JSF

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Java Server Faces (JSF) is a Java-based web application framework intended to simplify development integration of web-based user interfaces. JavaServer Faces is a standardized display technology, which was formalized in a specification through the Java Community Process.

This tutorial will teach you basic JSF concepts and will also take you through various advance concepts related to JSF framework.

Audience

This tutorial has been prepared for the beginners to help them understand basic JSF programming. After completing this tutorial, you will find yourself at a moderate level of expertise in JSF programming from where you can take yourself to the next levels.

Prerequisites

Before proceeding with this tutorial you should have a basic understanding of Java programming language, text editor, and execution of programs etc. Since we are going to develop web-based applications using JSF, it will be good if you have an understanding of other web technologies such as HTML, CSS, AJAX, etc.

Copyright & Disclaimer

Table of Contents

About the Tutorial i

Audience i

Prerequisites i

Copyright & Disclaimer i

Table of Contents ii

1. JSF – OVERVIEW 1

What is JSF? 1

Benefits 1

JSF UI Component Model 1

2. JSF – ENVIRONMENTAL SETUP 2

System Requirement 2

Environment Setup for JSF Application Development 2

3. JSF – ARCHITECTURE 10

What is MVC Design Pattern? 10

JSF Architecture 10

4. JSF – LIFE CYCLE 12

Phase 1: Restore view 12

Phase 2: Apply request values 12

Phase 3: Process validation 13

Phase 4: Update model values 13

Phase 5: Invoke application 13

Phase 6: Render response 13

5. JSF – FIRST APPLICATION 14

Create Project 14

Add JSF Capability to Project 15

Prepare Eclipse Project 17

Import Project in Eclipse 19

Configure Faces Servlet in web.xml 19

Create a Managed Bean 21

Create a JSF page 21

Build the Project 22

Deploy WAR file 23

Run Application 23

6. JSF – MANAGED BEANS 25

@ManagedBean Annotation 26

Scope Annotations 26

@ManagedProperty Annotation 26

7. JSF – PAGE NAVIGATION 30

Implicit Navigation 30

Auto Navigation in JSF Page 30

Auto Navigation in Managed Bean 31

Conditional Navigation 32

Resolving Navigation Based on from-action 34

Forward vs Redirect 36

8. JSF – BASIC TAGS 46

h:inputText 48

h:inputSecret 55

h:inputTextarea 60

h:inputHidden 66

h:selectBooleanCheckbox 72

h:selectManyCheckbox 79

h:selectOneRadio 88

h:selectOneListbox 97

h:selectManyListbox 106

h:selectOneMenu 115

h:outputText 124

h:outputFormat 126

h:graphicImage 129

h:outputStylesheet 133

h:outputScript 138

h:commandButton 143

h:Link 148

h:commandLink 155

h:outputLink 161

h:panelGrid 168

h:message 179

h:messages 185

f:param 190

f:attribute 196

h:setPropertyActionListener 200

9. JSF – FACELET TAGS 205

Template Tags 206

Creating Template 207

ui:param Tag 214

Parameter to Section of a Template 215

Parameter to Template 215

Custom Tag 219

ui:remove Tag 226

10. JSF - CONVERTOR TAGS 229

f:convertNumber 229

f:convertDateTime 233

Custom Converter 238

11. JSF - VALIDATOR TAGS 247

f:validateLength 248

f:validateLongRange 252

f:validateDoubleRange 255

f:validateRegex 259

Custom Validator 264

12. JSF – DATATABLE 273

Display DataTable 274

Add Data to DataTable 285

Edit Data of a DataTable 290

Delete Data of a DataTable 294

Using DataModel in a DataTable 297

13. JSF – COMPOSITE COMPONENTS 301

Define Custom Component 301

Use Custom Component 303

14. JSF – AJAX 309

15. JSF – EVENT HANDLING 315

valueChangeListener **316**

actionListener **324**

Application Events **331**

16. JSF - JDBC INTEGRATION.. 339

17. JSF - SPRING INTEGRATION.. 349

18. JSF - EXPRESSION LANGUAGE 360

19. JSF - INTERNATIONALIZATION 364

1. JSF – OVERVIEW

What is JSF?

JavaServer Faces (JSF) is a MVC web framework that simplifies the construction of User Interfaces (UI) for server-based applications using reusable UI components in a page. JSF provides a facility to connect UI widgets with data sources and to server-side event handlers. The JSF specification defines a set of standard UI components and provides an Application Programming Interface (API) for developing components. JSF enables the reuse and extension of the existing standard UI components.

Benefits

JSF reduces the effort in creating and maintaining applications, which will run on a Java application server and will render application UI on to a target client. JSF facilitates Web application development by -

- Providing reusable UI components
- Making easy data transfer between UI components
- Managing UI state across multiple server requests
- Enabling implementation of custom components
- Wiring client-side event to server-side application code

JSF UI Component Model

JSF provides the developers with the capability to create Web application from collections of UI components that can render themselves in different ways for multiple client types (for example - HTML browser, wireless, or WAP device).

JSF provides -

- Core library
- A set of base UI components - standard HTML input elements
- Extension of the base UI components to create additional UI component libraries or to extend existing components
- Multiple rendering capabilities that enable JSF UI components to render themselves differently depending on the client types

2. JSF – ENVIRONMENTAL SETUP

This chapter will guide you on how to prepare a development environment to start your work with JSF Framework. You will learn how to setup JDK, Eclipse, Maven, and Tomcat on your machine before you set up JSF Framework.

System Requirement

JSF requires JDK 1.5 or higher so the very first requirement is to have JDK installed on your machine.

JDK	1.5 or above
Memory	No minimum requirement
Disk Space	No minimum requirement
Operating System	No minimum requirement

Environment Setup for JSF Application Development

Follow the given steps to setup your environment to start with JSF application development.

Step 1: Verify Java installation on your machine.

Open console and execute the following **Java** command.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:~ joseph$ java -version

Let's verify the output for all the operating systems:

OS	Generated Output
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

Step 2: Set Up Java Development Kit (JDK).

If you do not have Java installed then you can install the Java Software Development Kit (SDK) from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally, set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine.

For example -

OS	Output

Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	Export JAVA_HOME=/usr/local/java-current
Mac	Export JAVA_HOME=/Library/Java/Home

Append Java compiler location to System Path.

OS	Output
Windows	Append the string ;%JAVA_HOME%\bin to the end of the system variable, Path.
Linux	Export PATH=$PATH:$JAVA_HOME/bin/
Mac	Not required

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java. Otherwise, carry out a proper setup according to the given document of the IDE.

Step 3: Set Up Eclipse IDE.

All the examples in this tutorial have been written using Eclipse IDE. Hence, we suggest you should have the latest version of Eclipse installed on your machine based on your operating system.

To install Eclipse IDE, download the latest Eclipse binaries with WTP support from http://www.eclipse.org/downloads/. Once you download the installation, unpack the binary distribution into a convenient location. For example, in C:\eclipse on Windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

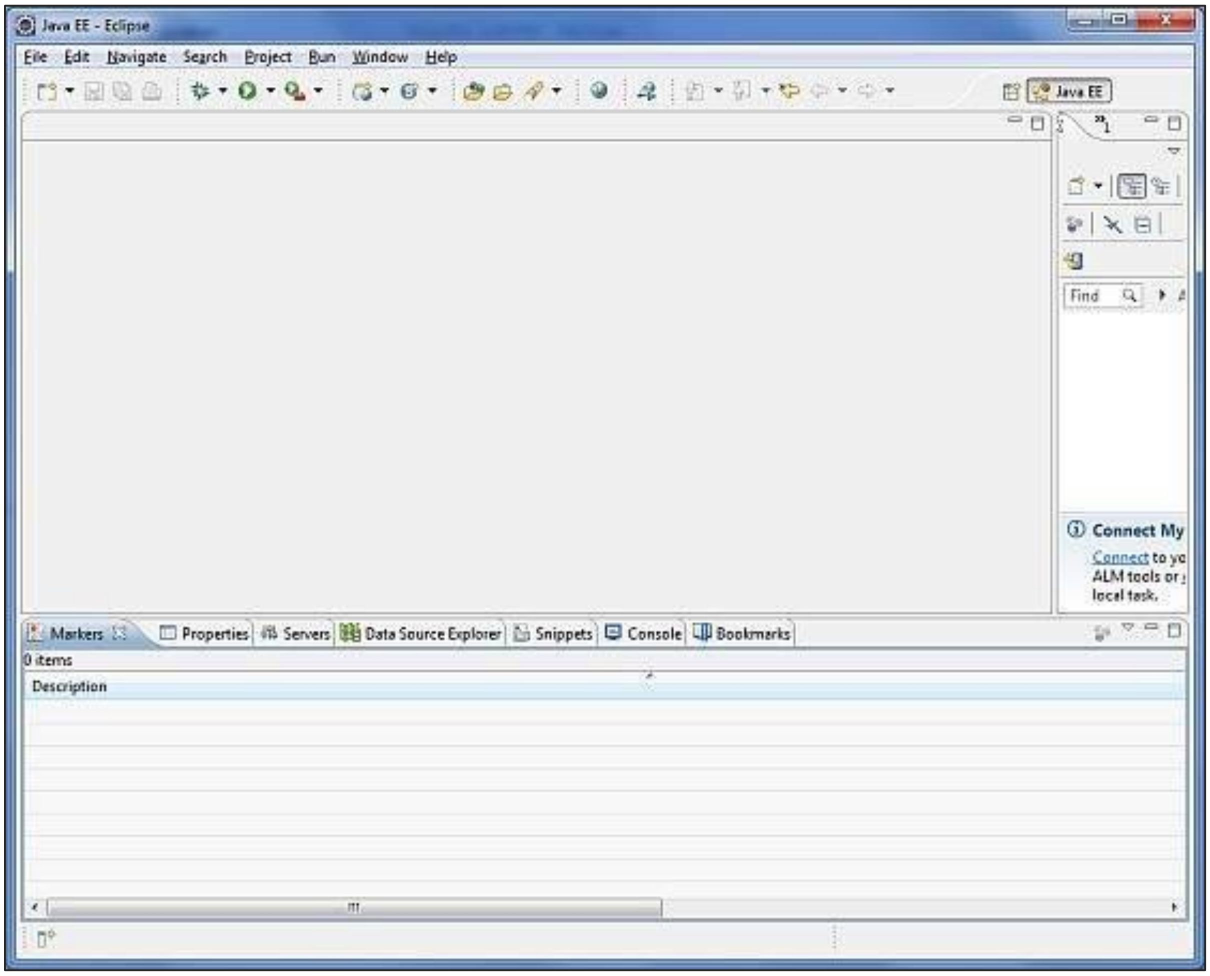
Eclipse can be started by executing the following commands on Windows machine, or you can simply double-click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it will display the following result.



