Java Server Faces

JSF

- We have discussed about tiers and containers in JavaEE application and what these tiers and containers consist of.
- JavaServer Faces (JSF) is a Java-based web application framework intended to simplify *development integration of web-based user interfaces*.
- This part of the application belongs to the web tier and is assembled to web container module.

JSF

- JavaServer Faces (JSF) is a MVC web framework that simplifies the construction of user interfaces (UI) for server-based applications by using reusable UI components in a page.
  - Connect UI widgets to data sources
  - Connect widgets to *server-side event handlers*

JSF Benefits

- Provides reusable UI components
- Easy data transfer between UI components
- Manages UI state across multiple server requests.
- Enables making custom UI components
- Wiring client side event to server side application code

JSF UI component model

- JSF provides developers capability to create Web application from collections of UI components that can render themselves in different ways in different clients.
- Client can be HTML browser in desktop, browser in mobile phone, or any other device capable of rendering HTML.

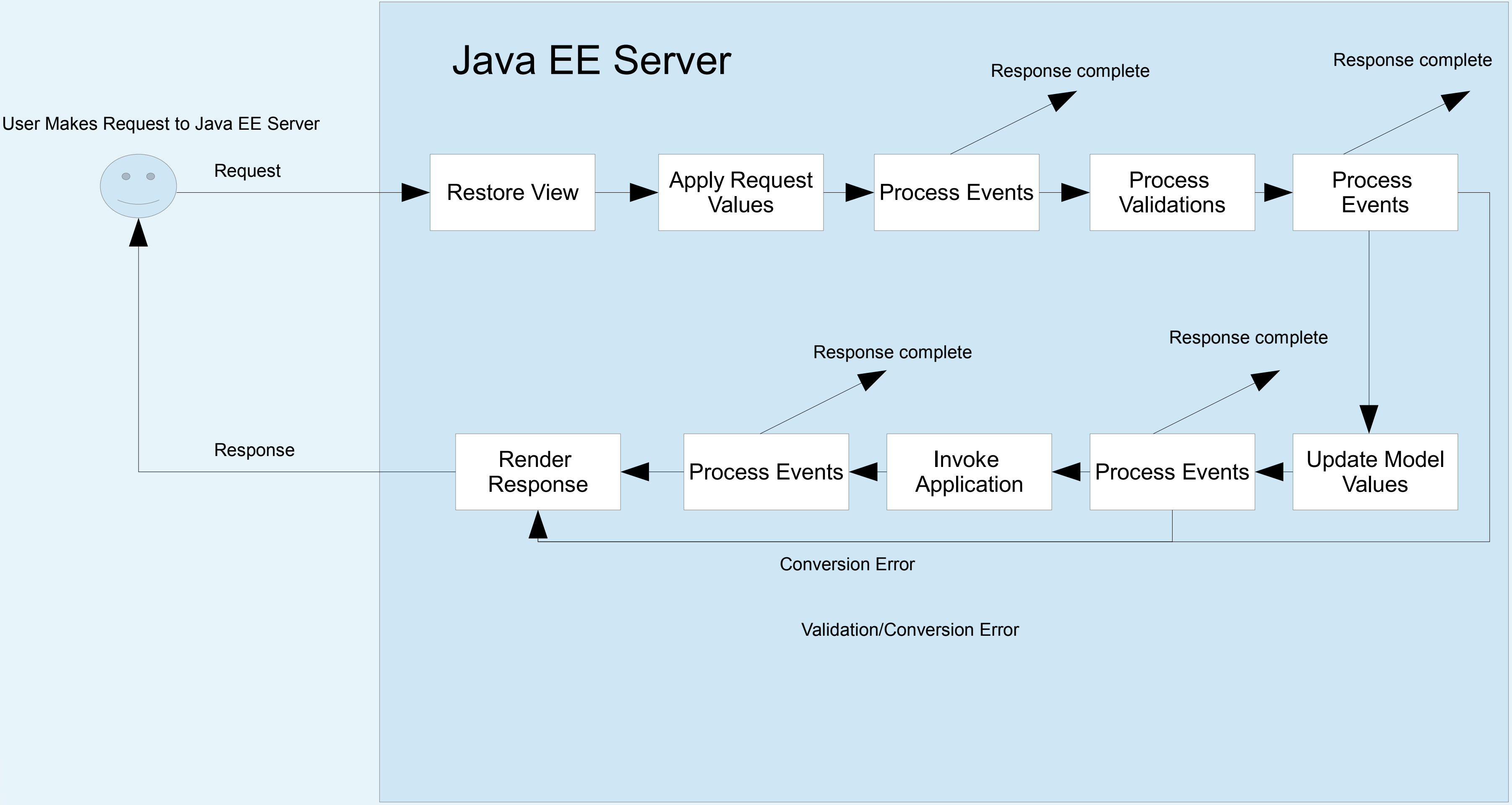
JSF

- JSF provides:
  - Core library
  - A set of base UI components - standard HTML input elements
  - Possibility to create custom components.
  - Multiple rendering capabilities that enable JSF UI components to render themselves differently depending on the client types.

JSF Application Lifecycle

- JSF application lifecycle consist of six phases:
  - Restore view phase
  - Apply request values phase
  - Process validations phase
  - Update model values phase
  - Invoke application phase
  - Render response phase

Basic Application Flow



Java EE Server

User Makes Request to Java EE Server

Request

Restore View

Apply Request Values

Process Events

Process Validations

Process Events

Response complete

Response complete

Update Model Values

Process Events

Invoke Application

Process Events

Render Response

Response complete

Response complete

Response

Conversion Error

Validation/Conversion Error

Restore View Phase

- JSF begins the restore view phase when there is an incoming request from client.
- In this phase JSF builds the view and wires event handlers and validator to UI components.
- Then it saves the instance of view in FaceContext instance.
- The FacesContext instance will now contain all the needed information required to process a request.

Apply Request Values Phase

- After the component tree is created/restored, each component in component tree uses 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.*

Process Validation Phase

- During this phase, the JSF processes all validators registered on 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, the JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and display the same page again with the error message.

Update Model Values Phase

- After the JSF checks that the data is valid, it walks over the component tree and set the corresponding server-side object properties to the components' local values. The JSF will update the bean properties corresponding to input component's value attribute.
- If any updateModels methods called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Invoke Application Phase

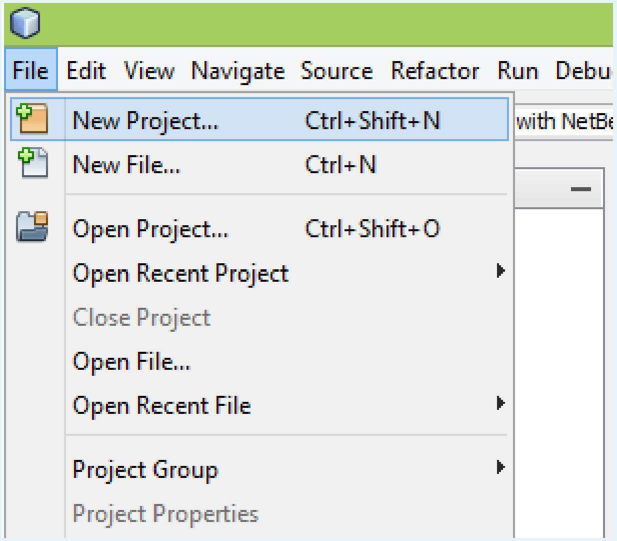
- During this phase, the JSF handles *any application-level events, such as submitting a form / linking to another page*.

Render Response Phase

- During this phase, the 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 the JSP container executes 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.

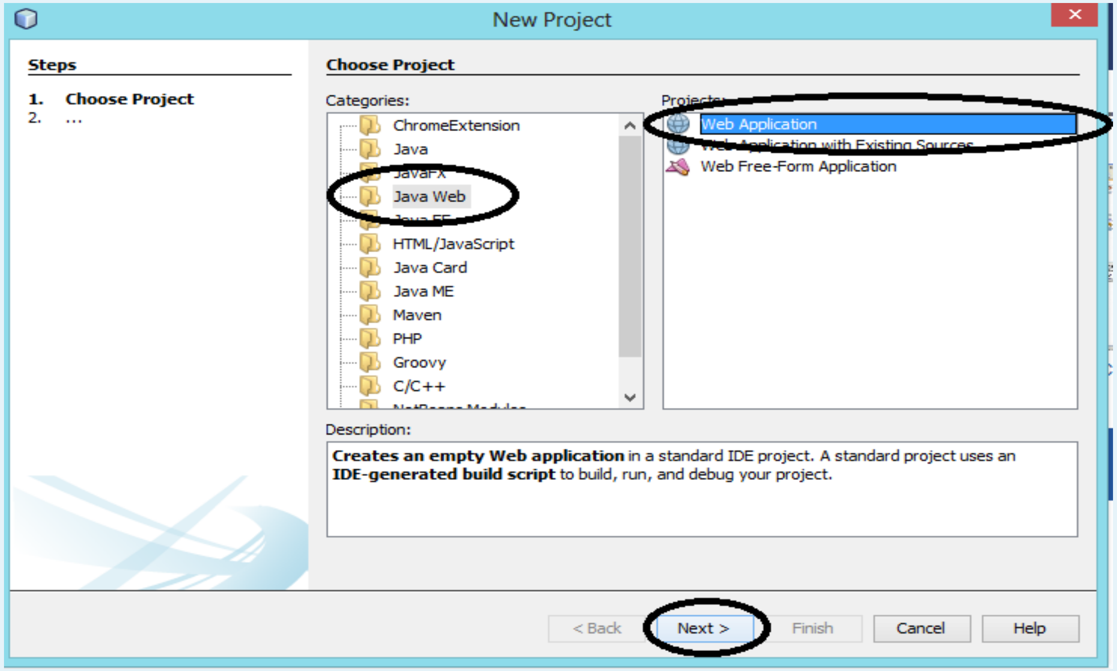
Creating Web Application Project With Java Server Faces Framework

- Launch NetBeans IDE.
- Select File-> New Project



First Project

- From *Categories* select Java Web
- From *Projects* select Web application
- Press Next



New Project

Steps

1. **Choose Project**
2. ...

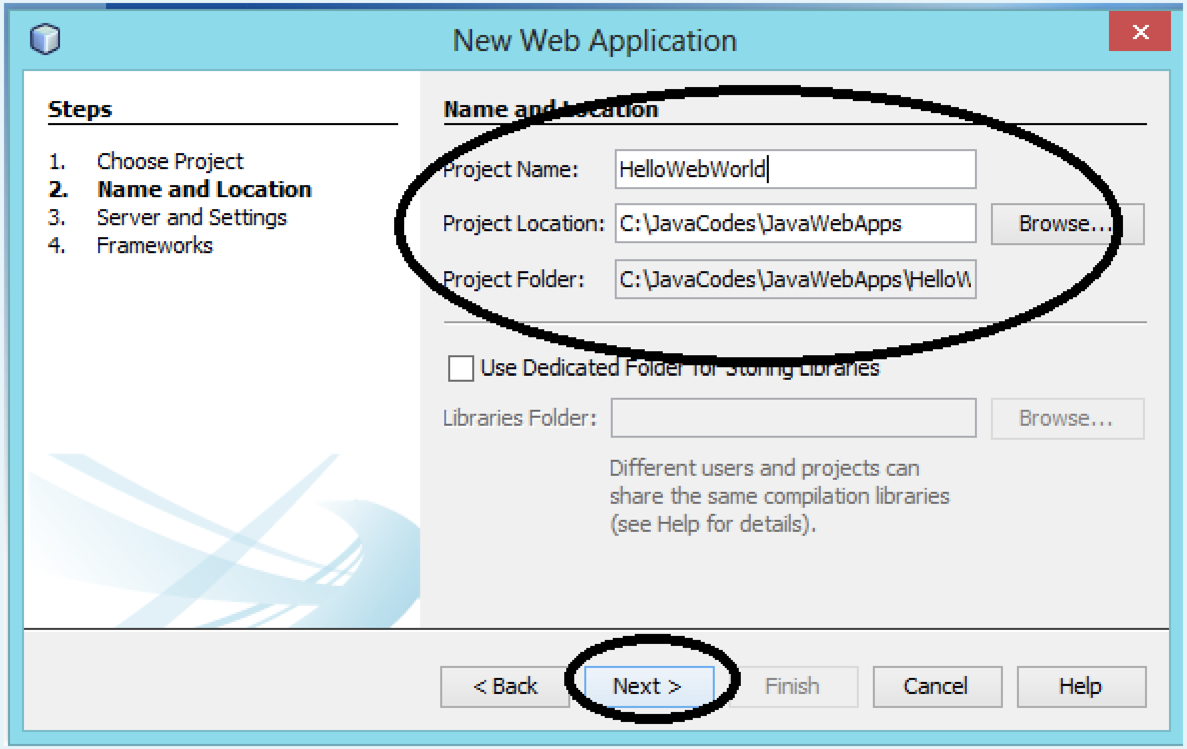
Choose Project

Description:

Creates an empty Web application in a standard IDE project. A standard project uses an **IDE-generated build script** to build, run, and debug your project.

