the files unchanged.
6	Compile and run the application to make sure the business logic is working as per the requirements.
7	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.
8	Launch your web application using appropriate URL as explained below in the last step.

UserData.java

```
import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Map;

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

@ManagedBean(name = "userData", eager = true)
@SessionScoped
```

```
  private static final long serialVersionUID = 1L;
  private static Map<String,String> countryMap;
  private String data = "sample data";

  public String showResult() {
    return "result";
  }

  public void updateData(ActionEvent e) {
    data="Hello World";
  }

  public String getData() {
    return data;
  }

  public void setData(String data) {
    this.data = data;
  }
}
```

UserActionListener.java

```
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class UserActionListener implements ActionListener {

  @Override
  public void processAction(ActionEvent arg0)
  throws AbortProcessingException {

    //access userData bean directly
    UserData userData = (UserData) FacesContext.getCurrentInstance().
    getExternalContext().getSessionMap().get("userData");
    userData.setData("Hello World");
  }
}
```

home.xhtml

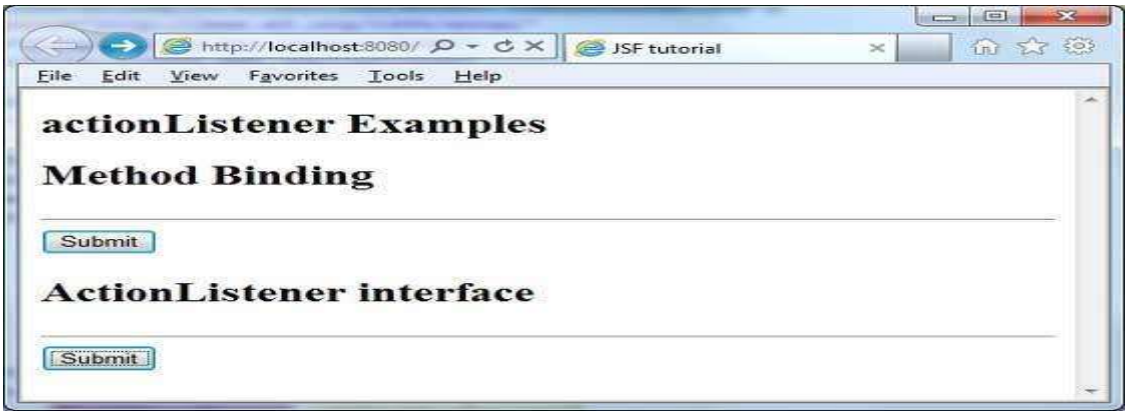
```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<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>JSF tutorial</title>
  </h:head>
```

```
    <h2>actionListener Examples</h2>

    <h:form>
      <h2>Method Binding</h2>
      <hr/>

      <h:commandButton id = "submitButton"
        value = "Submit" action = "#{userData.showResult}"
        actionListener = "#{userData.updateData}" />
      </h:commandButton>
      <h2>ActionListener interface</h2>
      <hr/>

      <h:commandButton id = "submitButton1"
        value = "Submit" action = "#{userData.showResult}" >
        <f:actionListener
          type = "com.tutorialspoint.test.UserActionListener" />
      </h:commandButton>
    </h:form>

  </h:body>
</html>
```