***Note:** Install m2eclipse plugin to eclipse using the following eclipse software update site

m2eclipse Plugin - http://m2eclipse.sonatype.org/update/

This plugin enables the developers to run maven commands within eclipse with embedded/external maven installation.

Step 4: Download Maven archive.

Download Maven 2.2.1 from http://maven.apache.org/download.html

OS	Archive name
Windows	apache-maven-2.0.11-bin.zip
Linux	apache-maven-2.0.11-bin.tar.gz
Mac	apache-maven-2.0.11-bin.tar.gz

Step 5: Extract the Maven archive.

Extract the archive to the directory you wish to install Maven 2.2.1. The subdirectory apache-maven-2.2.1 will be created from the archive.

OS	Location (can be different based on your installation)
Windows	C:\Program Files\Apache Software Foundation\apache-maven-2.2.1
Linux	/usr/local/apache-maven
Mac	/usr/local/apache-maven

Step 6: Set Maven environment variables.

Add M2_HOME, M2, MAVEN_OPTS to environment variables.

OS	Output
Windows	Set the environment variables using system properties.

	M2_HOME=C:\Program Files\Apache Software Foundation\apache-maven-2.2.1 *M2=%M2_HOME%\bin* *MAVEN_OPTS=-Xms256m -Xmx512m*
Linux	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-2.2.1* *export M2=%M2_HOME%\bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*
Mac	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-2.2.1* *export M2=%M2_HOME%\bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*

Step 7: Add Maven bin directory location to system path.

Now append M2 variable to System Path.

OS	Output
Windows	Append the string ;%M2% to the end of the system variable, Path.
Linux	export PATH=$M2:$PATH
Mac	export PATH=$M2:$PATH

Step 8: Verify Maven installation.

Open console, execute the following **mvn** command.

OS	Task	Command
Windows	Open Command Console	c:\> mvn --version
Linux	Open Command Terminal	$ mvn --version
Mac	Open Terminal	machine:~ joseph$ mvn --version

Finally, verify the output of the above commands, which should be as shown in the following table.

OS	Output
Windows	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre
Linux	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre
Mac	Apache Maven 2.2.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre

Step 9: Set Up Apache Tomcat.

You can download the latest version of Tomcat from http://tomcat.apache.org/. Once you download the installation, unpack the binary distribution into a convenient location. For example, in C:\apache-tomcat-6.0.33 on Windows, or /usr/local/apache-tomcat-6.0.33 on Linux/Unix and set CATALINA_HOME environment variable pointing to the installation locations.

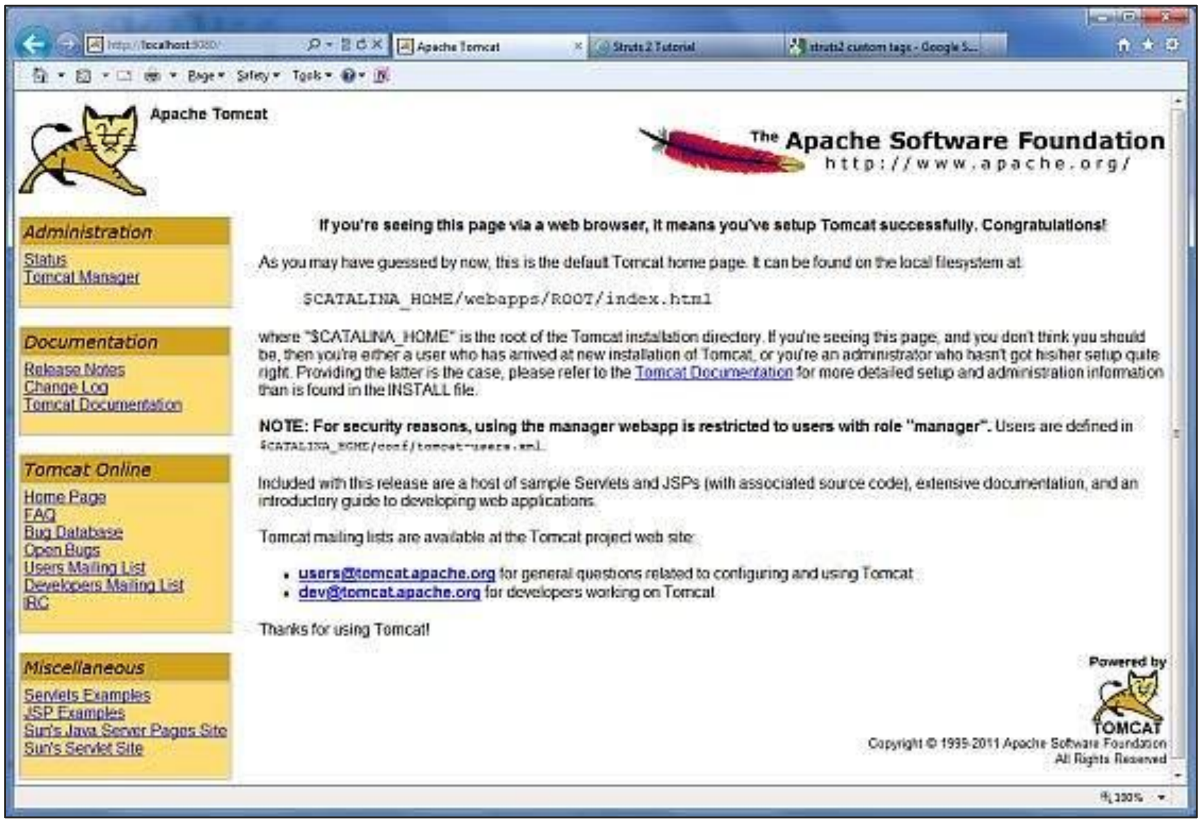
Tomcat can be started by executing the following commands on Windows machine, or you can simply double-click on startup.bat

```
%CATALINA_HOME%\bin\startup.bat


or


C:\apache-tomcat-6.0.33\bin\startup.bat
```

Tomcat can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine.

```
$CATALINA_HOME/bin/startup.sh


or


/usr/local/apache-tomcat-6.0.33/bin/startup.sh
```

After a successful startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**. If everything is fine, then it will display the following result.



If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.html
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

NOTE: For security reasons, using the manager webapp is restricted to users with role "manager". Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Tomcat project web site:

- users@tomcat.apache.org for general questions related to configuring and using Tomcat
- dev@tomcat.apache.org for developers working on Tomcat

Thanks for using Tomcat!

Copyright © 1999-2011 Apache Software Foundation
All Rights Reserved

Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site: http://tomcat.apache.org

Tomcat can be stopped by executing the following commands on Windows machine.

```
%CATALINA_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown
```