First Project

- Give name and location for your project.
- Press Next



New Web Application

Steps

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

Name and Location

Project Name: HelloWebWorld

Project Location: C:\JavaCodes\JavaWebApps

Project Folder: C:\JavaCodes\JavaWebApps\HelloW

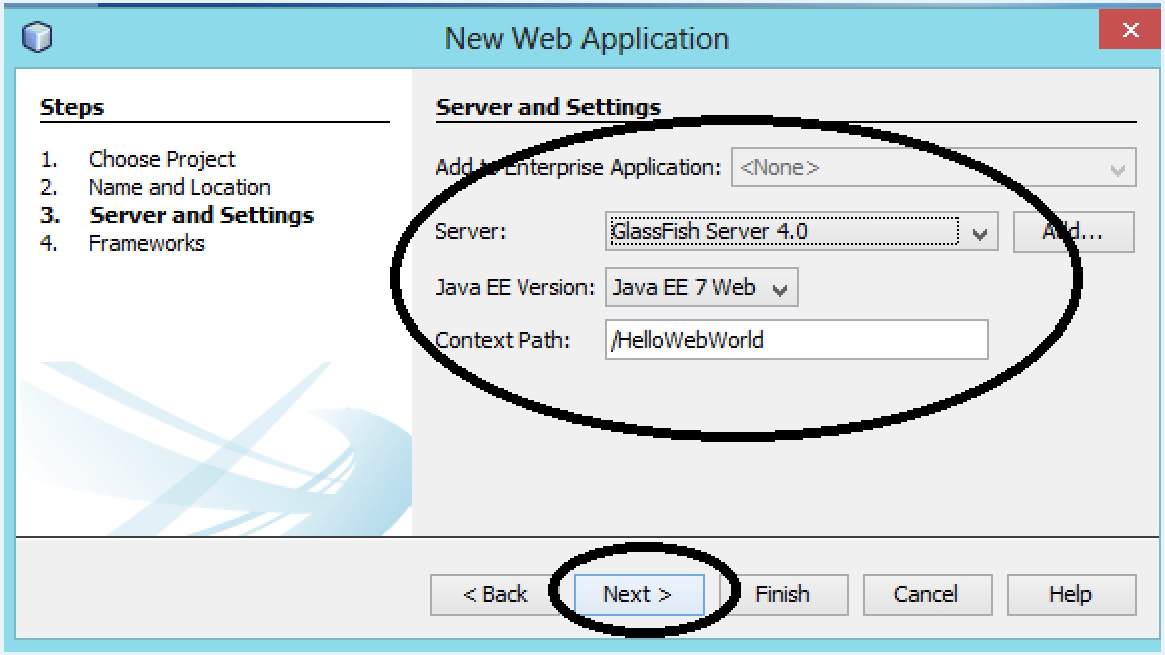
☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

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

First Project

- Define the project settings. Makes sure that GlassFish server 4.0 is selected and JavaEE version is 7.0
- Press Next



New Web Application

Steps

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

Server and Settings

Add to Enterprise Application: <None>

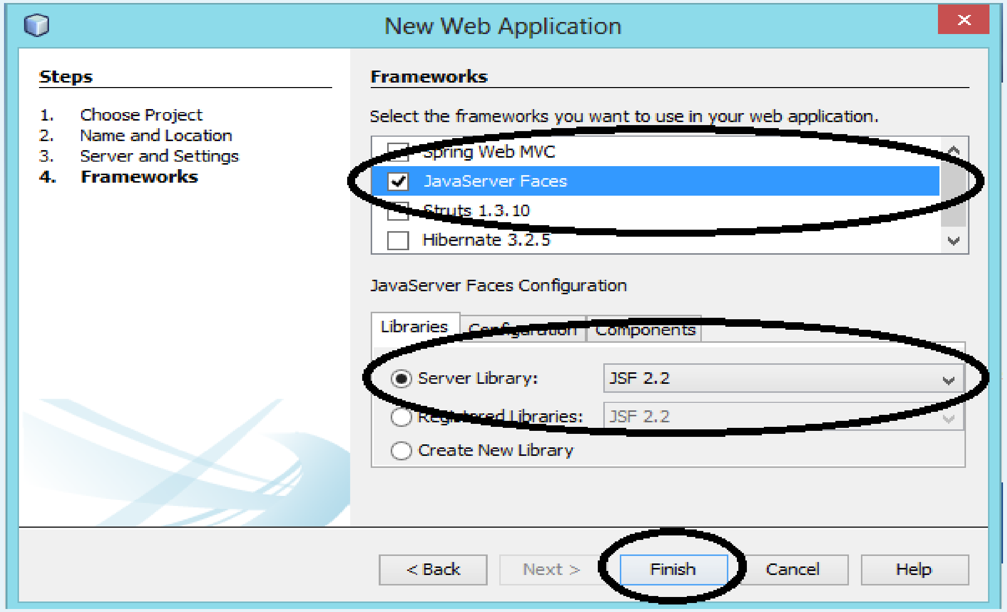
Server: GlassFish Server 4.0

Java EE Version: Java EE 7 Web

Context Path: /HelloWebWorld

First Project

- Select Java Server Faces from frameworks list.
- Makes sure that JSF version 2.2 is selected.
- Press Finish



New Web Application

Steps

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

Frameworks

Select the frameworks you want to use in your web application.

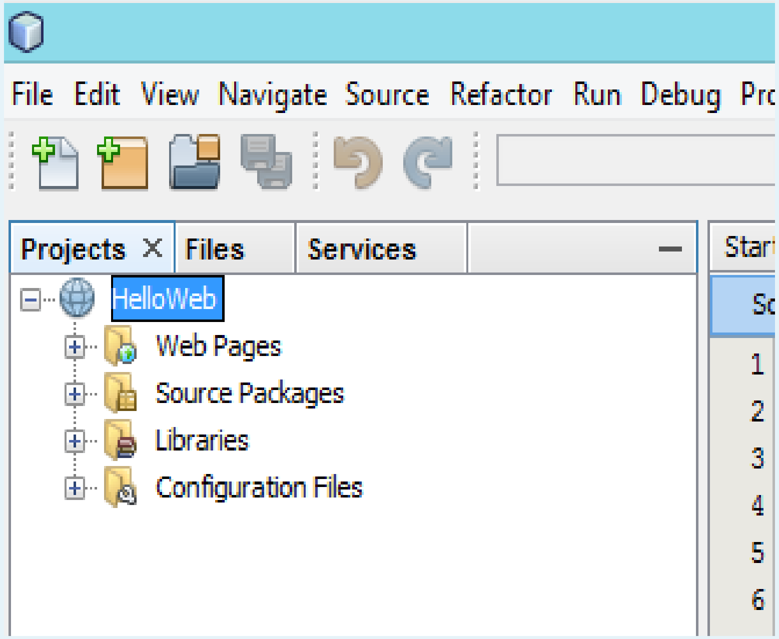
□ Spring Web MVC
☑ JavaServer Faces
□ Struts 1.3.10
□ Hibernate 3.2.5

JavaServer Faces Configuration

● Server Library: JSF 2.2
○ Registered Libraries: JSF 2.2
○ Create New Library

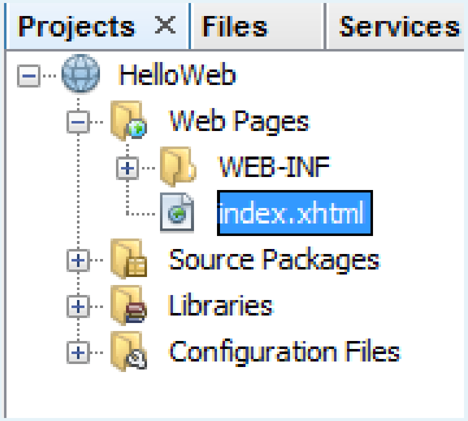
Project Content

- The project tree in this case is as follow:



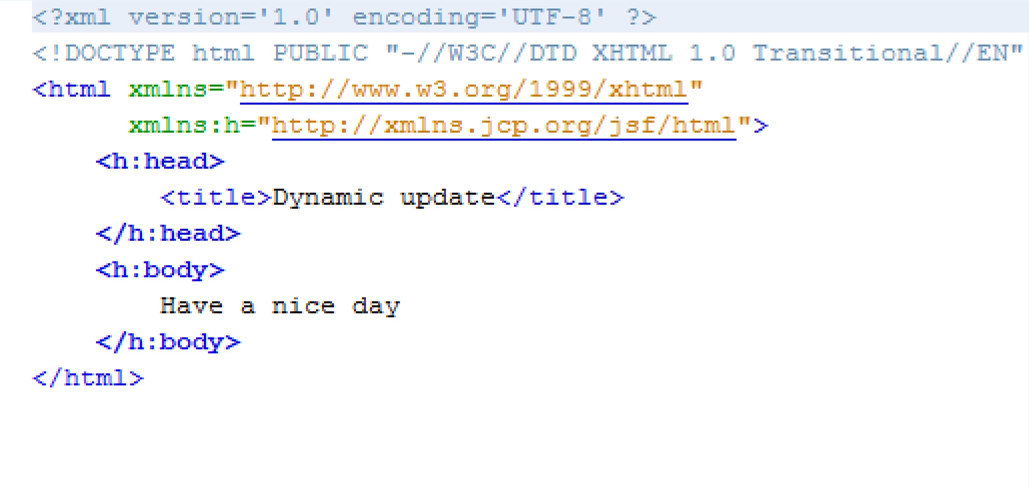
Project Content

- The Web Pages folder: Contains all web resources like JSF pages, HTML, CSS, Images and other resources
- If you open it you see WEB-INF folder and index.xhtml file



Project Content

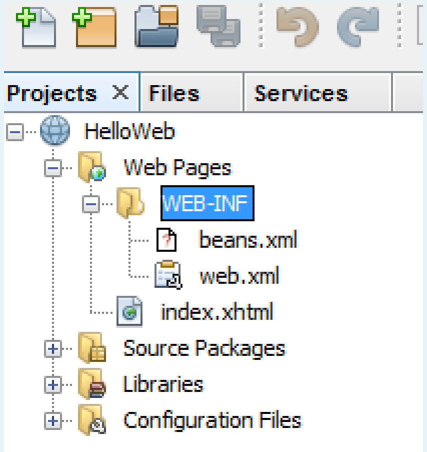
- Index.xhtml: This is your first JSF file in project. The content of file is as follow:



```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Dynamic update</title>
    </h:head>
    <h:body>
        Have a nice day
    </h:body>
</html>
```

Project Content

- WE-INF folder: Contains the project configuration files.
- If you open it you see two files web.xml and beans.xml

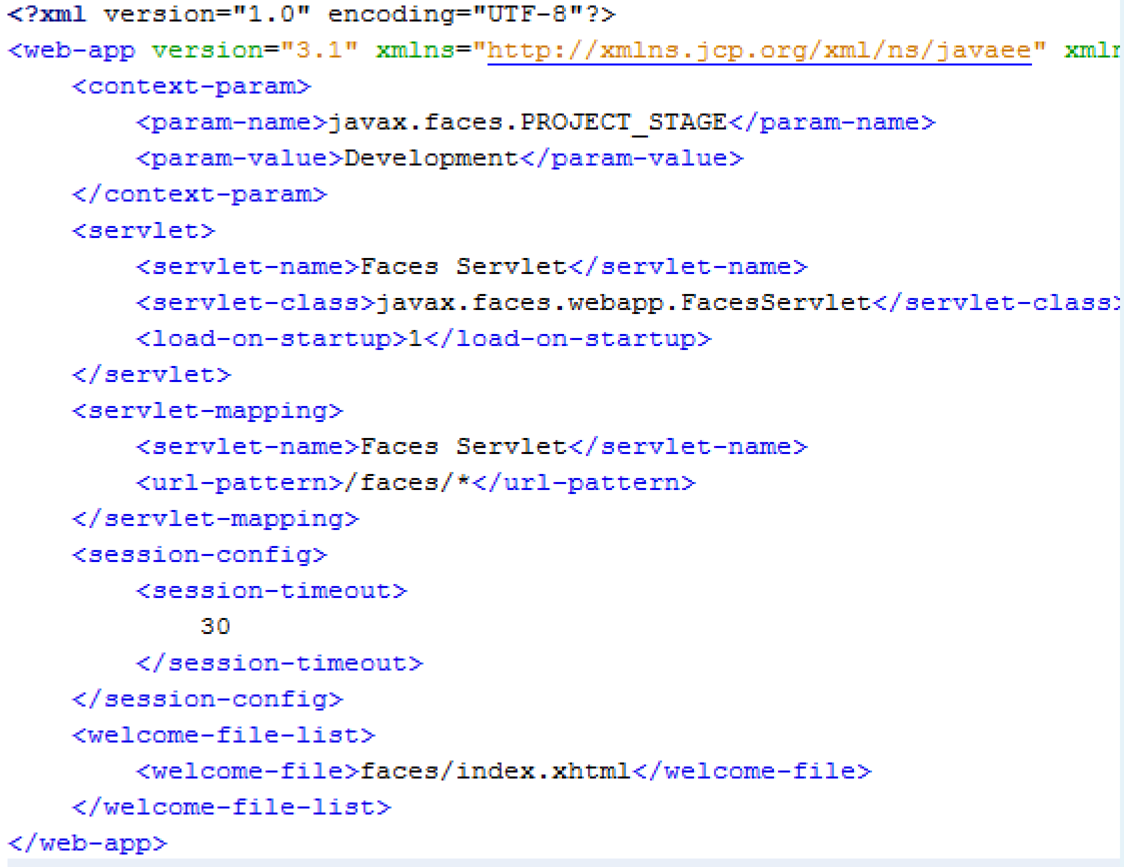


Project Content

- web.xml: Project main configuration file. Contains the needed Servlet mappings.
- In basic project you don't usually need to modify this file.
- beans.xml: Contains metadata about your JavaBeans defined in application. Because the basic project template don't create any beans for us, this file contains just basic XML data needed.