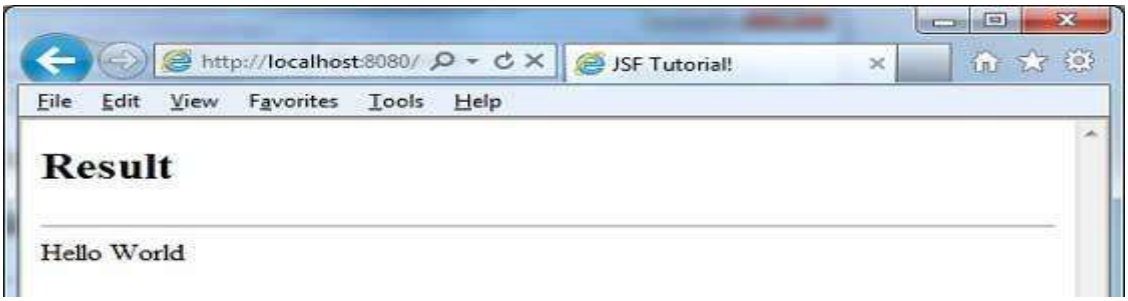
result.xhtml

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

  <h:head>
    <title>JSF Tutorial!</title>
  </h:head>

  <h:body>
    <h2>Result</h2>
    <hr />
    #{userData.data}
  </h:body>
</html>
```

Once you are ready with all the changes done, let us compile and run the application as we did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.



Click any submit button. You will see the following result.



JSF - APPLICATION EVENTS

JSF provides system event listeners to perform application specific tasks during JSF application Life Cycle.

S.No	System Event & Description
1	**PostConstructApplicationEvent** Fires when the application starts. Can be used to perform initialization tasks after the application has started.
2	**PreDestroyApplicationEvent** Fires when the application is about to shut down. Can be used to perform cleanup tasks before the application is about to shut down.
3	**PreRenderViewEvent** Fires before a JSF page is to be displayed. Can be used to authenticate the user and provide restricted access to JSF View.

System Events can be handled in the following manner.

S.No	Technique & Description
1	**SystemEventListener** Implement SystemEventListener interface and register the system-event-listener class in faces-config.xml
2	**Method Binding** Pass the name of the managed bean method in *listener* attribute of f:event.

SystemEventListener

Implement SystemEventListener Interface.

```
public class CustomSystemEventListener implements SystemEventListener {

   @Override
   public void processEvent(SystemEvent event) throws
      AbortProcessingException {

      if(event instanceof PostConstructApplicationEvent) {
         System.out.println("Application Started.
            PostConstructApplicationEvent occurred!");
      }
   }
}
```

Register custom system event listener for system event in faces-config.xml.

```
<system-event-listener>
   <system-event-listener-class>
      com.tutorialspoint.test.CustomSystemEventListener
   </system-event-listener-class>

   <system-event-class>
      javax.faces.event.PostConstructApplicationEvent
   </system-event-class>
</system-event-listener>
```

Method Binding

Define a method

```
public void handleEvent(ComponentSystemEvent event) {
   data = "Hello World";
}
```

Use the above method.

```
<f:event listener = "#{user.handleEvent}" type = "preRenderView" />
```

Example Application

Let us create a test JSF application to test the system events in JSF.

Step	Description
1	Create a project with a name *helloworld* under a package *com.tutorialspoint.test* as explained in the *JSF - First Application* chapter.
2	Modify *UserData.java* file as explained below.
3	Create *CustomSystemEventListener.java* file under a package *com.tutorialspoint.test*. Modify it as explained below
4	Modify *home.xhtml* as explained below.
5	Create *faces-config.xml* in *WEB-INF* folder.Modify it as explained below. Keep the rest of the files unchanged.
6	Compile and run the application to make sure the business logic is working as per the requirements.
7	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.

8	Launch your web application using appropriate URL as explained below in the last step.

UserData.java

```
import java.io.Serializable;

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

@ManagedBean(name = "userData", eager = true)
@SessionScoped
public class UserData implements Serializable {
  private static final long serialVersionUID = 1L;
  private String data = "sample data";

  public void handleEvent(ComponentSystemEvent event) {
    data = "Hello World";
  }

  public String getData() {
    return data;
  }

  public void setData(String data) {
    this.data = data;
  }
}
```

CustomSystemEventListener.java

```
import javax.faces.application.Application;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.PostConstructApplicationEvent;
import javax.faces.event.PreDestroyApplicationEvent;
import javax.faces.event.SystemEvent;
import javax.faces.event.SystemEventListener;

public class CustomSystemEventListener implements SystemEventListener {

  @Override
  public boolean isListenerForSource(Object value) {

    //only for Application
    return (value instanceof Application);
  }

  @Override
  public void processEvent(SystemEvent event)
    throws AbortProcessingException {

    if(event instanceof PostConstructApplicationEvent) {
      System.out.println("Application Started.
        PostConstructApplicationEvent occurred!");
    }
```

```
    if(event instanceof PreDestroyApplicationEvent) {
      System.out.println("PreDestroyApplicationEvent occurred.
        Application is stopping.");
    }
  }
}
```

home.xhtml

```
<?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>JSF tutorial</title>
  </h:head>

  <h:body>
    <h2>Application Events Examples</h2>
    <f:event listener = "#{userData.handleEvent}" type = "preRenderView" />
    #{userData.data}
  </h:body>
</html>
```

faces-config.xhtml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<faces-config
  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-facesconfig_2_0.xsd"
  version = "2.0">

  <application>
    <!-- Application Startup -->
    <system-event-listener>
      <system-event-listener-class>
        com.tutorialspoint.test.CustomSystemEventListener
      </system-event-listener-class>
      <system-event-class>
        javax.faces.event.PostConstructApplicationEvent
      </system-event-class>
    </system-event-listener>

    <!-- Before Application is to shut down -->
    <system-event-listener>
      <system-event-listener-class>
        com.tutorialspoint.test.CustomSystemEventListener
      </system-event-listener-class>
      <system-event-class>
        javax.faces.event.PreDestroyApplicationEvent
```

```
    </system-event-listener>
  </application>
</faces-config>
```

Once you are ready with all the changes done, let us compile and run the application as we did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.