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine.

```
$CATALINA_HOME/bin/shutdown.sh


or


/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

3. 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:

Module	Description
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.

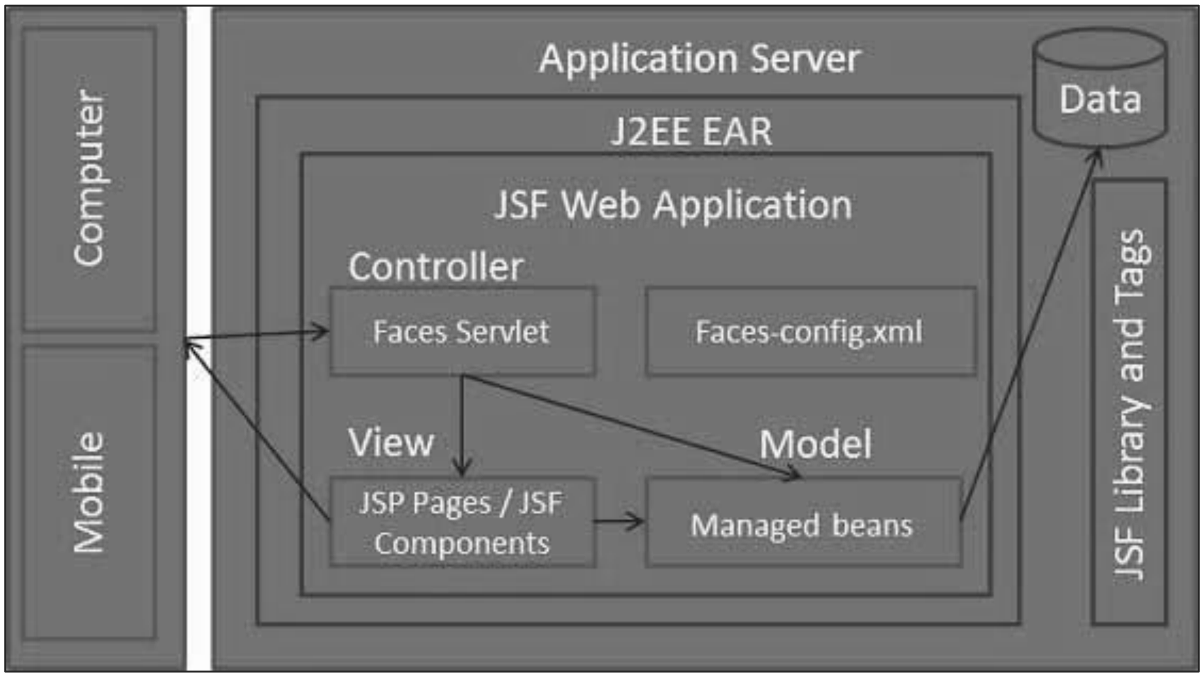
Web designers have to concentrate only on view layer rather than model and controller layer. Developers can change the code for model and typically need not change view layer. Controllers are used to process user actions. In this process, layer model and views may be changed.

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



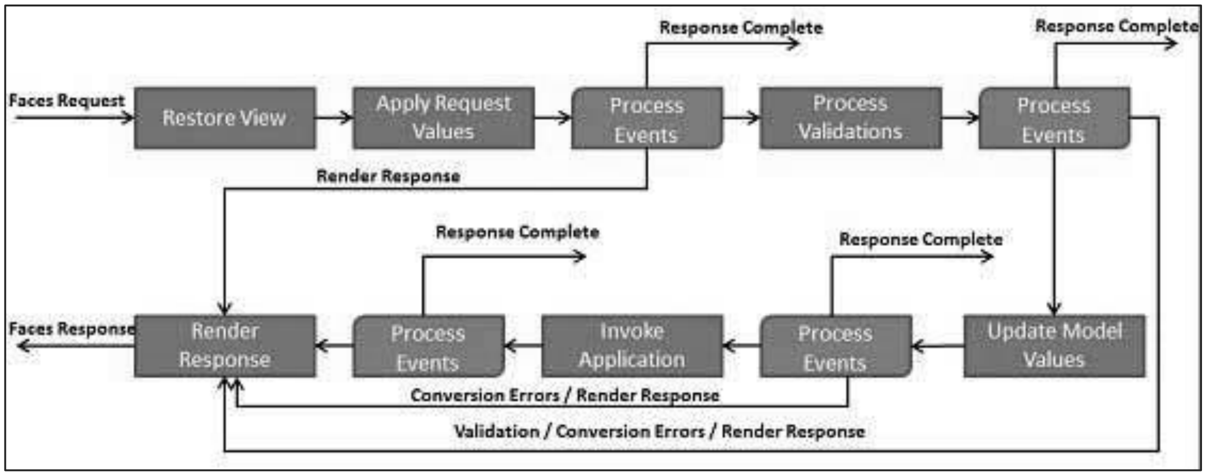
There are controllers which can be used to perform user actions. UI can be created by web page authors and the business logic can be utilized by managed beans.

JSF provides several mechanisms for rendering an individual component. It is up to 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.

4. JSF – LIFE CYCLE

JSF application life cycle consists of six phases which are as follows -

- Restore view phase
- Apply request values phase; process events
- Process validations phase; process events
- Update model values phase; process events
- Invoke application phase; process events
- Render response phase



The six phases show the order in which JSF processes a form. The list shows the phases in their likely order of execution with event processing at each phase.

Phase 1: Restore view

JSF begins the restore view phase as soon as a link or a button is clicked and JSF receives a request.

During this phase, JSF builds the view, wires event handlers and validators to UI components and saves the view in the FacesContext instance. The FacesContext instance will now contain all the information required to process a request.

Phase 2: Apply request values

After the component tree is created/restored, each component in the component tree uses the decode method to extract its new value from the request parameters. Component stores this value. If the conversion fails, an error message is generated and queued on FacesContext.

This message will be displayed during the render response phase, along with any validation errors.

If any decode methods event listeners called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Phase 3: Process validation

During this phase, JSF processes all validators registered on the component tree. It examines the component attribute rules for the validation and compares these rules to the local value stored for the component.

If the local value is invalid, JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and displays the same page again with the error message.

Phase 4: Update model values

After the JSF checks that the data is valid, it walks over the component tree and sets the corresponding server-side object properties to the components' local values. JSF will update the bean properties corresponding to the input component's value attribute.

If any updateModels methods called renderResponse on the current FacesContext instance, JSF moves to the render response phase.

Phase 5: Invoke application

During this phase, JSF handles any application-level events, such as submitting a form/linking to another page.

Phase 6: Render response

During this phase, JSF asks container/application server to render the page if the application is using JSP pages. For initial request, the components represented on the page will be added to the component tree as JSP container executes the page. If this is not an initial request, the component tree is already built so components need not be added again. In either case, the components will render themselves as the JSP container/Application server traverses the tags in the page.

After the content of the view is rendered, the response state is saved so that subsequent requests can access it and it is available to the restore view phase.

5. JSF – FIRST APPLICATION

To create a simple JSF application, we'll use maven-archetype-webapp plugin. In the following example, we'll create a maven-based web application project in C:\JSF folder.

Create Project

Let's open command console, go the **C:\ > JSF** directory and execute the following **mvn** command.

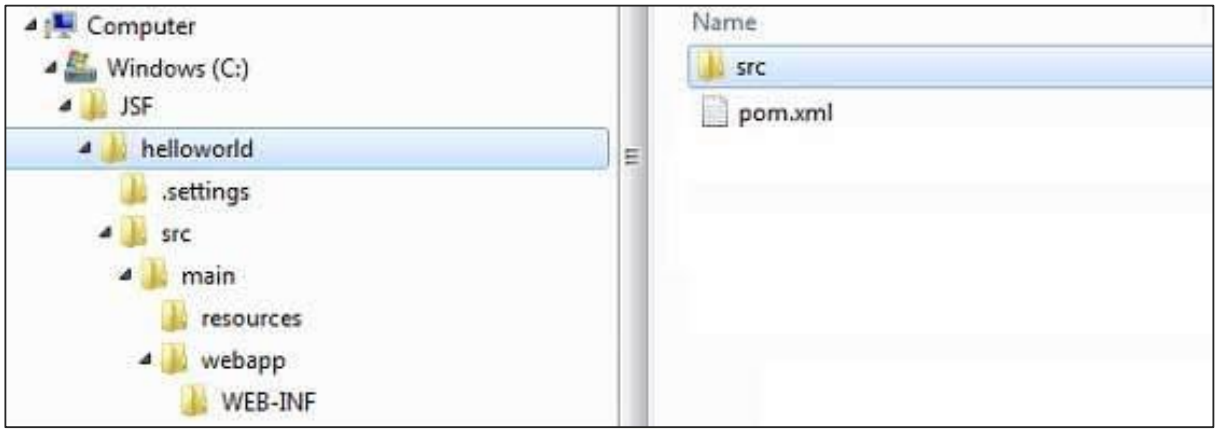
```
C:\JSF>mvn archetype:create
-DgroupId=com.tutorialspoint.test
-DartifactId=helloworld
-DarchetypeArtifactId=maven-archetype-webapp
```

Maven will start processing and will create the complete java web application project structure.

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] -------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] -------------------------------------------------------------
[INFO] [archetype:create {execution: default-cli}]
[INFO] Defaulting package to group ID: com.tutorialspoint.test
[INFO] artifact org.apache.maven.archetypes:maven-archetype-webapp:
checking for updates from central
[INFO] -------------------------------------------------------------
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-webapp:RELEASE
[INFO] -------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.tutorialspoint.test
[INFO] Parameter: packageName, Value: com.tutorialspoint.test
[INFO] Parameter: package, Value: com.tutorialspoint.test
```

```
[INFO] Parameter: artifactId, Value: helloworld
[INFO] Parameter: basedir, Value: C:\JSF
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir:
C:\JSF\helloworld
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 7 seconds
[INFO] Finished at: Mon Nov 05 16:05:04 IST 2012
[INFO] Final Memory: 12M/84M
[INFO] ------------------------------------------------------------
```

Now go to C:/JSF directory. You'll see a Java web application project created, named helloworld (as specified in artifactId). Maven uses a standard directory layout as shown in the following screenshot.



Using the above example, we can understand the following key concepts.