Project Content

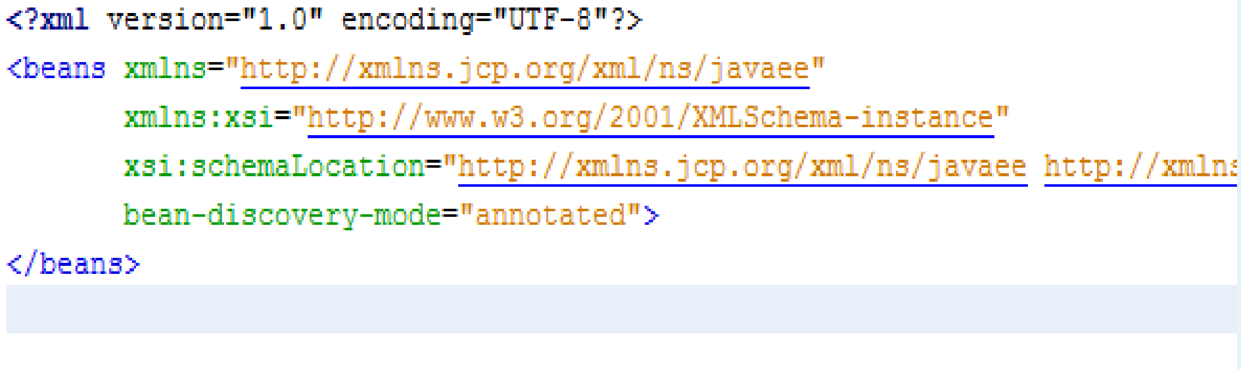
- web.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmln
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Project Content

- beans.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns
       bean-discovery-mode="annotated">
</beans>
```

Project Content

- Source Packages folder: Contains all the .java files of your project. In freshly created project this is empty.
- Configuration Files: Contains all the configuration files. By default contains the same files as WEB-INF + MANIFEST.MF file. This folder can contain also navigation rules file, localization files etc.

Web.xml

- If you open web.xml file you can see that <context-param> is set to *javax.faces.PROJECT_STAGE* and *Development*.
- When context is set to *development* it will provide many useful debugging information to let you track the bugs easily.
- For deployment, just change it to “Production“.

Web.xml

- The <servlet> element contains the servlet class which is set to *javax.faces.webapp.FacesServlet.*
- This just points to class that process the JavaServer Faces requests.
- The load-on-startup element indicates that this servlet should be loaded (instantiated and have its init() called) on the startup of the web application.
- The optional contents of these element must be an integer indicating the order in which the servlet should be loaded. If the value is a negative integer, or the element is not present, the container is free to load the servlet whenever it chooses.

Web.xml

- The servlet-mapping element defines a mapping between a servlet and a URL pattern.
- <servlet-name>:The name of the servlet to which you are mapping a URL pattern. This name corresponds to *the name you assigned a servlet in a <servlet> declaration tag.*

Web.xml

- <url-pattern>:Describes a pattern used to resolve URLs. The portion of the URL after the http://host:port + WebAppName is compared to the <url-pattern> by WebLogic Server. If the patterns match, the servlet mapped in this element will be called.
- Example:/foo/*

Web.xml

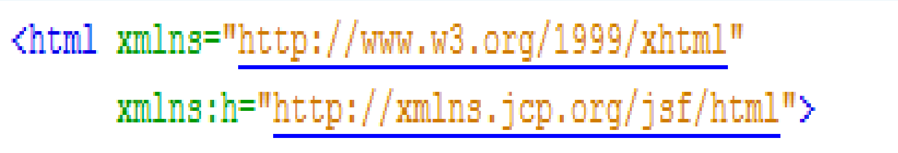
- The session-config element defines the session attributes for this Web application.
- <session-timeout>: The number of minutes after which sessions in this Web application expire.

Web.xml

- The optional welcome-file-list element contains an ordered list of welcome-file elements.
- When the URL request is a directory name, server serves the first file specified in this element. If that file is not found, the server then tries the next file in the list.

Index.html

- index.xhtml file is the default JSF page generated for us upon project creation.
- The first two lines are as follow:



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

- These lines adds the needed JSF tag library in html with prefix 'h' by default.
- This means if you want to use some element from JSF HTML tag library you need to prefix that element with letter *h* i.e.

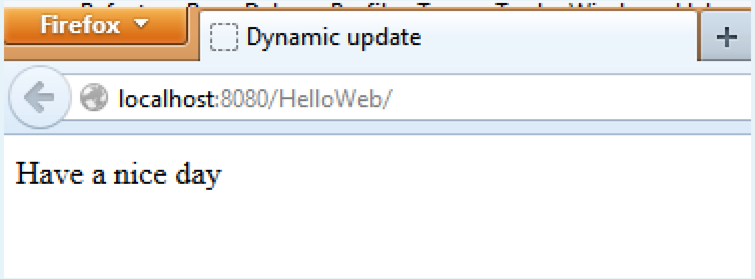
  <h:head>

Index.html

- The default template code contains the needed elements to display title and greeting message in browser.

Run the Application

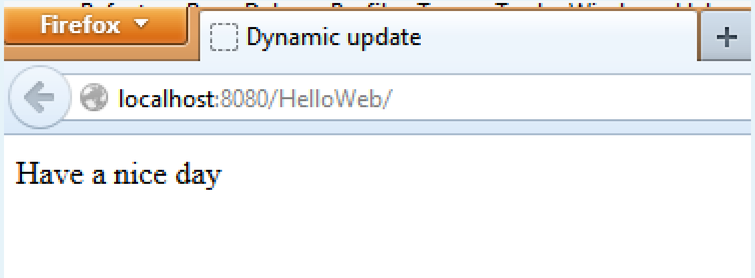
- To verify that our environment is working, we need to deploy the application to GlassFish server and check that the greeting message is rendered in our browser window.
- This is actually a very complex process, but thanks to framework YOU only need to press the green ”play” button in NetBeans IDE.
- What are you waiting? Press the button...
- If everything works fine the process kicks your default browser up and you should see the greeting text in browser window



Have a nice day

Run the Application

- To verify that our environment is working, we need to deploy the application to GlassFish server and check that the greeting message is rendered in our browser window.
- This is actually a very complex process, but thanks to framework YOU only need to press the green ”play” button in NetBeans IDE.
- What are you waiting? Press the button...
- If everything works fine the process kicks your default browser up and you should see the greeting text in browser window



Have a nice day

JSF

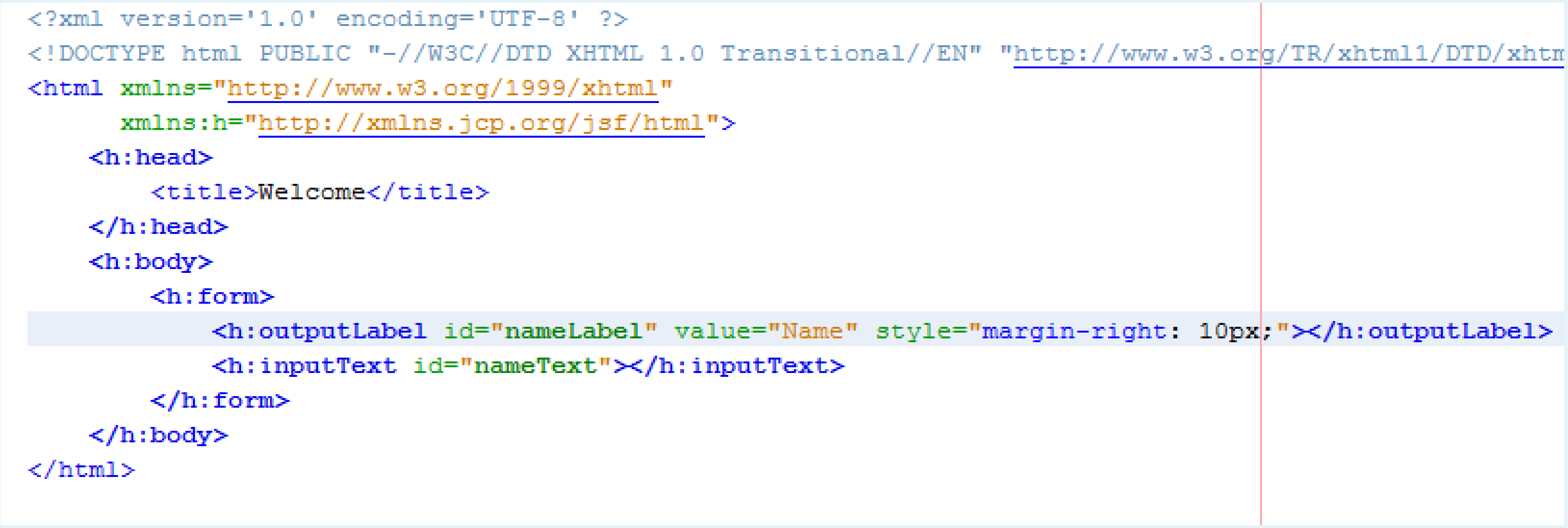
- Syntax of .xhtml file is very similar to HTML syntax, and basically it is HTML, the difference is that you can append dynamic content in .xhtml file.
- Before JSF page is sent to client (browser) the Java EE server generates an .html file from the dynamic data you define in the page.

JSF

- Creating Uis with JSF is done with components that are defined in JSF HTML tag library.
- This library contains all the basic components for building the UI.
- See more from references:TAG Library

JSF

- Appending Label and TextField:



```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtm
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText>
        </h:form>
    </h:body>
</html>
```

JSF

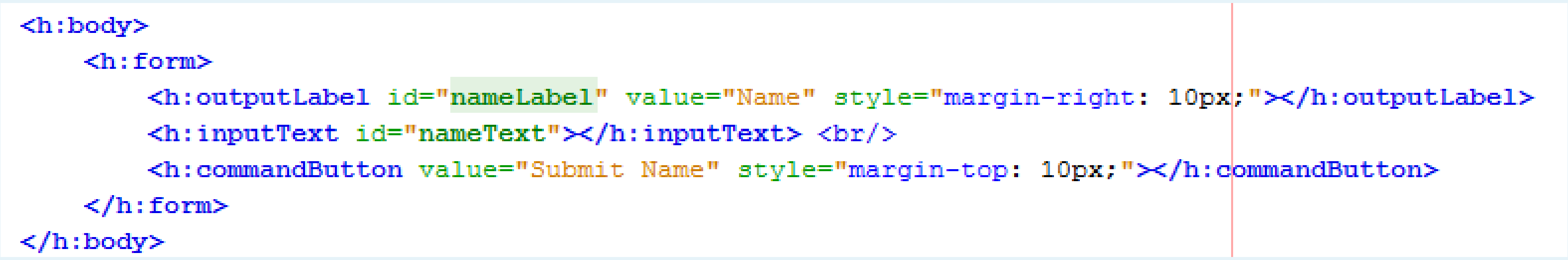
- Note in the previous example that form elements are inside the form tag.
- JSF actually forces you to do this, if you don't put form elements (input elements) inside the form tag, you will get an error displayed in generated HTML page (try it yourself if you don’t belive).

JSF

- 5 minute exercise
  - Append an command button in previous example. Set it below the label and text field.
  - Set the button text to ”Submit Name”
  - Make some margin between button and label
  - What is the difference in button label and value attributes?
  - See the Tag library documentation for help.
- Run the application to see that is works as specified.
- TIP! When ever you make changes to JSF file just save the file after changes. GlassFish deploys all changes automatically for you. You only need to refresh the browser window!

JSF

- As you see we can use ”normal” HTML elements inside the JSF file. Here we use <br/> element for making a line break.