Look into your web-server console output. You will see the following result.

```
INFO: Deploying web application archive helloworld.war
Dec 6, 2012 8:21:44 AM com.sun.faces.config.ConfigureListener contextInitialized

INFO: Initializing Mojarra 2.1.7 (SNAPSHOT 20120206) for context '/helloworld'
Application Started. PostConstructApplicationEvent occurred!
Dec 6, 2012 8:21:46 AM com.sun.faces.config.ConfigureListener
$WebConfigResourceMonitor$Monitor <init>

INFO: Monitoring jndi:/localhost/helloworld/WEB-INF/faces-config.xml
for modifications
Dec 6, 2012 8:21:46 AM org.apache.coyote.http11.Http11Protocol start

INFO: Starting Coyote HTTP/1.1 on http-8080
Dec 6, 2012 8:21:46 AM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009

Dec 6, 2012 8:21:46 AM org.apache.jk.server.JkMain start
INFO: Jk running ID = 0 time = 0/24  config = null
Dec 6, 2012 8:21:46 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 44272 ms
```

JSF PAGE NAVIGATION

Introduction

In simple language Page Navigation means flow of the application from one page to another page. Navigation in JSF defines the set of rules for choosing the next view to be displayed after a specified action is completed. In JSF 1.x, navigation is generally specified using the faces-config.xml file, but in JSF 2.0 that configuration file is optional. JSF2.0 gives new feature of Implicit Navigation . So in JSF2.0 we can use page navigation in two ways either implicit navigation or through configuration files. we will discuss both approach in detail.

Consider the scenario in which user has to enter his/her details in an online form.The form is spllitted into two pages page1.jsp and page2.jsp. After filling the form1 details user wants to move to second form

<h:commandLink value="Next" action="page2"/>

When this link is selected, navigation should proceed from page1.jsp to page22.jsp (assume that views are defined using JSP). This navigation rule should be defined in faces-config.xml as follows:

```
<navigation-rule>
 <from-view-id>/page1.jsp</from-view-id>
  <navigation-case>
  <from-outcome>Address 2</from-outcome>
 <to-view-id>/page2.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

In JavaServer Faces, navigation between application pages is defined by a set of rules. Navigation rules determine the next page to display when a user clicks a navigation component, such as a button or a hyperlink.These navigation rules are defined in JSF configuration files along with other definitions for a JSF application. Usually, this file is named faces-config.xml. However, you are free to assign any other name. You can use more than one file to store JSF configuration data.

We need to specify configuration files in your web.xml file. Use below line of codes in your web.xml file to declare configuration files:

```
<context-param>
<paramname>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml,/WEB-INF/faces-beans.xml</param-value>
</context-param>
```

How to Create Page Navigation Rules

Navigation rule definitions are stored in the JSF configuration file (faces-config.xml). You can define the rules directly in the configuration file, or you can use the JSF Navigation Modeler in eclipse.

The general syntax of a JSF navigation rule element in the faces-config.xml file is shown below.

```
<navigation-rule>
  <from-view-id>page-or-pattern</from-view-id>
  <navigation-case>
    <from-action>action-method</from-action>
    <from-outcome>outcome</from-outcome>
    <to-view-id>destination-page</to-view-id>
    <redirect/>
  </navigation-case>
  <navigation-case>
    ...
  </navigation-case>
</navigation-rule>
```

- *navigation-rule:* A mandatory element for navigation case elements.
- *from-view-id:* An optional element that contains either a complete page identifier or a page identifier prefix ending with the asterisk (*) wildcard character. If you use the wildcard character, the rule applies to all pages that match the wildcard pattern. To make a global rule that applies to all pages, leave this element blank.
- *navigation-case:* A mandatory element for each case in the navigation rule. Each case defines the different navigation paths from the same page. A navigation rule must have at least one navigation case.
- *from-action:* An optional element that limits the application of the rule only to outcomes from the specified action method.
- *from-outcome:* A mandatory element that contains an outcome value that is matched against values specified in the action attribute of UI components. Later you will see how the outcome value is referenced in a UI component either explicitly or dynamically through an action method return.
- *to-view-id:* A mandatory element that contains the complete page identifier of the page to which the navigation is routed when the rule is implemented.
- *redirect:* An optional element that indicates that the new view is to be requested through a redirect response instead of being rendering as the response to the current request. This element requires no value.

A sample navigation example

```
<navigation-rule>
   <from-view-id>/pages/index.jsp</from-view-id>
   <navigation-case>
     <from-outcome>success</from-outcome>
     <to-view-id>/pages/welcome.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
     <from-outcome>failure</from-outcome>
     <to-view-id>/pages/error.jsp</to-view-id>
   </navigation-case>
 </navigation-rule>
```

This code specifies that view /pages/index.jsp has two outputs, success and failure, associated with particular pages.

Managed Bean for above configuration is given bellow.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;
 @ManagedBean
@SessionScoped
public class PageController implements Serializable {

private static final long serialVersionUID = 1L;

public String processPage1(){
return "success";
```

```
public String processPage2(){
return "failure";
}
}
```

Implicit Navigation In JSF 2.0 :

Implicit Navigation requires no entry in the configuration file. It allows the default navigation handler to choose the XHTML page with the matching name as specified in the action attribute of the UIComponent.