Folder Structure	Description
helloworld	Contains src folder and pom.xml
src/main/wepapp	Contains WEB-INF folder and index.jsp page
src/main/resources	It contains images/properties files (In the above example, we need to create this structure manually)

Add JSF Capability to Project

Add the following JSF dependencies.

```
<dependencies>
   <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>2.1.7</version>
   </dependency>

   <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>2.1.7</version>
   </dependency>

</dependencies>
```

Complete POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">

   <modelVersion>4.0.0</modelVersion>
   <groupId>com.tutorialspoint.test</groupId>
   <artifactId>helloworld</artifactId>
   <packaging>war</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>helloworld Maven Webapp</name>
   <url>http://maven.apache.org</url>

   <dependencies>
```

```
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
      </dependency>

      <dependency>
         <groupId>com.sun.faces</groupId>
         <artifactId>jsf-api</artifactId>
         <version>2.1.7</version>
      </dependency>

      <dependency>
         <groupId>com.sun.faces</groupId>
         <artifactId>jsf-impl</artifactId>
         <version>2.1.7</version>
      </dependency>

   </dependencies>

   <build>
      <finalName>helloworld</finalName>

      <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3.1</version>

            <configuration>
               <source>1.6</source>
               <target>1.6</target>
            </configuration>
```

```
          </plugin>

       </plugins>

    </build>

</project>
```

Prepare Eclipse Project

Let's open the command console. Go the **C:\ > JSF > helloworld** directory and execute the following **mvn** command.

```
C:\JSF\helloworld>mvn eclipse:eclipse -Dwtpversion=2.0
```

Maven will start processing, create the eclipse ready project, and will add wtp capability.

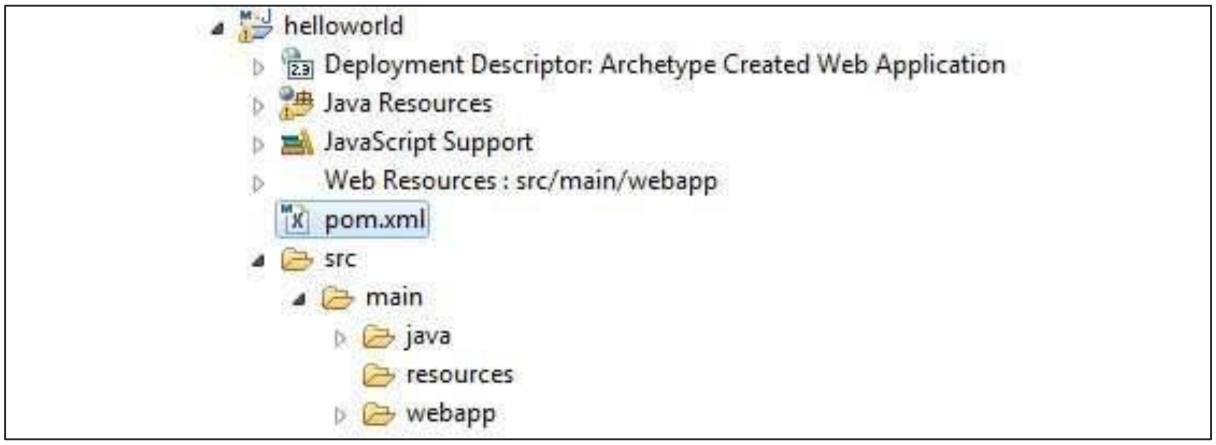
```
Downloading: http://repo.maven.apache.org/org/apache/maven/plugins/
maven-compiler-plugin/2.3.1/maven-compiler-plugin-2.3.1.pom
5K downloaded  (maven-compiler-plugin-2.3.1.pom)
Downloading: http://repo.maven.apache.org/org/apache/maven/plugins/
maven-compiler-plugin/2.3.1/maven-compiler-plugin-2.3.1.jar
29K downloaded  (maven-compiler-plugin-2.3.1.jar)
[INFO] Searching repository for plugin with prefix: 'eclipse'.
[INFO] ------------------------------------------------------------
[INFO] Building helloworld Maven Webapp
[INFO]    task-segment: [eclipse:eclipse]
[INFO] ------------------------------------------------------------
[INFO] Preparing eclipse:eclipse
[INFO] No goals needed for project - skipping
[INFO] [eclipse:eclipse {execution: default-cli}]
[INFO] Adding support for WTP version 2.0.
[INFO] Using Eclipse Workspace: null
[INFO] Adding default classpath container: org.eclipse.jdt.
launching.JRE_CONTAINER
Downloading: http://repo.maven.apache.org/
```

```
com/sun/faces/jsf-api/2.1.7/jsf-api-2.1.7.pom
12K downloaded  (jsf-api-2.1.7.pom)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-impl/2.1.7/jsf-impl-2.1.7.pom
10K downloaded  (jsf-impl-2.1.7.pom)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-api/2.1.7/jsf-api-2.1.7.jar
619K downloaded  (jsf-api-2.1.7.jar)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-impl/2.1.7/jsf-impl-2.1.7.jar
1916K downloaded  (jsf-impl-2.1.7.jar)
[INFO] Wrote settings to C:\JSF\helloworld\.settings\
org.eclipse.jdt.core.prefs
[INFO] Wrote Eclipse project for "helloworld" to C:\JSF\helloworld.
[INFO]
[INFO] -----------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -----------------------------------------------------------
[INFO] Total time: 6 minutes 7 seconds
[INFO] Finished at: Mon Nov 05 16:16:25 IST 2012
[INFO] Final Memory: 10M/89M
[INFO] -----------------------------------------------------------
```

Import Project in Eclipse

Following are the steps:

- Import project in eclipse using Import wizard.
- Go to **File -> Import... -> Existing project into workspace**.
- Select root directory to helloworld.
- Keep **Copy projects into workspace** to be checked.
- Click Finish button.
- Eclipse will import and copy the project in its workspace **C:\ -> Projects -> Data -> WorkSpace.**


helloworld
Deployment Descriptor: Archetype Created Web Application
Java Resources
JavaScript Support
Web Resources : src/main/webapp
pom.xml
src
main
java
resources
webapp

Configure Faces Servlet in web.xml

Locate web.xml in **webapp -> WEB-INF** folder and update it as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   id="WebApp_ID" version="2.5">

   <welcome-file-list>
      <welcome-file>faces/home.xhtml</welcome-file>
   </welcome-file-list>

   <!--
      FacesServlet is main servlet responsible to handle all request.
      It acts as central controller.
      This servlet initializes the JSF components before the JSP is displayed.
   -->

   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>

   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.jsf</url-pattern>
   </servlet-mapping>

   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.faces</url-pattern>
   </servlet-mapping>

   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.xhtml</url-pattern>
   </servlet-mapping>

</web-app>
```

Create a Managed Bean

Create a package structure under **src -> main -> java** as **com -> tutorialspoint -> test**. Create HelloWorld.java class in this package. Update the code of **HelloWorld.java** as shown below.

```
package com.tutorialspoint.test;

import javax.faces.bean.ManagedBean;

@ManagedBean(name = "helloWorld", eager = true)
public class HelloWorld {
   public HelloWorld() {
      System.out.println("HelloWorld started!");
   }

   public String getMessage() {
      return "Hello World!";
   }
}
```

Create a JSF page

Create a page home.xhtml under **webapp** folder. Update the code of **home.xhtml** as shown below.

```
<!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">
<head>
   <title>JSF Tutorial!</title>
</head>

<body>
   #{helloWorld.getMessage()}
</body>
</html>
```

Build the Project

Following are the steps.

- Select helloworld project in eclipse
- Use Run As wizard
- Select **Run As -> Maven package**

- Maven will start building the project and will create helloworld.war under **C:\ -> Projects -> Data -> WorkSpace -> helloworld -> target** folder.

```
[INFO] Scanning for projects...
[INFO] -----------------------------------------------------
[INFO] Building helloworld Maven Webapp
[INFO]
[INFO] Id: com.tutorialspoint.test:helloworld:war:1.0-SNAPSHOT
[INFO] task-segment: [package]
[INFO] -----------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] Surefire report directory:
C:\Projects\Data\WorkSpace\helloworld\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
There are no tests to run.

Results :
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0
[INFO] [war:war]
[INFO] Packaging webapp
[INFO] Assembling webapp[helloworld] in
[C:\Projects\Data\WorkSpace\helloworld\target\helloworld]
[INFO] Processing war project
[INFO] Webapp assembled in[150 msecs]
[INFO] Building war:
```

```
C:\Projects\Data\WorkSpace\helloworld\target\helloworld.war
[INFO] ------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Mon Nov 05 16:34:46 IST 2012
[INFO] Final Memory: 2M/15M
[INFO] ------------------------------------------------
```

Deploy WAR file

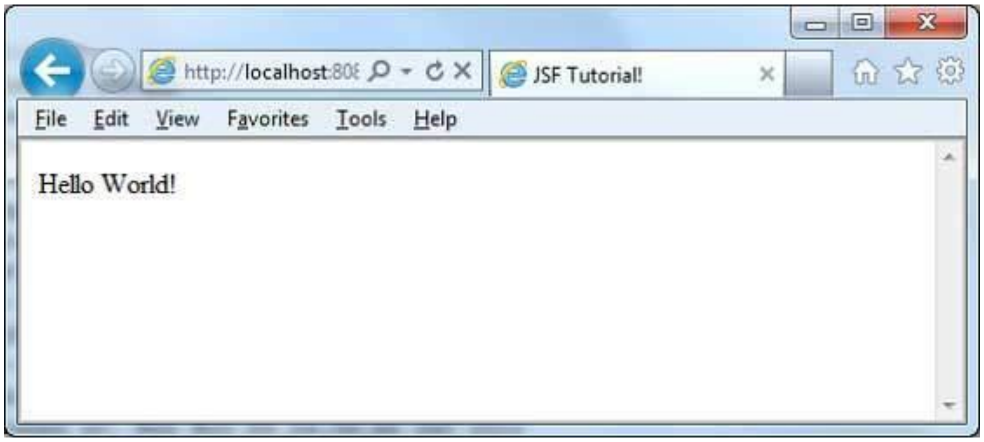
Following are the steps.