```
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;"></h:commandButton>
    </h:form>
</h:body>
```

JSF

- One important point to take notice of is that JSF contains two types of form elements
    - Input
    - Output
- The input fields can only set the data
- The output fields can only read data
- This is important at the moment when we append Java Beans to our project and set data from these objects to our UI components.

JSF Basic Navigation

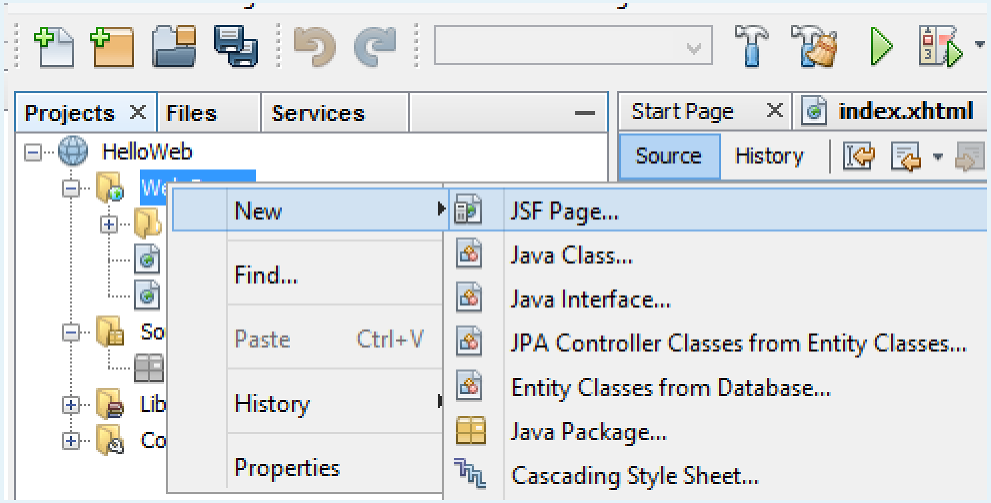
- Talking about web applications we can agree that there is probably more than one page in it.
- User wants to navigate between these pages and make some actions.
- JSF implements navigation rules provided by JSF Framework which describe which view is to be shown when a button or link, or some other element is clicked on the page.

JSF Basic Navigation

- There are many ways to define navigation in JSF
  - Navigation rules can be defined in JSF configuration file named faces-config.xml.
  - Navigation rules can be defined in managed beans.
  - Navigation rules can contain conditions based on which 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.

JSF Basic Navigation

- First of all we need another page in our application.
- Right mouse click on Web Pages folder in project tree. Select New ->JSF Page

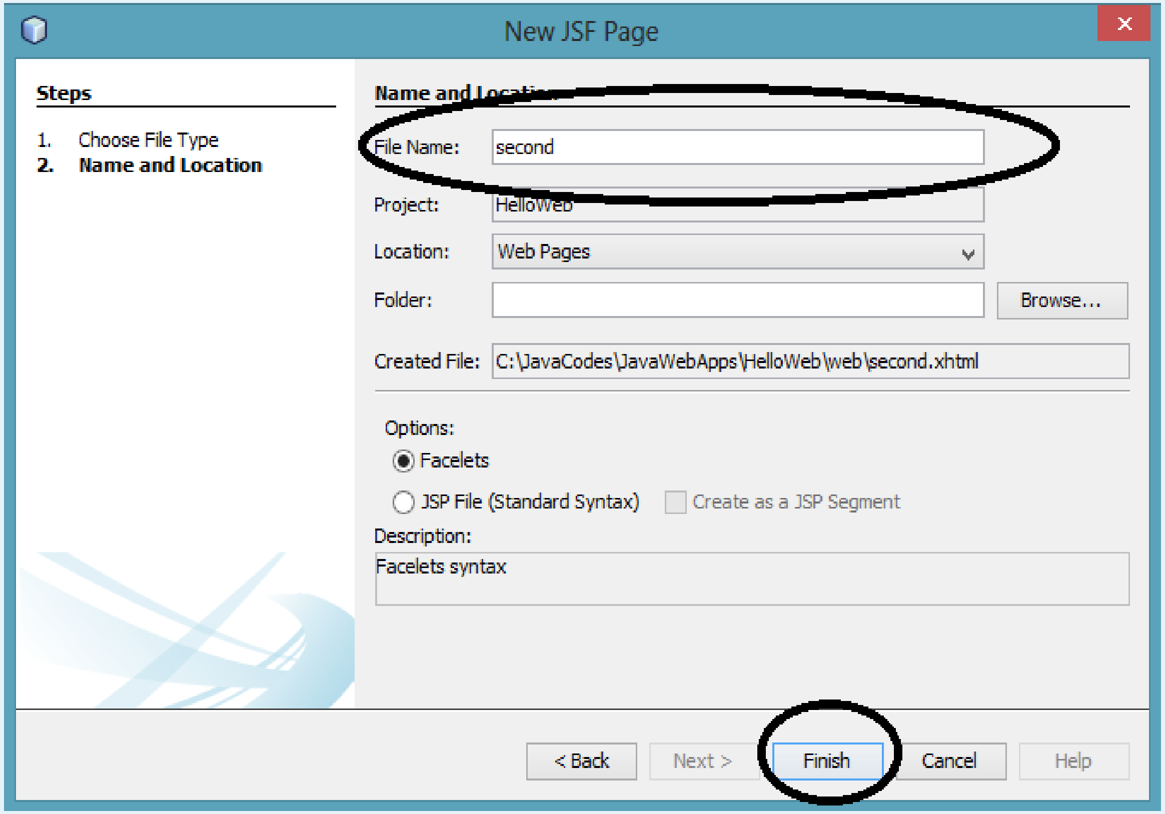


JSF Basic Navigation

- TIP! If JSF page is not available in list select New->Other...
- From *Categories* select *JavaServer Faces*
- From *File Types* select *JSF Page*
- Press Next button.

JSF Basic Navigation

- Give a name for your JSF page and press Finish.



New JSF Page

Steps

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

Name and Location

File Name: second

Project: HelloWeb

Location: Web Pages

Folder:

Created File: C:\JavaCodes\JavaWebApps\HelloWeb\web\second.xhtml

Options:

● Facelets

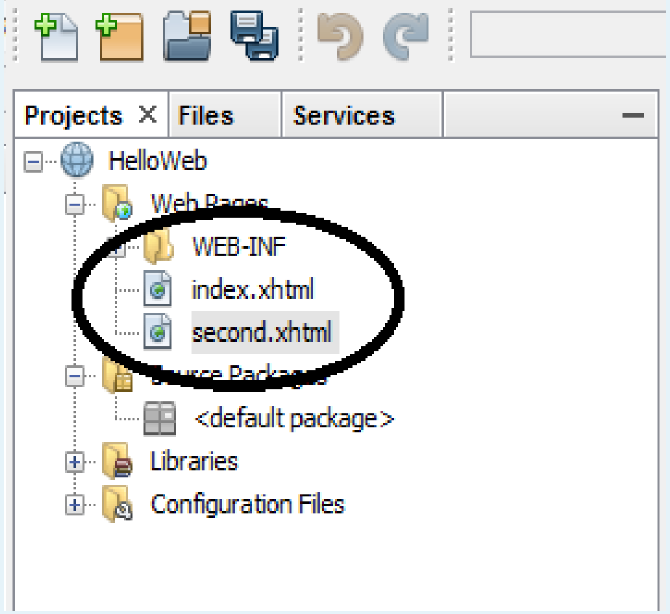
○ JSP File (Standard Syntax) □ Create as a JSP Segment

Description:

Facelets syntax

JSF Basic Navigation

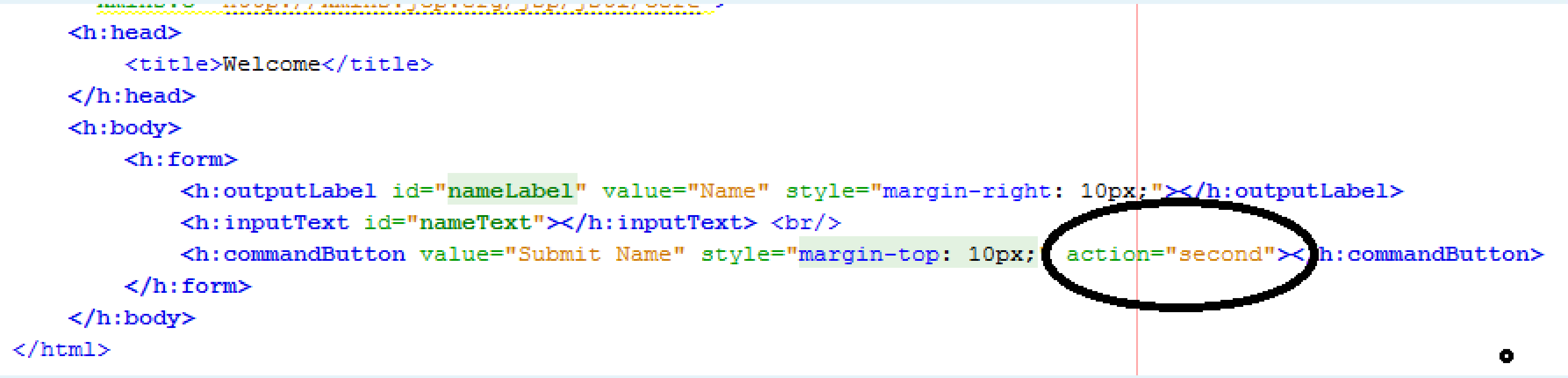
- Now you should have two JSF pages in your project.



Implicit Navigation

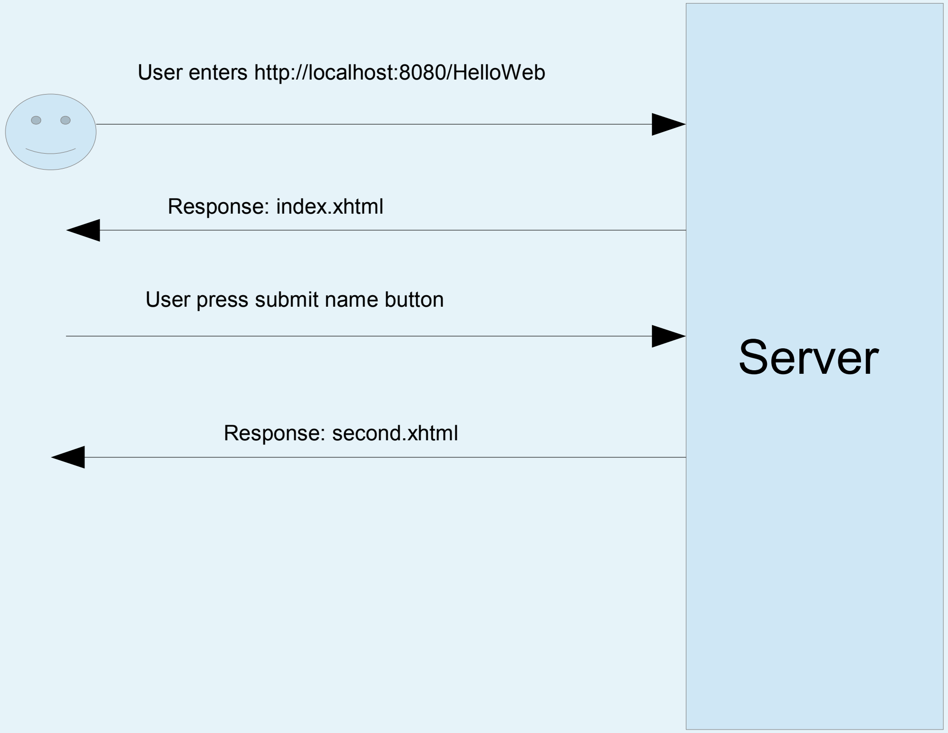
- You can define an *action* attribute for command button or a link.
- When user press the link or command button JSF automatically searches a .xhtml file with the same name defined in action attribute.
- For example navigating in our application from index.xhtml to second.xhtml

Implicit Navigation



```
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText> <br/>
            <h:commandButton value="Submit Name" style="margin-top: 10px;" action="second"></h:commandButton>
        </h:form>
    </h:body>
</html>
```

Sequence



User enters http://localhost:8080/HelloWeb

Response: index.xhtml

User press submit name button

Response: second.xhtml

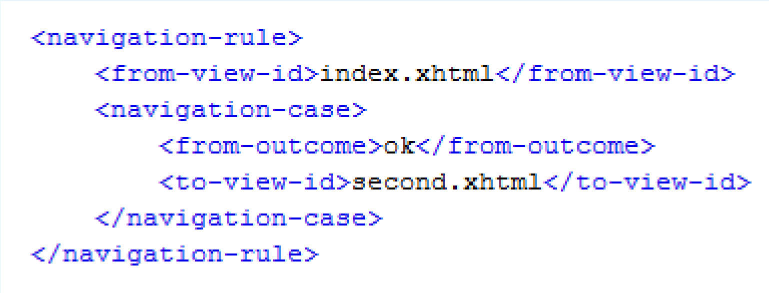
Server

Navigation Rules

- If navigation from one page to another requires more complex logic, then one possibility is to create a navigation rule file.
- Creating a config file: Right click over the project name in project tree.
- Select New->Other
- From *Categories* Select JavaServer Faces
- From *File Types* select J*SF Faces configuration.*
- *Press Next and the press Finish (don't change the file name)*

Navigation Rules

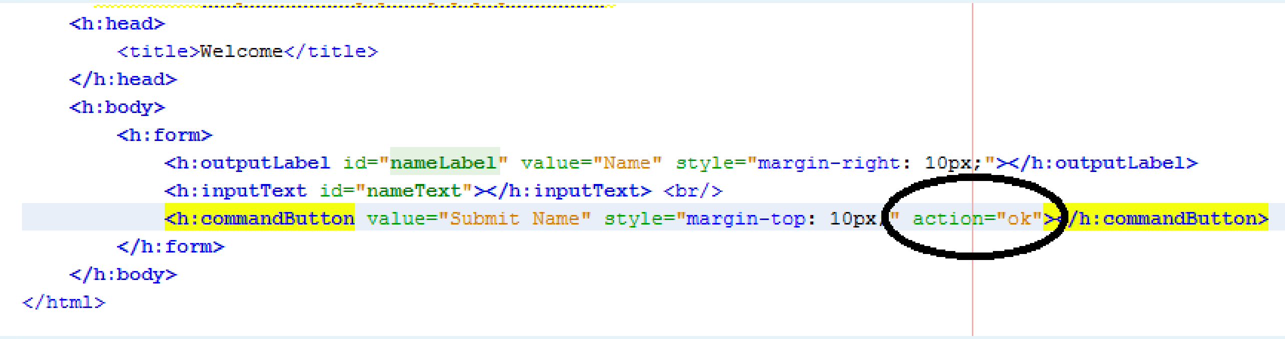
- The faces-config.xml file can contain different configurations for our project, like localization info or the navigation rules. The rules are defined by using XML tags.



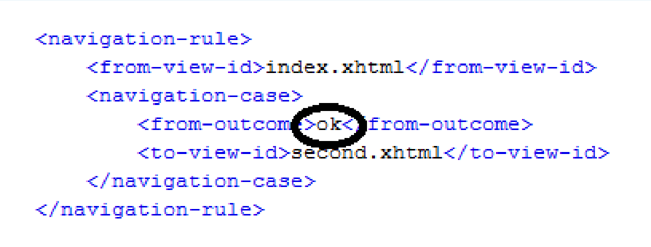
```
<navigation-rule>
    <from-view-id>index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>ok</from-outcome>
        <to-view-id>second.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Navigation Rules

- The <from-view-id> field contains the .xhtml file name, from where the navigation starts. The <to-view-id> field contains the target view file name. The from-outcome tag contains the name of the action where navigation starts.



```
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText> <br/>
            <h:commandButton value="Submit Name" style="margin-top: 10px;" action="ok"></h:commandButton>
        </h:form>
    </h:body>
</html>
```



```
<navigation-rule>
    <from-view-id>index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>ok</from-outcome>
        <to-view-id>second.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Exercise

- Define the faces-config.xml file in your project.
- Define one navigation rule where you navigate from index.xhtml page to your second.xhtml page.

Auto navigation in Managed Bean

- You can also define navigation with managed bean.
- Few things you should know about managed beans before you use them...

Introduction to 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 or some other framework.*
- The managed bean contains the getter and setter methods, business logic or even a backing bean (a bean contains all the HTML form value).

Introduction to Managed Beans

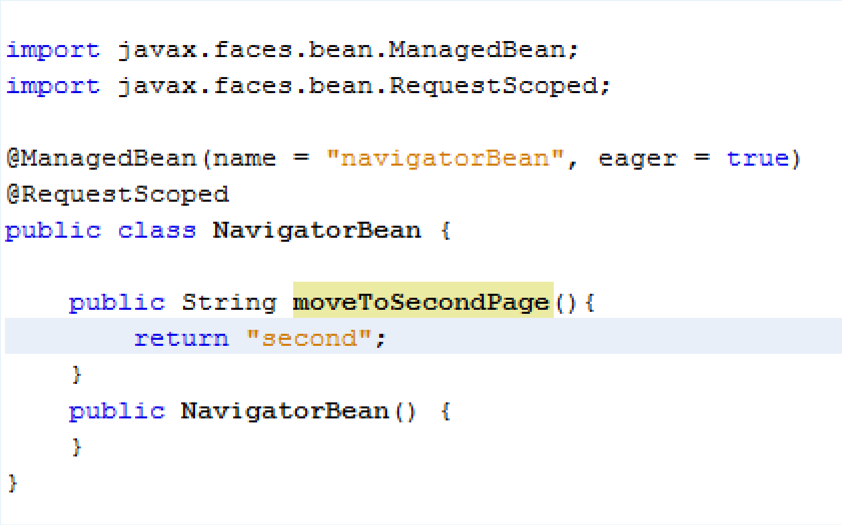
- *Managed beans works as Model for UI component.*
- *In JSF 1.2,a managed bean had to register it in JSF configuration file such as faces-config.xml.*
- *From JSF 2.0 onwards, Managed beans can be easily registered using annotations. This approach keeps beans and there registration at one place and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

Introduction to Managed Beans

- @ManagedBean Annotation
  - 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.
  - 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.

Example

- Just create a regular java class file. Use annotations to make class as an manage bean.



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

@ManagedBean(name = "navigatorBean", eager = true)
@RequestScoped
public class NavigatorBean {

    public String moveToSecondPage(){
        return "second";
    }
    public NavigatorBean() {
    }
}
```

Managed Beans

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

Managed Beans

Scope	Description
@RequestScoped	Bean lives as long as the HTTP request-response lives. It get created upon a HTTP request and get destroyed when the HTTP response associated with the HTTP request is finished.
@NoneScoped	Bean lives as long as a single EL evaluation. It get created upon an EL evaluation and get destroyed immediately after the EL evaluation.
@ViewScoped	Bean lives as long as user is interacting with the same JSF view in the browser window/tab. It get created upon a HTTP request and get destroyed once user postback to a different view.
@SessionScoped	Bean lives as long as the HTTP session lives. It get created upon the first HTTP request involving this bean in the session and get destroyed when the HTTP session is invalidated.
@ApplicationScoped	Bean lives as long as the web application lives. It get 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 get 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.