In JSF 2, it treats "outcome" as the page name, for example, navigate to "page1.xhtml", you have to put the "outcome" as "page1". This mechanism is called "Implicit Navigation", where no need to declare the navigation rule, instead, just put the "outcome" in the action attribute directly and JSF will find the correct "view id" automatically.

There are two ways to implements the implicit navigation in JSF 2.

1. By using Outcome in JSF page
2. By using Outcome in Managed Bean

1. Outcome in JSF page

In this implicit navigation approach we can directly provide the next flow page in action attribute of JSF 2.0 tags. For example,below code we have one submit button in page1.xhtml, when we click that button we should go to page2.xhtml page

```
1<h:form>
2<h:commandButton action="page2" value="Next"/>
3</h:form>
```

Once the button is clicked, JSF will merge the action value or outcome, "page2" with "xhtml" extension, and find the view name "page2.xhtml" in the current "page1.xhtml" directory.

2. Outcome in Managed Bean :

We can also define the "outcome" in a managed bean like this :

```
@ManagedBean
@SessionScoped

public class PageController implements Serializable {

public String moveToPage2(){
return "page2"; //outcome

}
}
```

In JSF page, action attribute, just call the method by using "method expression".

```
1 <h:form>
2 <h:commandButton action="#{pageController.moveToPage2}" value="Move to page2.xhtml by managed bean"/>
3 </h:form>
```

JSF implicit navigation example

page1.xhtml

```
<?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:body>

     <h2>This is page1.xhtml</h2>

      <h:form>

        <h:commandButton action="page2" value="Go to Page2" />

        <br></br>

        <p><b>Move to page2 by managed bean</b></p>

        <h:commandButton action="#{pageController.moveToPage2}"

            value="Go to Page2" />

     </h:form>

    </h:body>

</html>
```

page2.xhtml

```
<?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:body>

     <h2>This is page2.xhtml</h2>
```

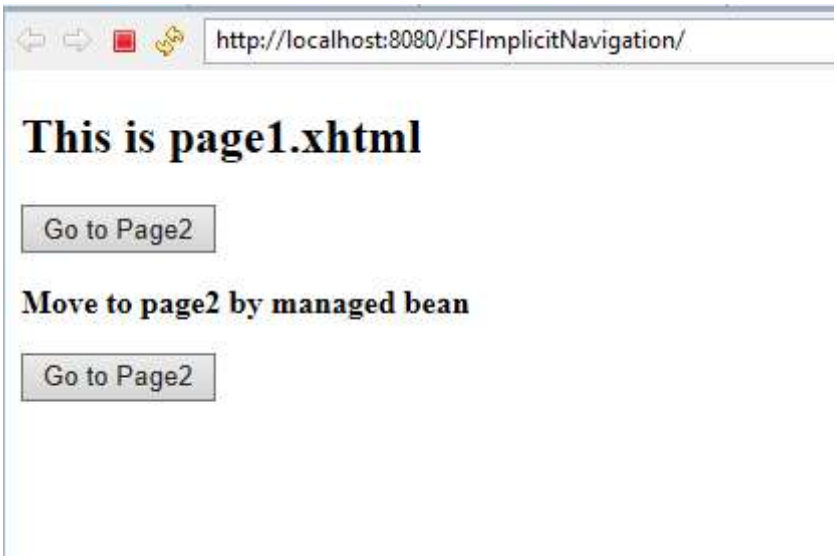
```
        <h:form>
          <h:commandButton action="page1" value="Go to Page1" />
          <br></br>
          <p><b>Move to page1 by managed bean</b></p>
          <h:commandButton action="#{pageController.moveToPage1}" value="Go to Page1" />
       </h:form>
     </h:body>
</html>
```

Create Managed Bean

PageController.java

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
  import java.io.Serializable;
  @ManagedBean
@SessionScoped
public class PageController implements Serializable {
     private static final long serialVersionUID = 1L;
     public String moveToPage1(){
       return "page1";
   }
     public String moveToPage2(){
       return "page2";
   }
}
```

Output


http://localhost:8080/JSFImplicitNavigation/
This is page1.xhtml
Go to Page2
Move to page2 by managed bean
Go to Page2


http://localhost:8080/JSFImplicitNavigation/faces/page1.xhtml
This is page2.xhtml
Go to Page1
Move to page1 by managed bean
Go to Page1