- Stop the tomcat server.
- Copy the helloworld.war file to **tomcat installation directory -> webapps folder**.
- Start the tomcat server.
- Look inside webapps directory, there should be a folder **helloworld** got created.
- Now helloworld.war is successfully deployed in Tomcat Webserver root.

Run Application

Enter a url in web browser: **http://localhost:8080/helloworld/home.jsf** to launch the application.

Server name (localhost) and port (8080) may vary as per your tomcat configuration.



Hello World!

6. JSF – MANAGED BEANS

Managed Bean is a regular Java Bean class registered with JSF. In other words, Managed Beans is a Java bean managed by JSF framework. Managed bean contains the getter and setter methods, business logic, or even a backing bean (a bean contains all the HTML form value).

Managed beans works as Model for UI component. Managed Bean can be accessed from JSF page.

In **JSF 1.2,** a managed bean had to register it in JSF configuration file such as faces-config.xml. From **JSF 2.0** onwards, managed beans can be easily registered using annotations. This approach keeps beans and its registration at one place hence it becomes easier to manage.

Using XML Configuration

```
<managed-bean>
  <managed-bean-name>helloWorld</managed-bean-name>
  <managed-bean-class>com.tutorialspoint.test.HelloWorld</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>message</managed-bean-name>
  <managed-bean-class>com.tutorialspoint.test.Message</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Using Annotation

```
@ManagedBean(name = "helloWorld", eager = true)
@RequestScoped
public class HelloWorld {

   @ManagedProperty(value="#{message}")
   private Message message;
   ...
}
```

@ManagedBean Annotation

@ManagedBean marks a bean to be a managed bean with the name specified in **name** attribute. If the name attribute is not specified, then the managed bean name will default to class name portion of the fully qualified class name. In our case, it would be helloWorld.

Another important attribute is **eager**. If eager="true" then managed bean is created before it is requested for the first time otherwise "lazy" initialization is used in which bean will be created only when it is requested.

Scope Annotations

Scope annotations set the scope into which the managed bean will be placed. If the scope is not specified, then bean will default to request scope. Each scope is briefly discussed in the following table.

Scope	Description
@RequestScoped	Bean lives as long as the HTTP request-response lives. It gets created upon a HTTP request and gets destroyed when the HTTP response associated with the HTTP request is finished.
@NoneScoped	Bean lives as long as a single EL evaluation. It gets created upon an EL evaluation and gets destroyed immediately after the EL evaluation.
@ViewScoped	Bean lives as long as the user is interacting with the same JSF view in the browser window/tab. It gets created upon a HTTP request and gets destroyed once the user postbacks to a different view.
@SessionScoped	Bean lives as long as the HTTP session lives. It gets created upon the first HTTP request involving this bean in the session and gets destroyed when the HTTP session is invalidated.
@ApplicationScoped	Bean lives as long as the web application lives. It gets created upon the first HTTP request involving this bean in the application (or when the web application starts up and the eager=true attribute is set in @ManagedBean) and gets destroyed when the web application shuts down.
@CustomScoped	Bean lives as long as the bean's entry in the custom Map, which is created for this scope lives.

@ManagedProperty Annotation

JSF is a simple static Dependency Injection (DI) framework. Using **@ManagedProperty** annotation, a managed bean's property can be injected in another managed bean.

Example Application

Let us create a test JSF application to test the above annotations for managed beans.

Step	Description
1	Create a project with a name *helloworld* under a package *com.tutorialspoint.test* as explained in the *JSF - Create Application* chapter.
2	Modify *HelloWorld.java* as explained below. Keep the rest of the files unchanged.
3	Create *Message.java* under a package *com.tutorialspoint.test* as explained below.
4	Compile and run the application to make sure 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.

HelloWorld.java

```
package com.tutorialspoint.test;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.RequestScoped;

@ManagedBean(name = "helloWorld", eager = true)
@RequestScoped
public class HelloWorld {

   @ManagedProperty(value="#{message}")
   private Message messageBean;
   private String message;
   public HelloWorld() {
      System.out.println("HelloWorld started!");
   }
   public String getMessage() {
      if(messageBean != null){
         message = messageBean.getMessage();
      }
      return message;
   }
   public void setMessageBean(Message message) {
      this.messageBean = message;
   }
}
```

Message.java

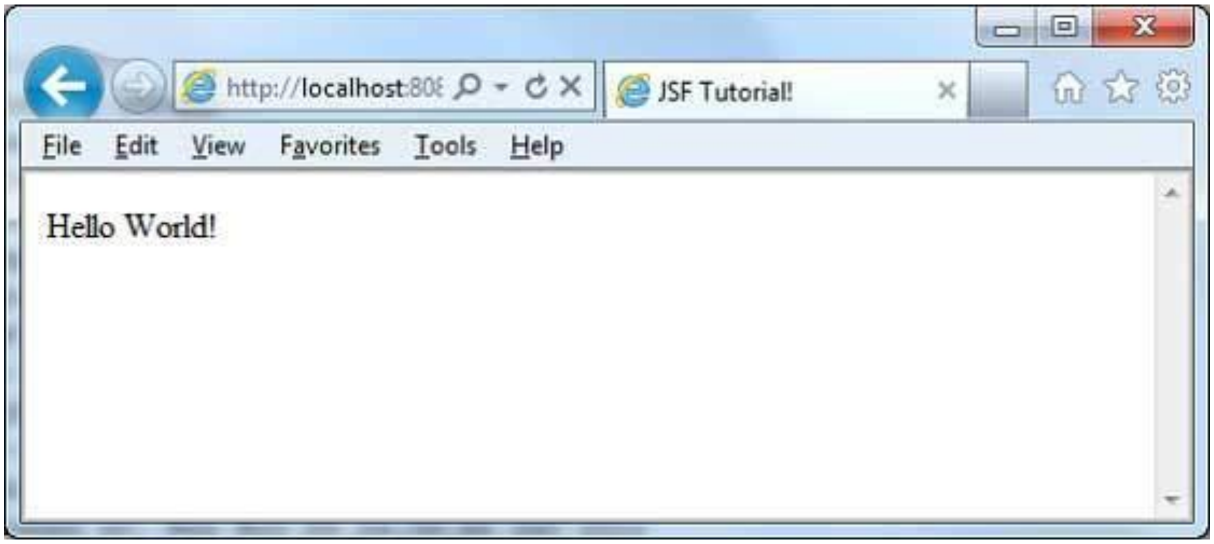
```
package com.tutorialspoint.test;

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

@ManagedBean(name = "message", eager = true)
@RequestScoped
public class Message {

     private String message = "Hello World!";
     public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

home.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">
<head>
   <title>JSF Tutorial!</title>
</head>
<body>
   #{helloWorld.message}
</body>
</html>
```

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



Hello World!

7. JSF – PAGE NAVIGATION

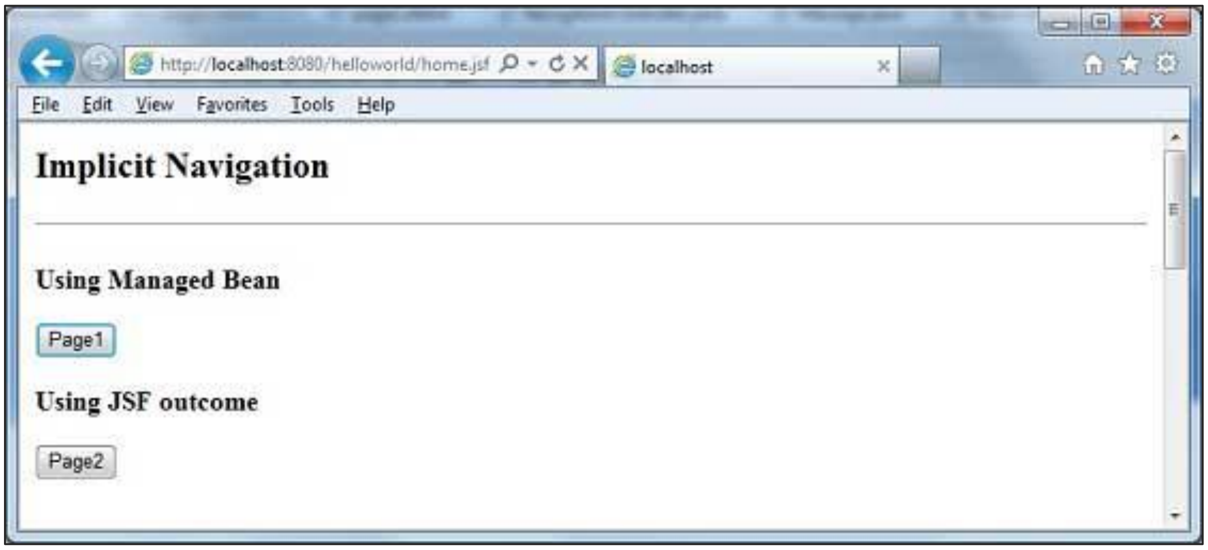
Navigation rules are those rules provided by JSF Framework that describes which view is to be shown when a button or a link is clicked.

Navigation rules can be defined in JSF configuration file named faces-config.xml. They can be defined in managed beans.

Navigation rules can contain conditions based on which the resulted view can be shown. JSF 2.0 provides implicit navigation as well in which there is no need to define navigation rules as such.

Implicit Navigation

JSF 2.0 provides **auto view page resolver** mechanism named **implicit navigation**. In this case, you only need to put view name in **action** attribute and JSF will search the correct **view** page automatically in the deployed application.

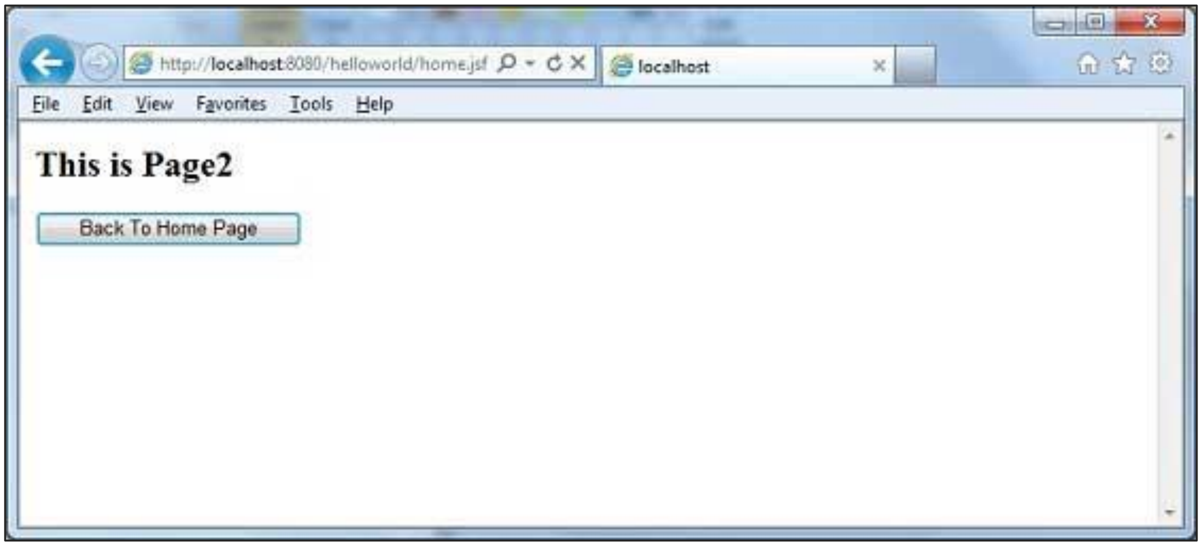

http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
Implicit Navigation
Using Managed Bean
Page1
Using JSF outcome
Page2

Auto Navigation in JSF Page

Set view name in action attribute of any JSF UI Component.