Managed Beans

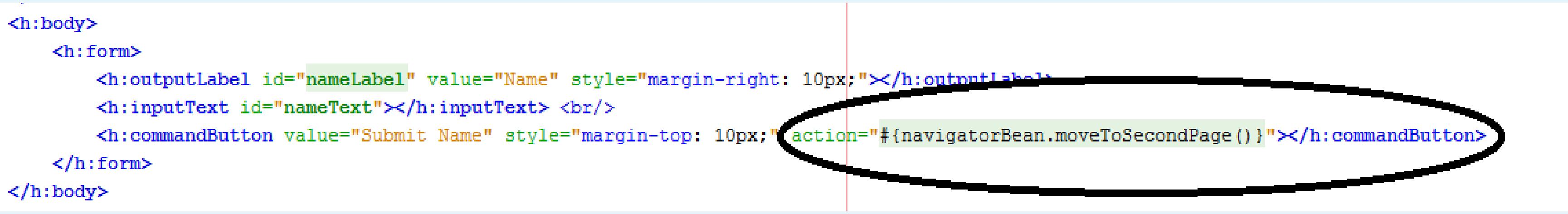
- @ManagedProperty Annotation
  - JSF is a simple static Dependency Injection(DI) framework.Using @ManagedProperty annotation a managed bean's property can be injected in another managed bean.

The Expression Language (EL)

- Now that we have a bean and JSF view we need to bind the bean property to JSF file.
- For value binding the universal Expression Language (EL) is used (to access bean and / or methods).

Example

- Calling NavigatorBean method when button in JSF page is pressed.

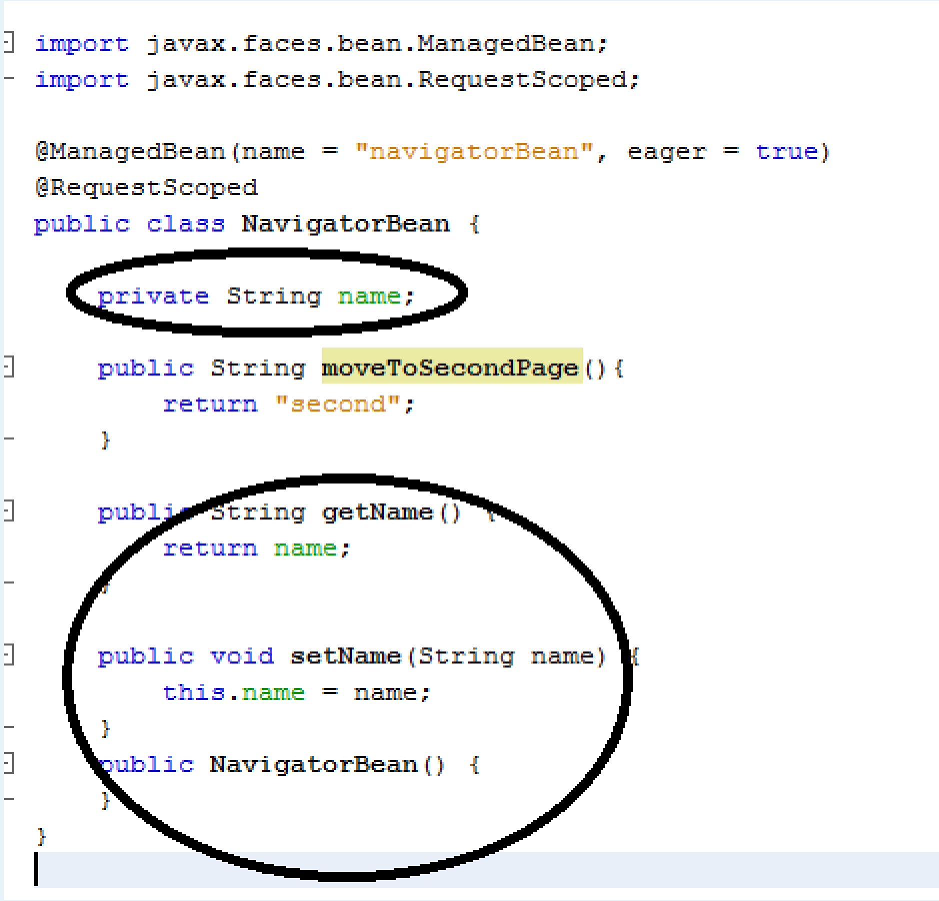


```
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;" action="#{navigatorBean.moveToSecondPage()}"></h:commandButton>
    </h:form>
</h:body>
```

Passing Data

- Passing data between pages is done by using managed beans.
- You define a property for your ManageBean class and get and set methods for it.
- Append new property for you bean.

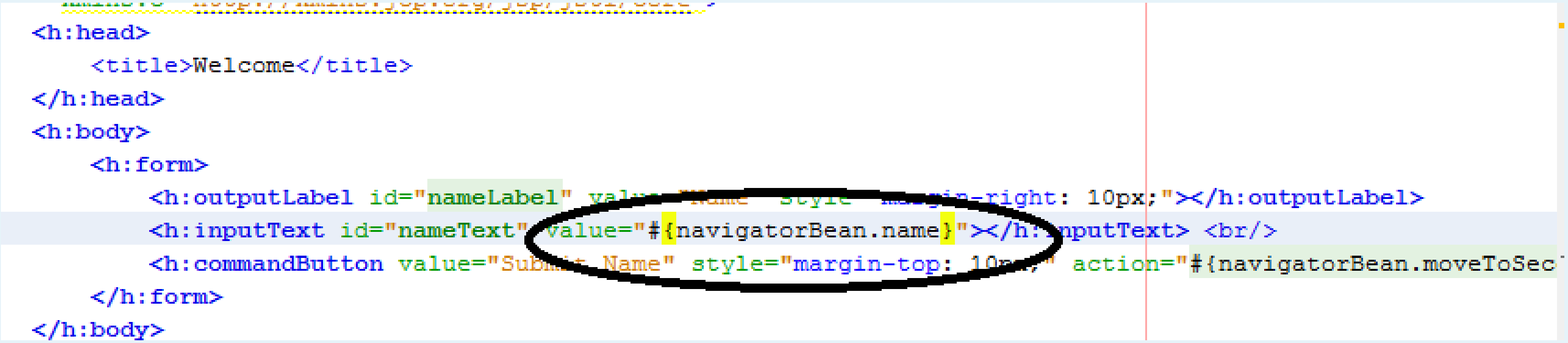
Passing Data



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

@ManagedBean(name = "navigatorBean", eager = true)
@RequestScoped
public class NavigatorBean {

    private String name;

    public String moveToSecondPage(){
        return "second";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public NavigatorBean() {
    }
}
```

Passing Data

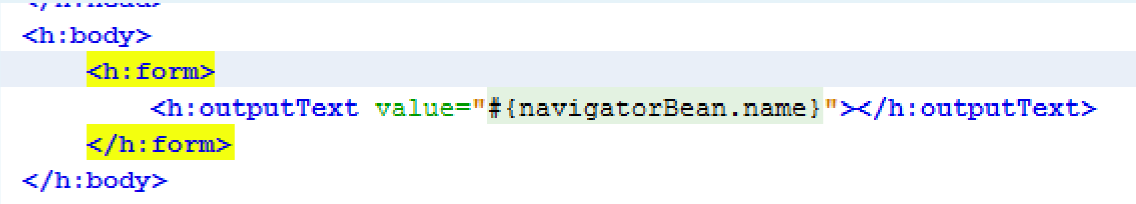
- In index.xhtml define the value for your inputText element. This will store what ever is typed to the field in our managed bean when form is submitted (the button is pressed).



```
  xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
<h:head>
    <title>Welcome</title>
</h:head>
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText" value="#{navigatorBean.name}"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;" action="#{navigatorBean.moveToSec
    </h:form>
</h:body>
```

Passing Data

- In second.xhtml define an outputText field. From value field define the name attribute from our manage bean class. The ouputText field will call automatically the get function from class to retrieve the name attribute value.



```
</h:head>
<h:body>
    <h:form>
        <h:outputText value="#{navigatorBean.name}"></h:outputText>
    </h:form>
</h:body>
```

JSF

- We have discussed about tiers and containers in JavaEE application and what these tiers and containers consist of.
- JavaServer Faces (JSF) is a Java-based web application framework intended to simplify *development integration of web-based user interfaces*.
- This part of the application belongs to the web tier and is assembled to web container module.

JSF

- JavaServer Faces (JSF) is a MVC web framework that simplifies the construction of user interfaces (UI) for server-based applications by using reusable UI components in a page.
  - Connect UI widgets to data sources
  - Connect widgets to *server-side event handlers*

JSF Benefits

- Provides reusable UI components
- Easy data transfer between UI components
- Manages UI state across multiple server requests.
- Enables making custom UI components
- Wiring client side event to server side application code

JSF UI component model

- JSF provides developers capability to create Web application from collections of UI components that can render themselves in different ways in different clients.
- Client can be HTML browser in desktop, browser in mobile phone, or any other device capable of rendering HTML.

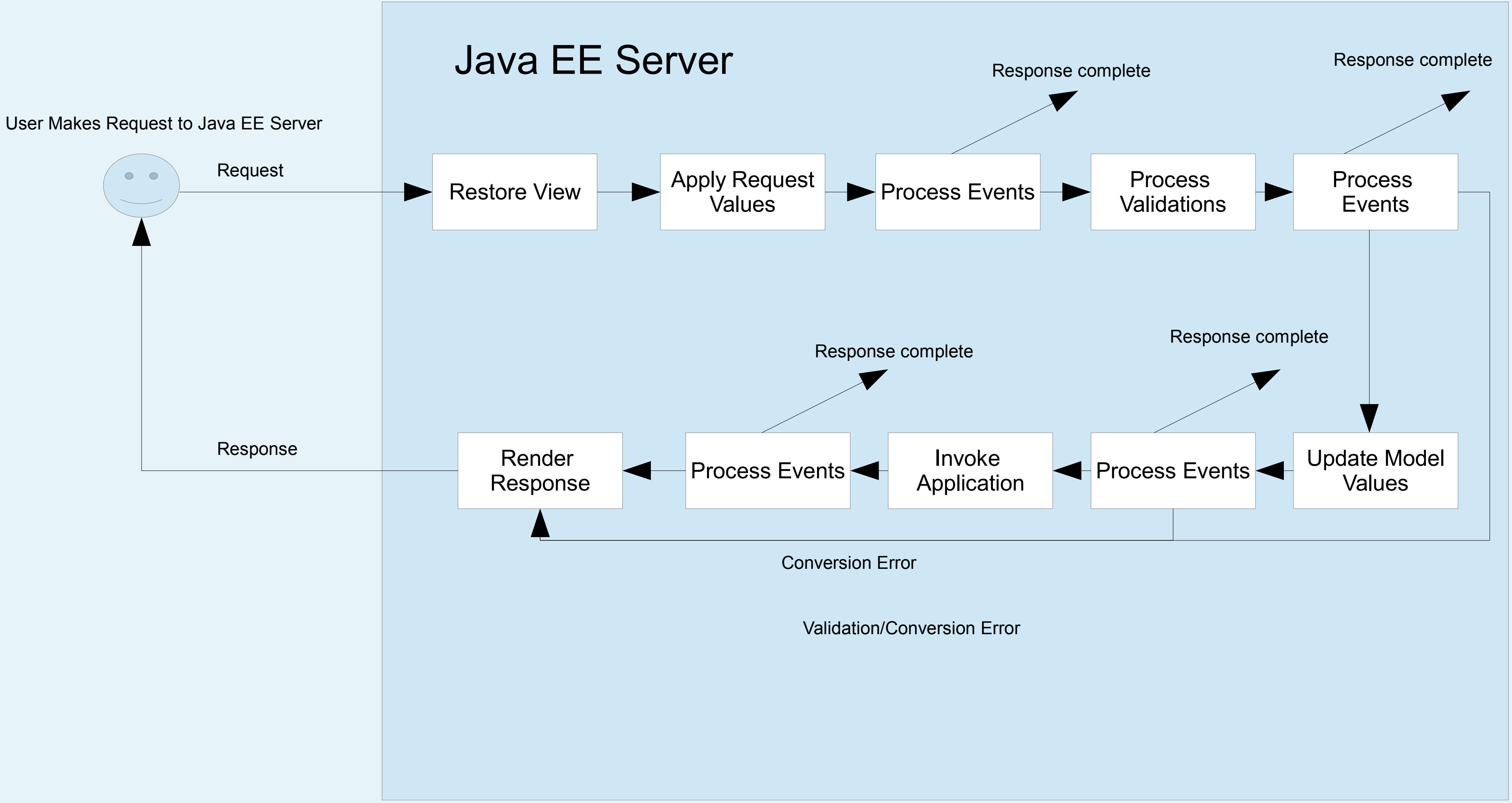
JSF

- JSF provides:
  - Core library
  - A set of base UI components - standard HTML input elements
  - Possibility to create custom components.
  - Multiple rendering capabilities that enable JSF UI components to render themselves differently depending on the client types.

JSF Application Lifecycle

- JSF application lifecycle consist of six phases:
  - Restore view phase
  - Apply request values phase
  - Process validations phase
  - Update model values phase
  - Invoke application phase
  - Render response phase

Basic Application Flow



Restore View Phase

- JSF begins the restore view phase when there is an incoming request from client.
- In this phase JSF builds the view and wires event handlers and validator to UI components.
- Then it saves the instance of view in FaceContext instance.
- The FacesContext instance will now contain all the needed information required to process a request.

Apply Request Values Phase

- After the component tree is created/restored, each component in component tree uses 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.*

Process Validation Phase

- During this phase, the JSF processes all validators registered on 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, the JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and display the same page again with the error message.

Update Model Values Phase

- After the JSF checks that the data is valid, it walks over the component tree and set the corresponding server-side object properties to the components' local values. The JSF will update the bean properties corresponding to input component's value attribute.
- If any updateModels methods called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Invoke Application Phase

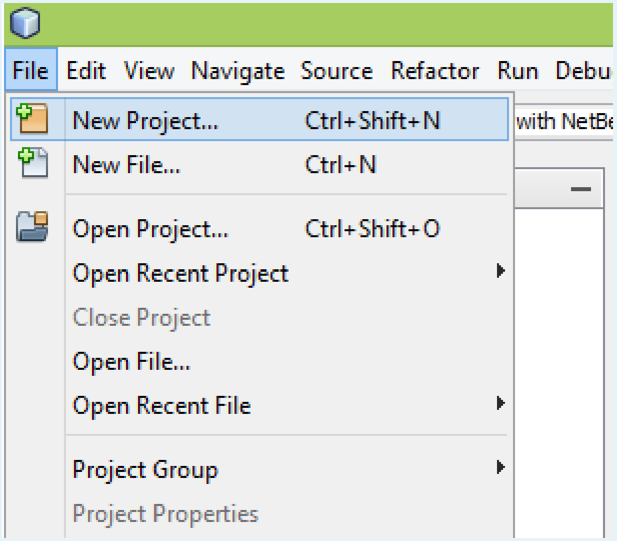
- During this phase, the JSF handles *any application-level events, such as submitting a form / linking to another page*.

Render Response Phase

- During this phase, the 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 the JSP container executes 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.

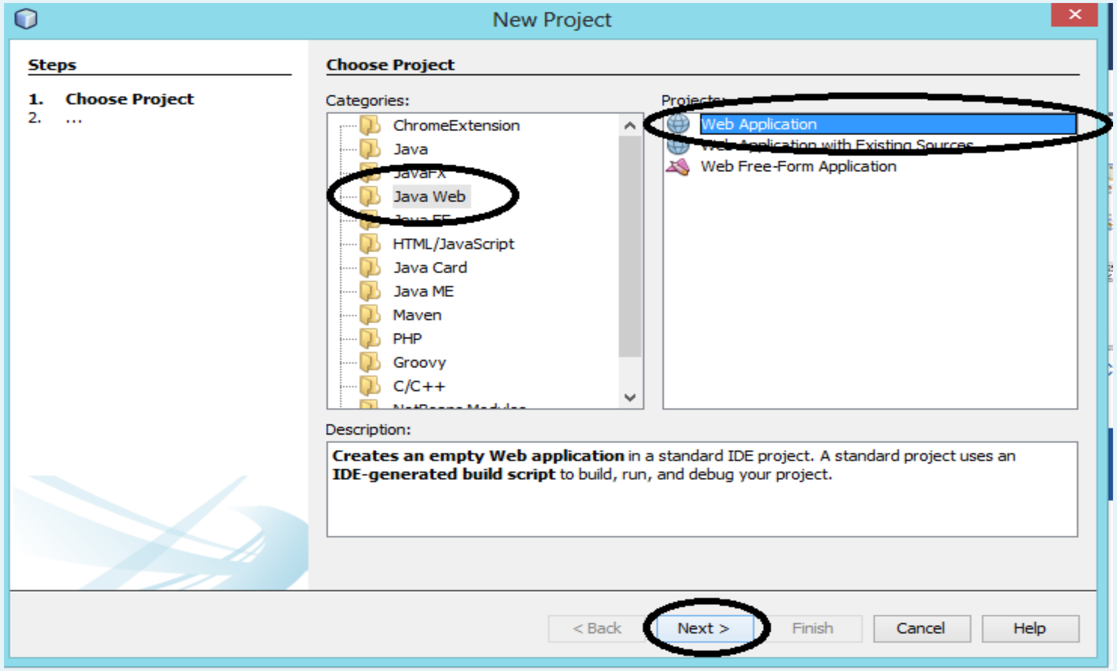
Creating Web Application Project With Java Server Faces Framework

- Launch NetBeans IDE.
- Select File-> New Project



First Project

- From *Categories* select Java Web
- From *Projects* select Web application
- Press Next

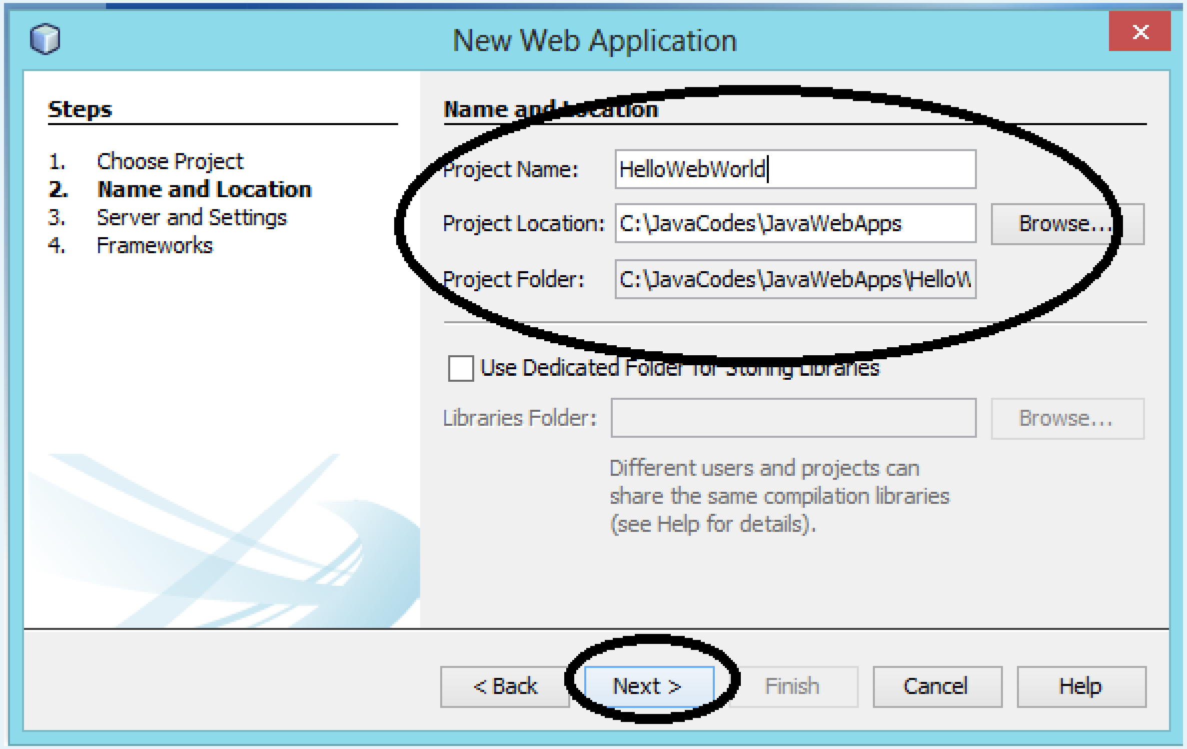


1. Choose Project
2. ...

Creates an empty Web application in a standard IDE project. A standard project uses an **IDE-generated build script** to build, run, and debug your project.

First Project

- Give name and location for your project.
- Press Next



New Web Application

Steps

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

Name and Location

Project Name: HelloWebWorld

Project Location: C:\JavaCodes\JavaWebApps

Project Folder: C:\JavaCodes\JavaWebApps\HelloW

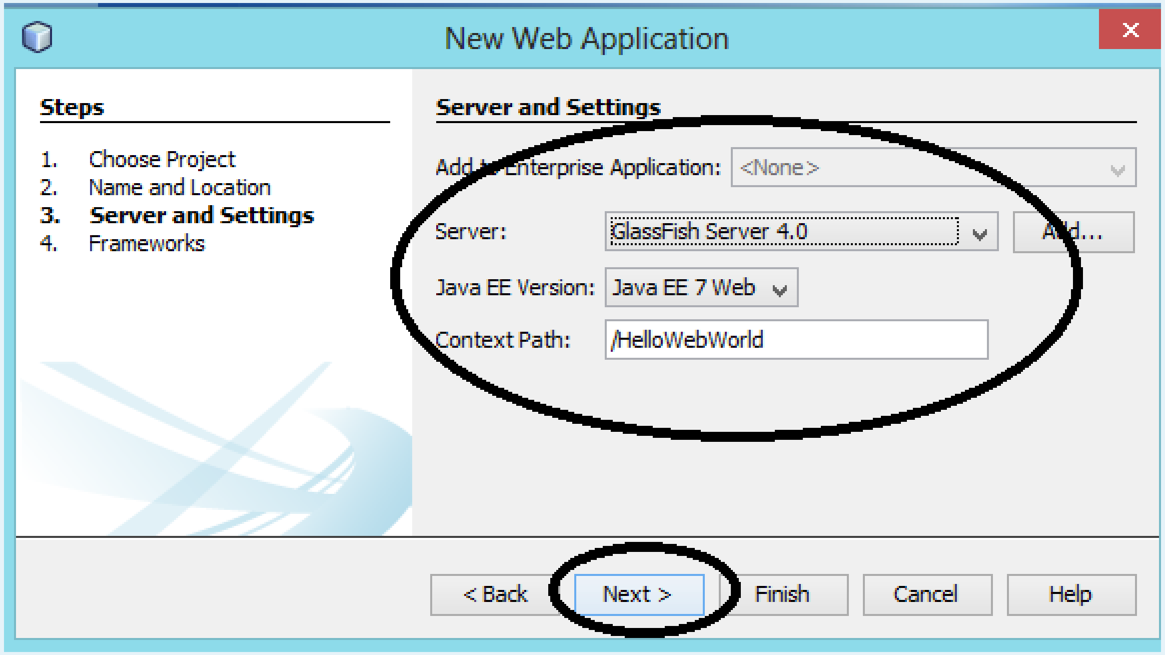
□ Use Dedicated Folder for Storing Libraries

Libraries Folder:

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

First Project

- Define the project settings. Makes sure that GlassFish server 4.0 is selected and JavaEE version is 7.0
- Press Next



New Web Application

Steps

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

Server and Settings

Add to Enterprise Application: <None>

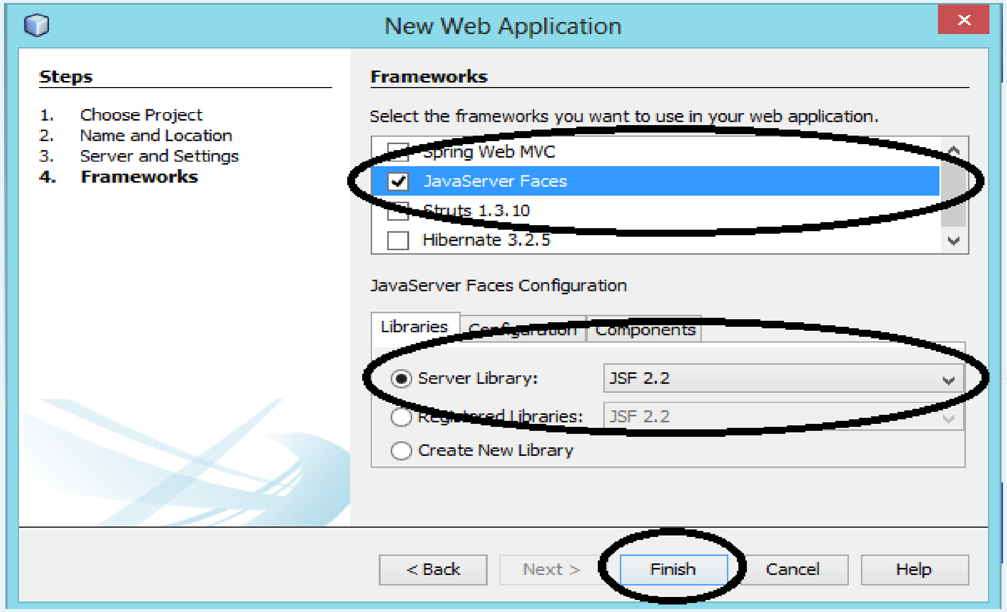
Server: GlassFish Server 4.0

Java EE Version: Java EE 7 Web

Context Path: /HelloWebWorld

First Project

- Select Java Server Faces from frameworks list.
- Makes sure that JSF version 2.2 is selected.
- Press Finish



New Web Application

Steps

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

Frameworks

Select the frameworks you want to use in your web application.

□ Spring Web MVC
☑ JavaServer Faces
□ Struts 1.3.10
□ Hibernate 3.2.5

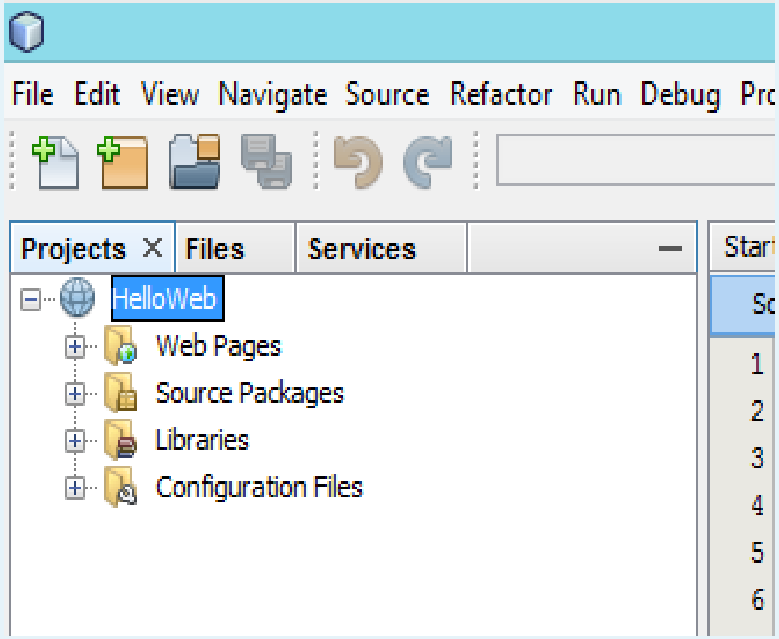
JavaServer Faces Configuration

Libraries | Configuration | Components

● Server Library: JSF 2.2
○ Registered Libraries: JSF 2.2
○ Create New Library

Project Content

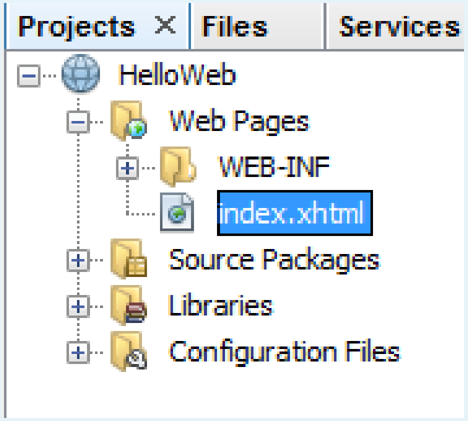
- The project tree in this case is as follow:



HelloWeb
- Web Pages
- Source Packages
- Libraries
- Configuration Files

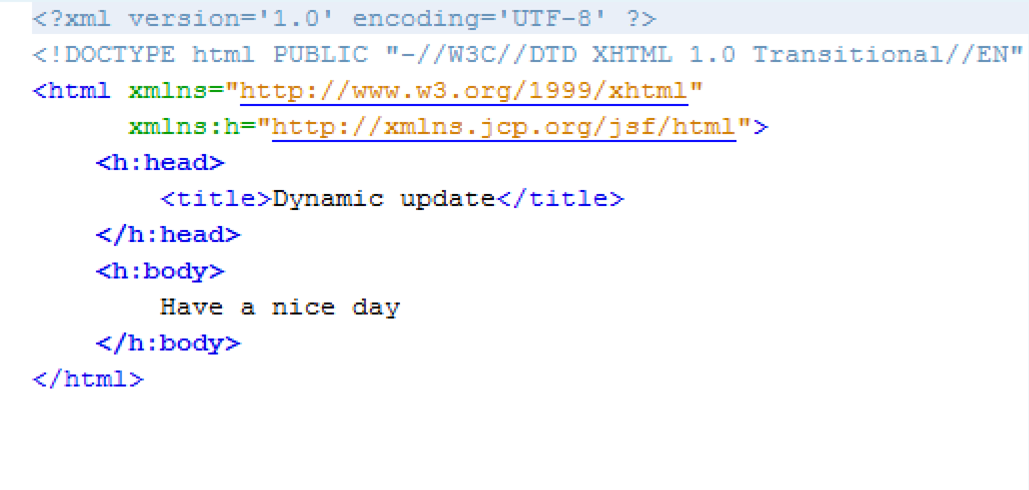
Project Content

- The Web Pages folder: Contains all web resources like JSF pages, HTML, CSS, Images and other resources
- If you open it you see WEB-INF folder and index.xhtml file



Project Content

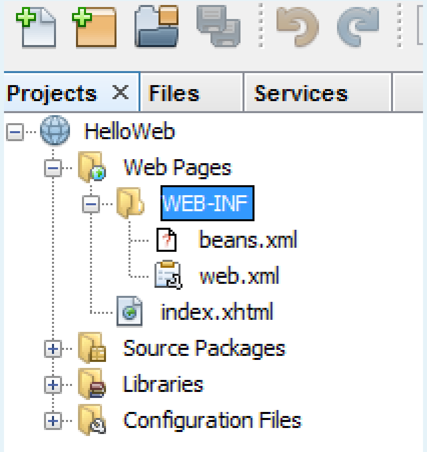
- Index.xhtml: This is your first JSF file in project. The content of file is as follow:



```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Dynamic update</title>
    </h:head>
    <h:body>
        Have a nice day
    </h:body>
</html>
```

Project Content

- WE-INF folder: Contains the project configuration files.
- If you open it you see two files web.xml and beans.xml

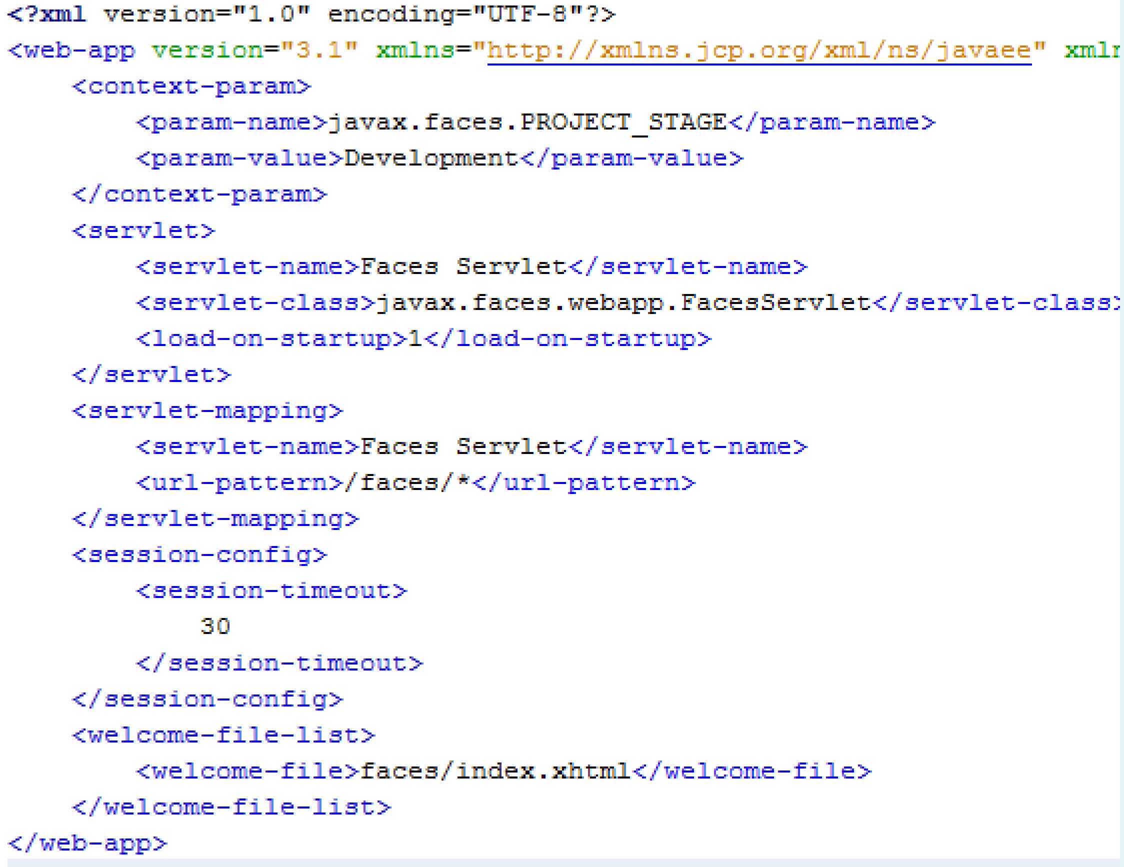


Project Content

- web.xml: Project main configuration file. Contains the needed Servlet mappings.
- In basic project you don't usually need to modify this file.
- beans.xml: Contains metadata about your JavaBeans defined in application. Because the basic project template don't create any beans for us, this file contains just basic XML data needed.

Project Content

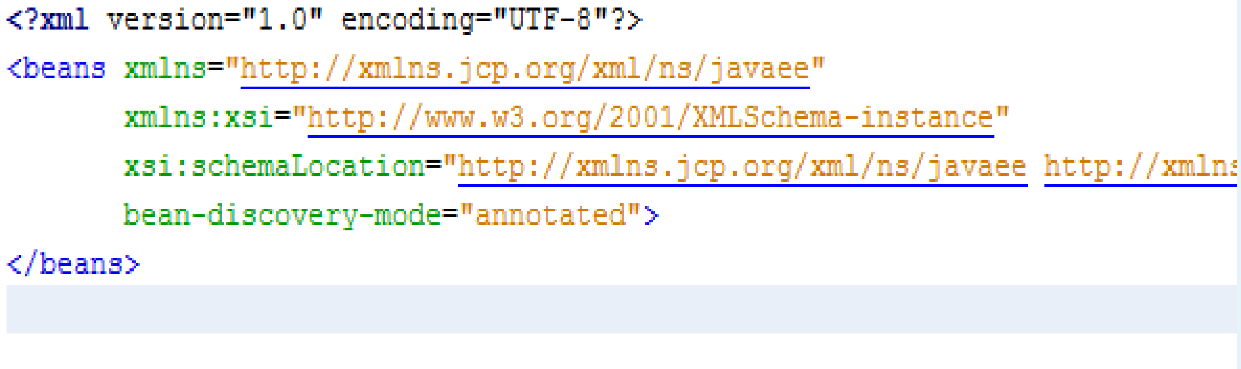
- web.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xml
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Project Content

- beans.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns
       bean-discovery-mode="annotated">
</beans>
```

Project Content

- Source Packages folder: Contains all the .java files of your project. In freshly created project this is empty.
- Configuration Files: Contains all the configuration files. By default contains the same files as WEB-INF + MANIFEST.MF file. This folder can contain also navigation rules file, localization files etc.

Web.xml

- If you open web.xml file you can see that <context-param> is set to *javax.faces.PROJECT_STAGE* and *Development*.
- When context is set to *development* it will provide many useful debugging information to let you track the bugs easily.
- For deployment, just change it to “Production“.

Web.xml

- The <servlet> element contains the servlet class which is set to *javax.faces.webapp.FacesServlet.*
- This just points to class that process the JavaServer Faces requests.
- The load-on-startup element indicates that this servlet should be loaded (instantiated and have its init() called) on the startup of the web application.
- The optional contents of these element must be an integer indicating the order in which the servlet should be loaded. If the value is a negative integer, or the element is not present, the container is free to load the servlet whenever it chooses.

Web.xml

- The servlet-mapping element defines a mapping between a servlet and a URL pattern.
- <servlet-name>:The name of the servlet to which you are mapping a URL pattern. This name corresponds to *the name you assigned a servlet in a <servlet> declaration tag.*

Web.xml

- <url-pattern>:Describes a pattern used to resolve URLs. The portion of the URL after the http://host:port + WebAppName is compared to the <url-pattern> by WebLogic Server. If the patterns match, the servlet mapped in this element will be called.
- Example:/foo/*

Web.xml

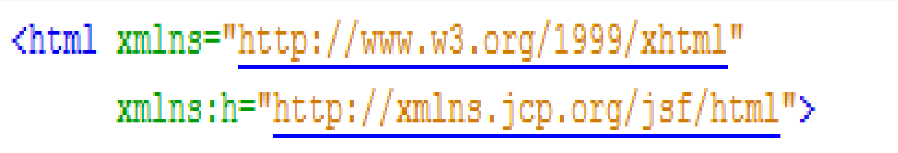
- The session-config element defines the session attributes for this Web application.
- <session-timeout>: The number of minutes after which sessions in this Web application expire.

Web.xml

- The optional welcome-file-list element contains an ordered list of welcome-file elements.
- When the URL request is a directory name, server serves the first file specified in this element. If that file is not found, the server then tries the next file in the list.

Index.html

- index.xhtml file is the default JSF page generated for us upon project creation.
- The first two lines are as follow:
-
-



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

- These lines adds the needed JSF tag library in html with prefix 'h' by default.
- This means if you want to use some element from JSF HTML tag library you need to prefix that element with letter *h* i.e.

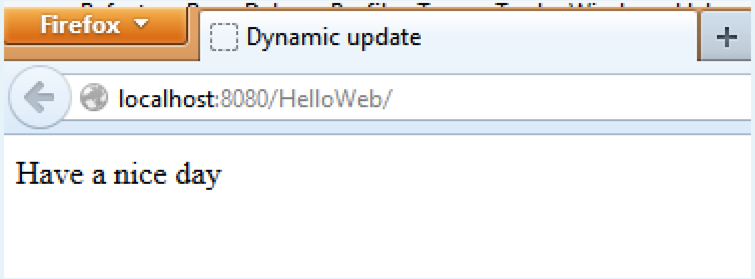
  <h:head>

Index.html

- The default template code contains the needed elements to display title and greeting message in browser.