```
<h:form>
   <h3>Using JSF outcome</h3>
   <h:commandButton action="page2" value="Page2" />
</h:form>
```

Here, when **Page2** button is clicked, JSF will resolve the view name, **page2** as page2.xhtml extension, and find the corresponding view file **page2.xhtml** in the current directory.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
This is Page2
Back To Home Page

Auto Navigation in Managed Bean

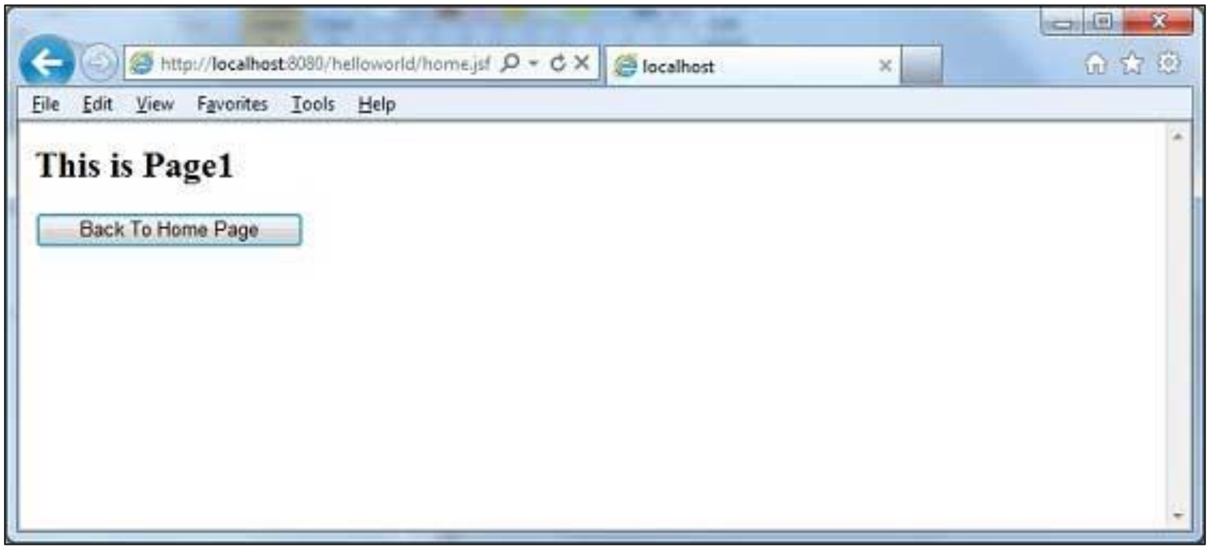
Define a method in managed bean to return a view name.

```
@ManagedBean(name = "navigationController", eager = true)
@RequestScoped
public class NavigationController implements Serializable {
   public String moveToPage1(){
      return "page1";
   }
}
```

Get view name in action attribute of any JSF UI Component using managed bean.

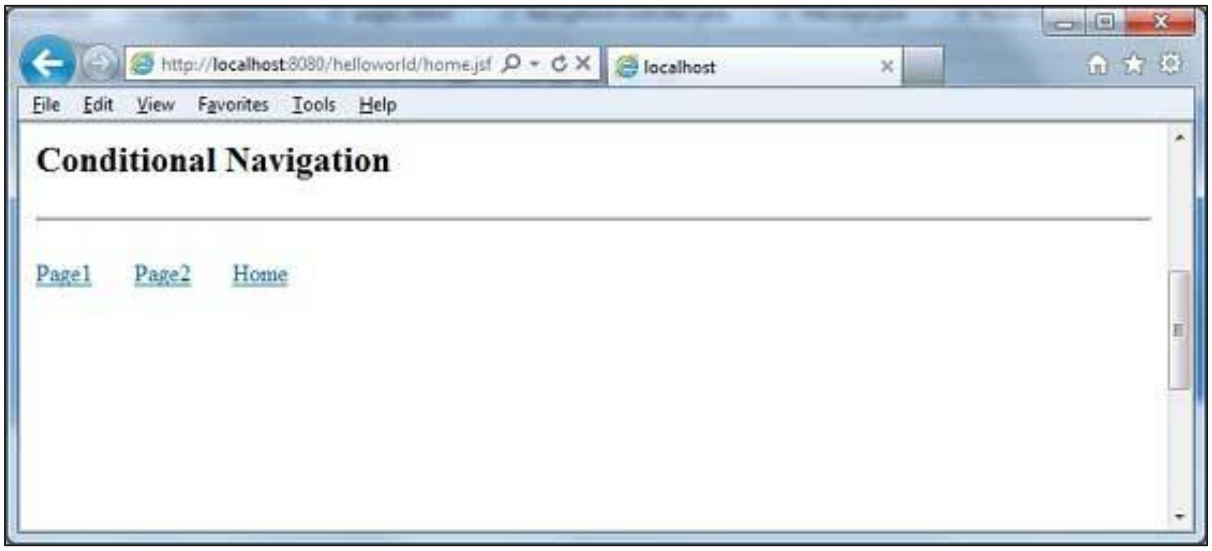
```
<h:form>
   <h3>Using Managed Bean</h3>
   <h:commandButton action="#{navigationController.moveToPage1}"
   value="Page1" />
</h:form>
```

Here, when **Page1** button is clicked, JSF will resolve the view name, **page1** as page1.xhtml extension, and find the corresponding view file **page1.xhtml** in the current directory.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
This is Page1
Back To Home Page

Conditional Navigation

Using managed bean, we can very easily control the navigation. Look at the following code in a managed bean.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
Conditional Navigation
Page1 Page2 Home

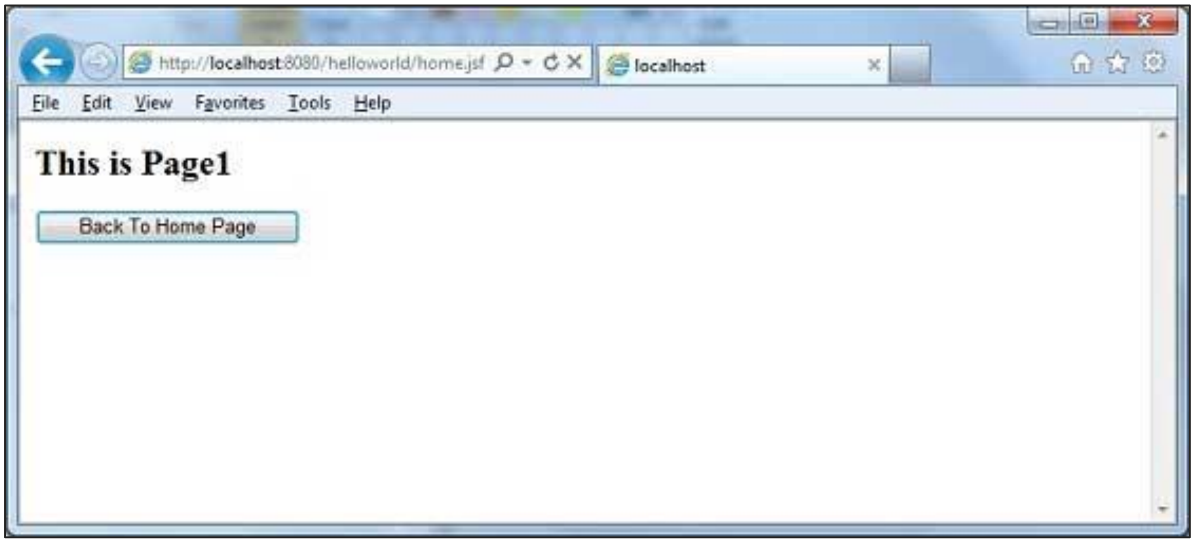
```
@ManagedBean(name = "navigationController", eager = true)
@RequestScoped
public class NavigationController implements Serializable {


   //this managed property will read value from request parameter pageId
   @ManagedProperty(value="#{param.pageId}")
   private String pageId;

   //condional navigation based on pageId
   //if pageId is 1 show page1.xhtml,
   //if pageId is 2 show page2.xhtml
   //else show home.xhtml
   public String showPage(){
      if(pageId == null){
         return "home";
      }
      if(pageId.equals("1")){
         return "page1";
      }else if(pageId.equals("2")){
         return "page2";
      }else{
         return "home";
      }
   }
}
```

Pass pageId as a request parameter in JSF UI Component.

```
<h:form>
   <h:commandLink action="#{navigationController.showPage}" value="Page1">
      <f:param name="pageId" value="1" />
   </h:commandLink>
   <h:commandLink action="#{navigationController.showPage}" value="Page2">
      <f:param name="pageId" value="2" />
   </h:commandLink>
   <h:commandLink action="#{navigationController.showPage}" value="Home">
      <f:param name="pageId" value="3" />
   </h:commandLink>
</h:form>
```

Here, when "Page1" button is clicked.

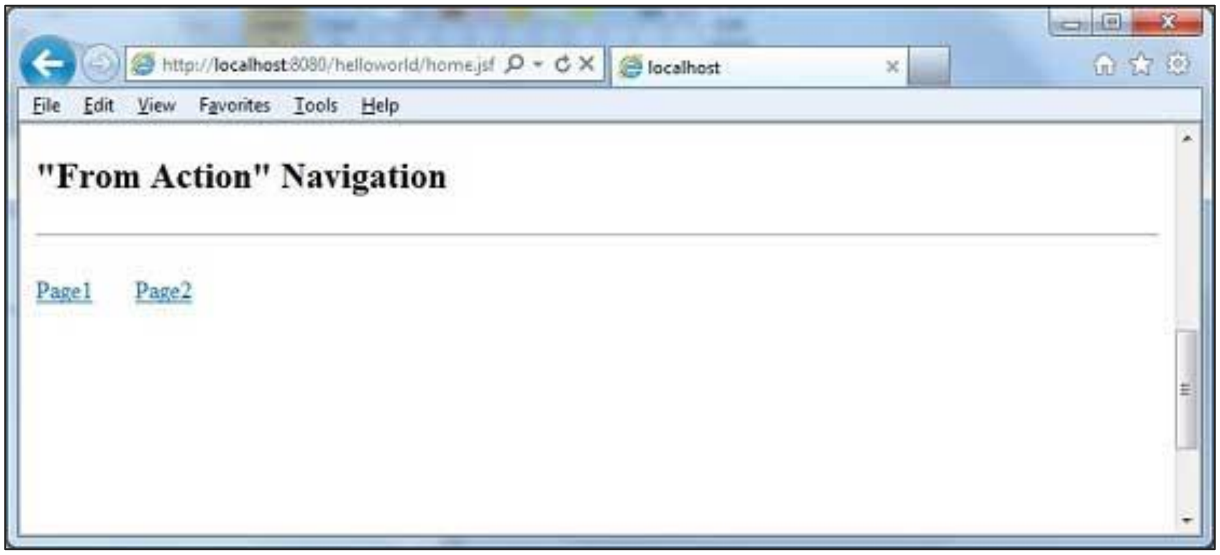
- JSF will create a request with parameter pageId=1
- Then JSF will pass this parameter to managed property pageId of navigationController
- Now navigationController.showPage() is called which will return view as page1 after checking the pageId
- JSF will resolve the view name, page1 as page1.xhtml extension
- Find the corresponding view file page1.xhtml in the current directory



This is Page1

Resolving Navigation Based on from-action

JSF provides navigation resolution option even if managed bean different methods returns the same view name.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help

"From Action" Navigation

Page1 Page2

Look at the following code in a managed bean.

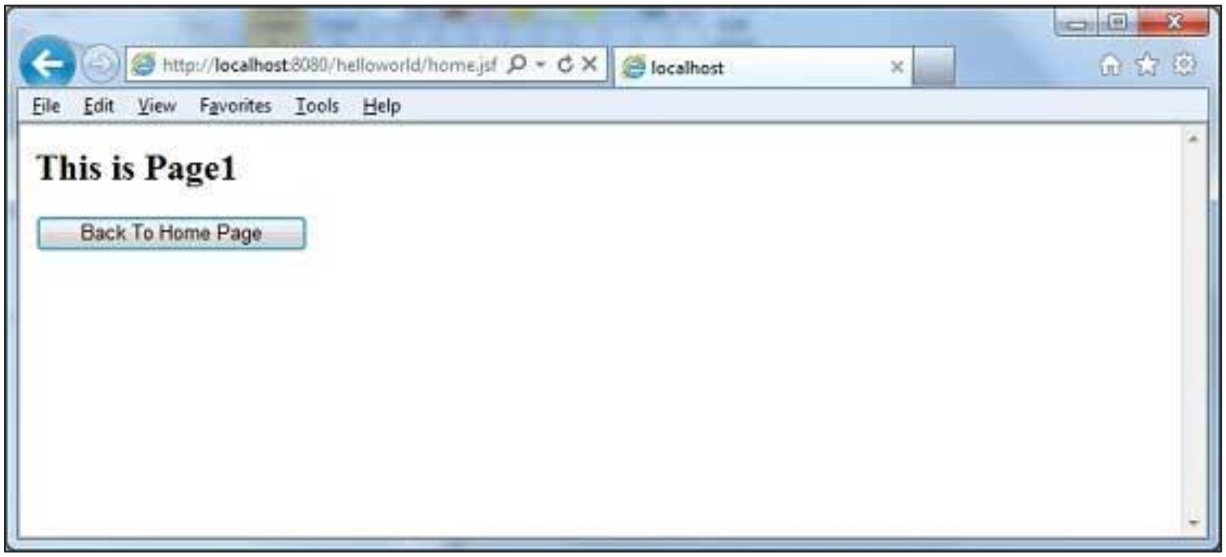
```
public String processPage1(){
   return "page";
}
public String processPage2(){
   return "page";
}
```

To resolve views, define the following navigation rules in **faces-config.xml**

```
<navigation-rule>
   <from-view-id>home.xhtml</from-view-id>
   <navigation-case>
      <from-action>#{navigationController.processPage1}</from-action>
      <from-outcome>page</from-outcome>
      <to-view-id>page1.jsf</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-action>#{navigationController.processPage2}</from-action>
      <from-outcome>page</from-outcome>
      <to-view-id>page2.jsf</to-view-id>
   </navigation-case>
</navigation-rule>
```

Here, when **Page1** button is clicked -

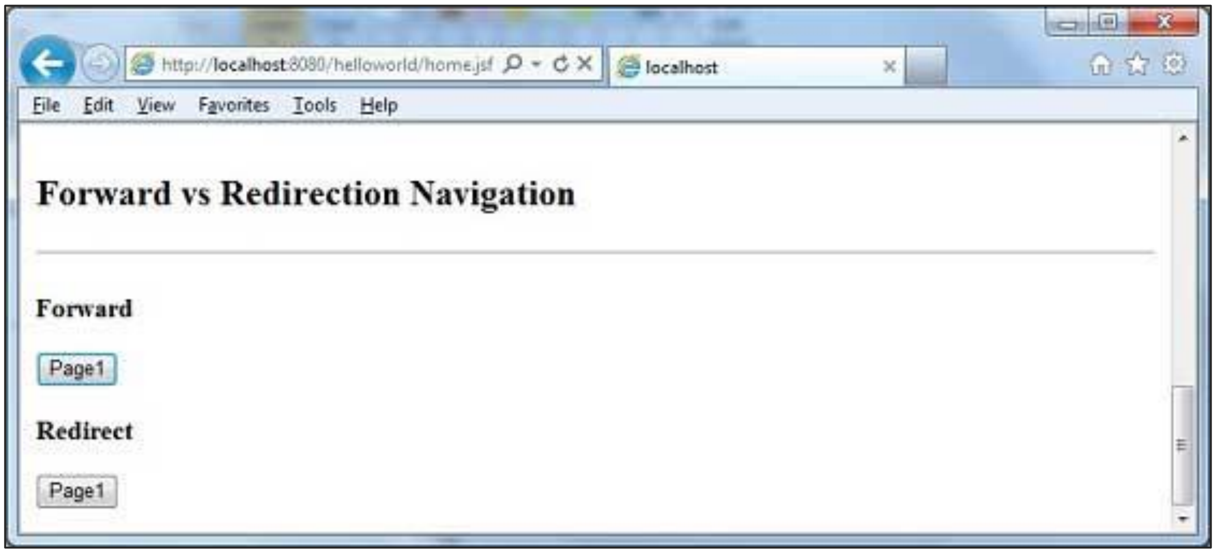
- **navigationController.processPage1()** is called which will return view as page
- JSF will resolve the view name, **page1** as view name is **page and from-action** in **faces-config** is **navigationController.processPage1**
- Find the corresponding view file **page1.xhtml** in the current directory


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
This is Page1
Back To Home Page

Forward vs Redirect

JSF by default performs a server page forward while navigating to another page and the URL of the application does not change.

To enable the page redirection, append **faces-redirect=true** at the end of the view name.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help

Forward vs Redirection Navigation

Forward

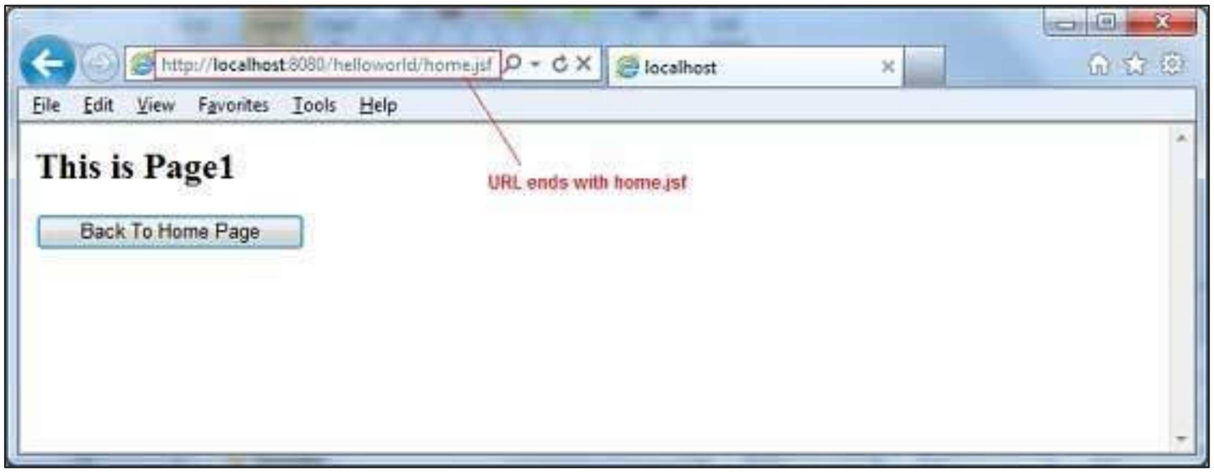
Page1

Redirect

Page1

```
<h:form>
   <h3>Forward</h3>
   <h:commandButton action="page1" value="Page1" />
   <h3>Redirect</h3>
   <h:commandButton action="page1?faces-redirect=true" value="Page1" />
</h:form>
```

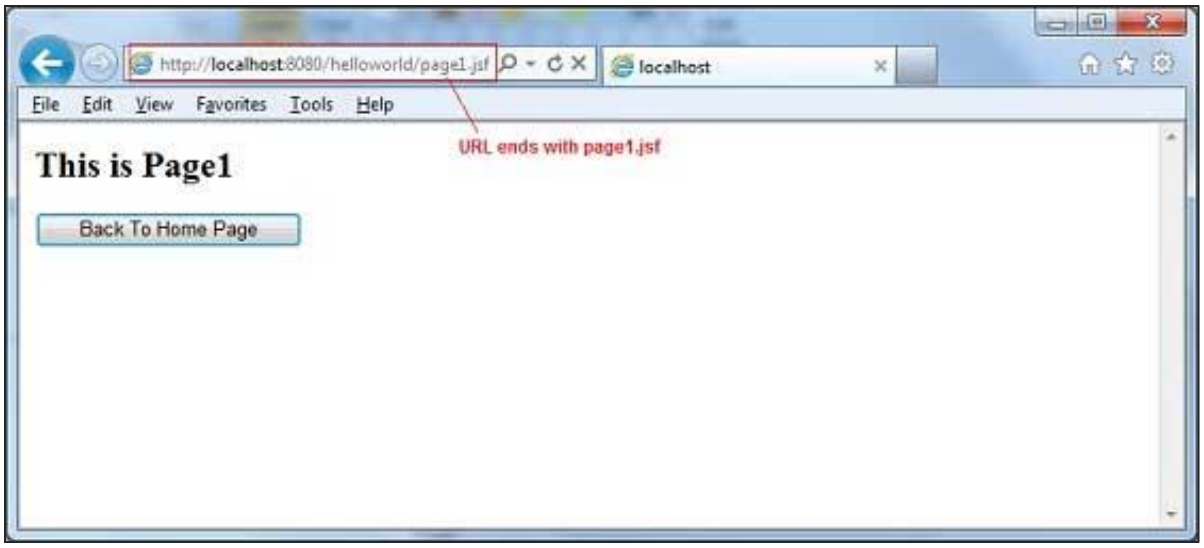
Here, when **Page1** button under **Forward** is clicked, you will get the following result.


http://localhost:8080/helloworld/home.jsf
localhost
File Edit View Favorites Tools Help
URL ends with home.jsf

This is Page1

Back To Home Page

Here when **Page1** button under **Redirect** is clicked, you will get the following result.


http://localhost:8080/helloworld/page1.jsf
localhost
File Edit View Favorites Tools Help
URL ends with page1.jsf
This is Page1
Back To Home Page

Example Application

Let us create a test JSF application to test all of the above navigation examples.

Step	Description
1	Create a project with a name *helloworld* under a package *com.tutorialspoint.test* as explained in the *JSF - Create Application* chapter.
2	Create *NavigationController.java* under a package *com.tutorialspoint.test* as explained below.
3	Create *faces-config.xml* under a *WEB-INF* folder and updated its contents as explained below.
4	Update *web.xml* under a *WEB-INF* folder as explained below.
5	Create *page1.xhtml* and *page2.xhtml* and modify *home.xhtml* under a *webapp* folder as explained below.
6	Compile and run the application to make sure 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.

NavigationController.java

```
package com.tutorialspoint.test;

import java.io.Serializable;

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

@ManagedBean(name = "navigationController", eager = true)
@RequestScoped
public class NavigationController implements Serializable {

   private static final long serialVersionUID = 1L;

   @ManagedProperty(value="#{param.pageId}")
   private String pageId;

   public String moveToPage1(){
      return "page1";
   }

   public String moveToPage2(){
      return "page2";
   }

   public String moveToHomePage(){
      return "home";
   }

   public String processPage1(){
      return "page";
   }

   public String processPage2(){
      return "page";
   }

   public String showPage(){
      if(pageId == null){
         return "home";
      }
      if(pageId.equals("1")){
         return "page1";
      }else if(pageId.equals("2")){
         return "page2";
      }else{
         return "home";
      }
   }

   public String getPageId() {
      return pageId;
   }

   public void setPageId(String pageId) {
      this.pageId = pageId;
   }
}
```

faces-config.xml

```
<?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">
   <navigation-rule>
      <from-view-id>home.xhtml</from-view-id>
      <navigation-case>
         <from-action>#{navigationController.processPage1}</from-action>
         <from-outcome>page</from-outcome>
         <to-view-id>page1.jsf</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-action>#{navigationController.processPage2}</from-action>
         <from-outcome>page</from-outcome>
         <to-view-id>page2.jsf</to-view-id>
      </navigation-case>
   </navigation-rule>
</faces-config>
```

web.xml

```
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd" >

   <web-app>
   <display-name>Archetype Created Web Application</display-name>

   <context-param>
      <param-name>javax.faces.PROJECT_STAGE</param-name>
      <param-value>Development</param-value>
   </context-param>
   <context-param>
      <param-name>javax.faces.CONFIG_FILES</param-name>
      <param-value>/WEB-INF/faces-config.xml</param-value>
   </context-param>
   <servlet>
```

```
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.jsf</url-pattern>
   </servlet-mapping>
</web-app>
```

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</h2>
      <h:form>
         <h:commandButton action="home?faces-redirect=true"
            value="Back To Home Page" />
      </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</h2>
      <h:form>
         <h:commandButton action="home?faces-redirect=true"
            value="Back To Home Page" />
      </h:form>
   </h:body>
</html>
```

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:body>
      <h2>Implicit Navigation</h2>
      <hr />
      <h:form>
         <h3>Using Managed Bean</h3>
         <h:commandButton action="#{navigationController.moveToPage1}"
            value="Page1" />
         <h3>Using JSF outcome</h3>
```

```
          <h:commandButton action="page2" value="Page2" />
       </h:form>
       <br/>
       <h2>Conditional Navigation</h2>
       <hr />
       <h:form>
          <h:commandLink action="#{navigationController.showPage}"
             value="Page1">
             <f:param name="pageId" value="1" />
          </h:commandLink>

          <h:commandLink action="#{navigationController.showPage}"
             value="Page2">
             <f:param name="pageId" value="2" />
          </h:commandLink>

          <h:commandLink action="#{navigationController.showPage}"
             value="Home">
             <f:param name="pageId" value="3" />
          </h:commandLink>
       </h:form>
       <br/>
       <h2>"From Action" Navigation</h2>
       <hr />
       <h:form>
          <h:commandLink action="#{navigationController.processPage1}"
          value="Page1" />

          <h:commandLink action="#{navigationController.processPage2}"
          value="Page2" />

       </h:form>
       <br/>
       <h2>Forward vs Redirection Navigation</h2>
       <hr />
```

```
      <h:form>
         <h3>Forward</h3>
         <h:commandButton action="page1" value="Page1" />
         <h3>Redirect</h3>
         <h:commandButton action="page1?faces-redirect=true"
         value="Page1" />
      </h:form>
   </h:body>
</html>
```