Run the Application

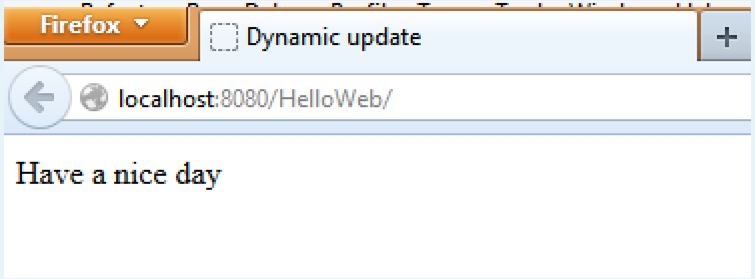
- To verify that our environment is working, we need to deploy the application to GlassFish server and check that the greeting message is rendered in our browser window.
- This is actually a very complex process, but thanks to framework YOU only need to press the green ”play” button in NetBeans IDE.
- What are you waiting? Press the button...
- If everything works fine the process kicks your default browser up and you should see the greeting text in browser window



Have a nice day

Run the Application

- To verify that our environment is working, we need to deploy the application to GlassFish server and check that the greeting message is rendered in our browser window.
- This is actually a very complex process, but thanks to framework YOU only need to press the green ”play” button in NetBeans IDE.
- What are you waiting? Press the button...
- If everything works fine the process kicks your default browser up and you should see the greeting text in browser window



Have a nice day

JSF

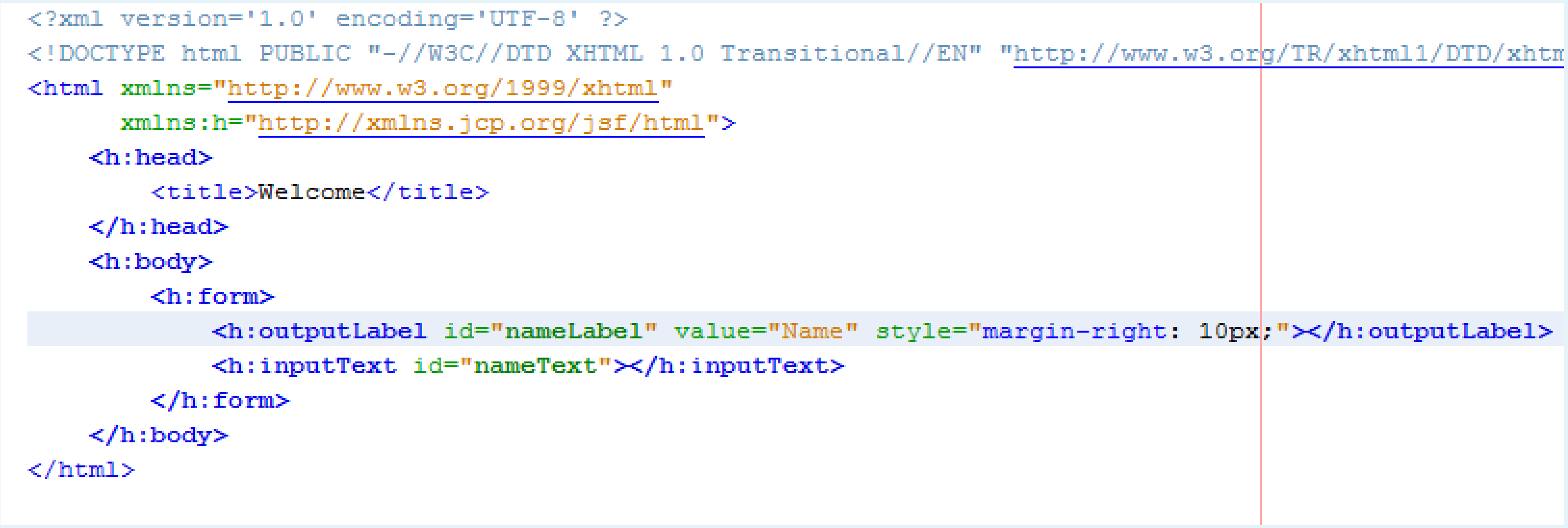
- Syntax of .xhtml file is very similar to HTML syntax, and basically it is HTML, the difference is that you can append dynamic content in .xhtml file.
- Before JSF page is sent to client (browser) the Java EE server generates an .html file from the dynamic data you define in the page.

JSF

- Creating Uis with JSF is done with components that are defined in JSF HTML tag library.
- This library contains all the basic components for building the UI.
- See more from references:TAG Library

JSF

- Appending Label and TextField:



```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtm
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText>
        </h:form>
    </h:body>
</html>
```

JSF

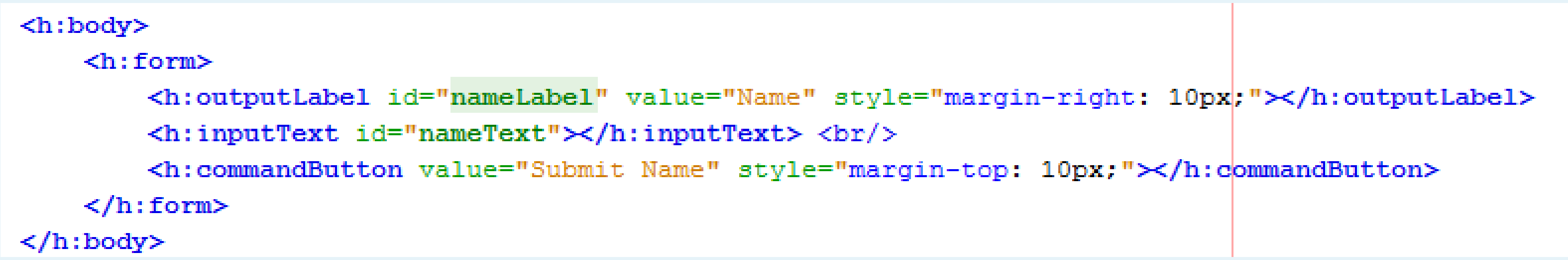
- Note in the previous example that form elements are inside the form tag.
- JSF actually forces you to do this, if you don't put form elements (input elements) inside the form tag, you will get an error displayed in generated HTML page (try it yourself if you don’t belive).

JSF

- 5 minute exercise
  - Append an command button in previous example. Set it below the label and text field.
  - Set the button text to ”Submit Name”
  - Make some margin between button and label
  - What is the difference in button label and value attributes?
  - See the Tag library documentation for help.
- Run the application to see that is works as specified.
- TIP! When ever you make changes to JSF file just save the file after changes. GlassFish deploys all changes automatically for you. You only need to refresh the browser window!

JSF

- As you see we can use ”normal” HTML elements inside the JSF file. Here we use <br/> element for making a line break.



```
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;"></h:commandButton>
    </h:form>
</h:body>
```

JSF

- One important point to take notice of is that JSF contains two types of form elements
  - Input
  - Output
- The input fields can only set the data
- The output fields can only read data
- This is important at the moment when we append Java Beans to our project and set data from these objects to our UI components.

JSF Basic Navigation

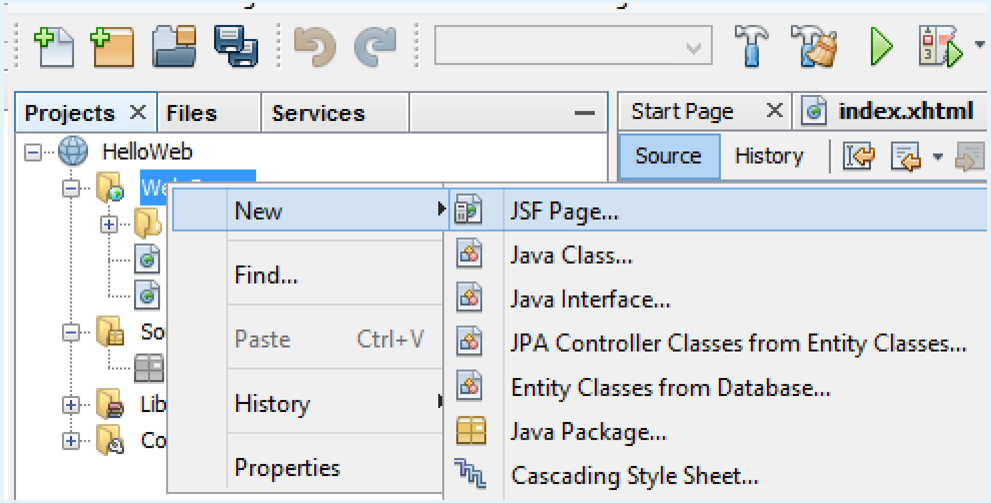
- Talking about web applications we can agree that there is probably more than one page in it.
- User wants to navigate between these pages and make some actions.
- JSF implements navigation rules provided by JSF Framework which describe which view is to be shown when a button or link, or some other element is clicked on the page.

JSF Basic Navigation

- There are many ways to define navigation in JSF
  - Navigation rules can be defined in JSF configuration file named faces-config.xml.
  - Navigation rules can be defined in managed beans.
  - Navigation rules can contain conditions based on which 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.

JSF Basic Navigation

- First of all we need another page in our application.
- Right mouse click on Web Pages folder in project tree. Select New ->JSF Page

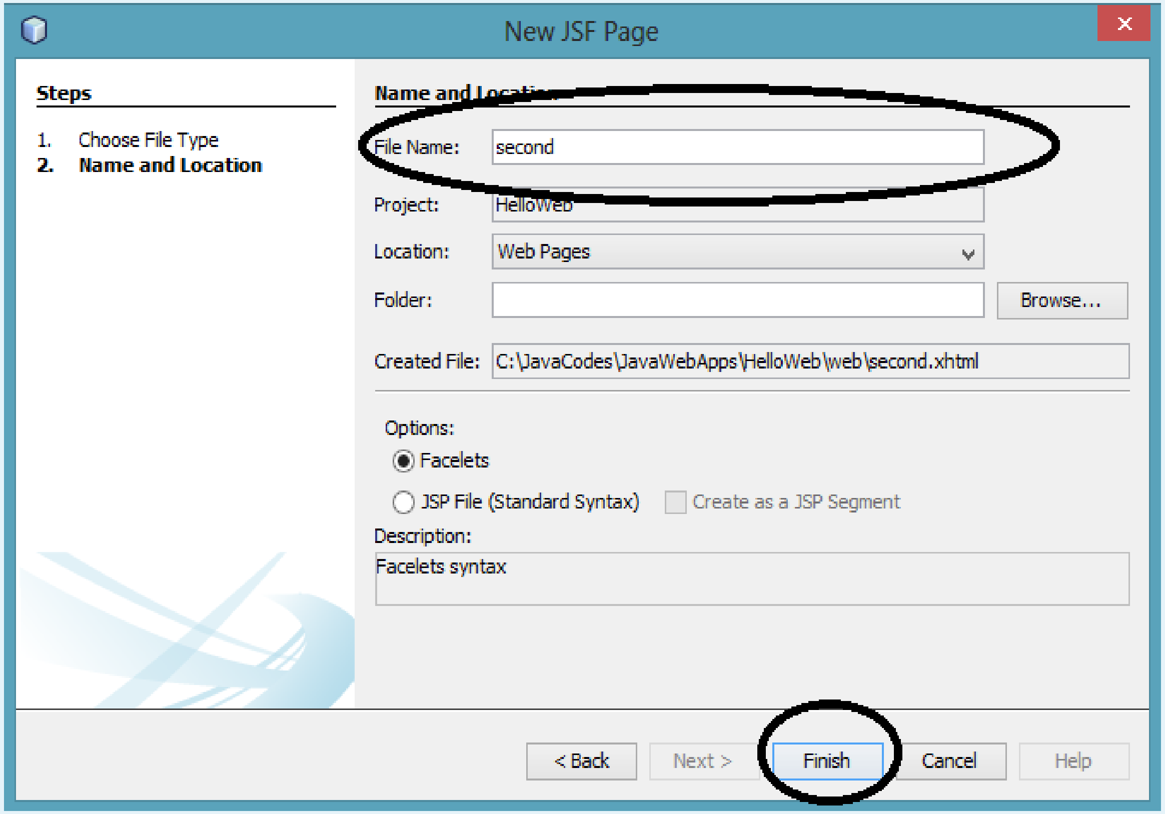


JSF Basic Navigation

- TIP! If JSF page is not available in list select New->Other...
- From *Categories* select *JavaServer Faces*
- From *File Types* select *JSF Page*
- Press Next button.

JSF Basic Navigation

- Give a name for your JSF page and press Finish.



New JSF Page

Steps

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

Name and Location

File Name: second

Project: HelloWeb

Location: Web Pages

Folder:

Created File: C:\JavaCodes\JavaWebApps\HelloWeb\web\second.xhtml

Options:

● Facelets

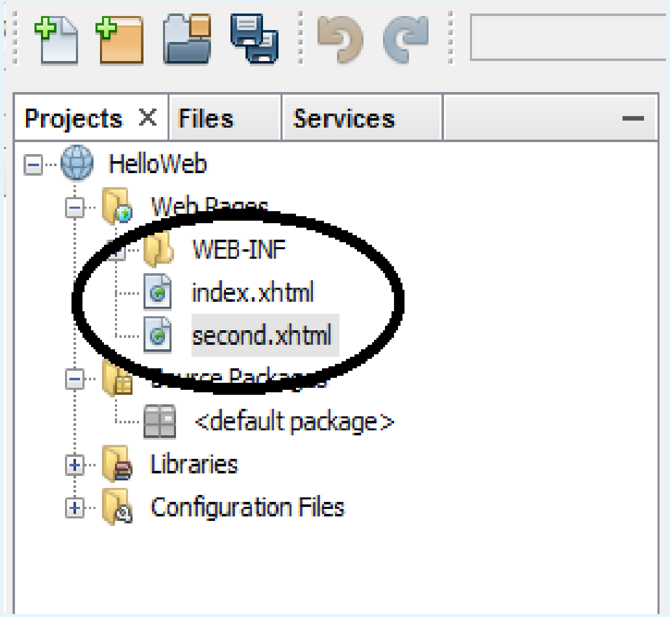
○ JSP File (Standard Syntax) □ Create as a JSP Segment

Description:

Facelets syntax

JSF Basic Navigation

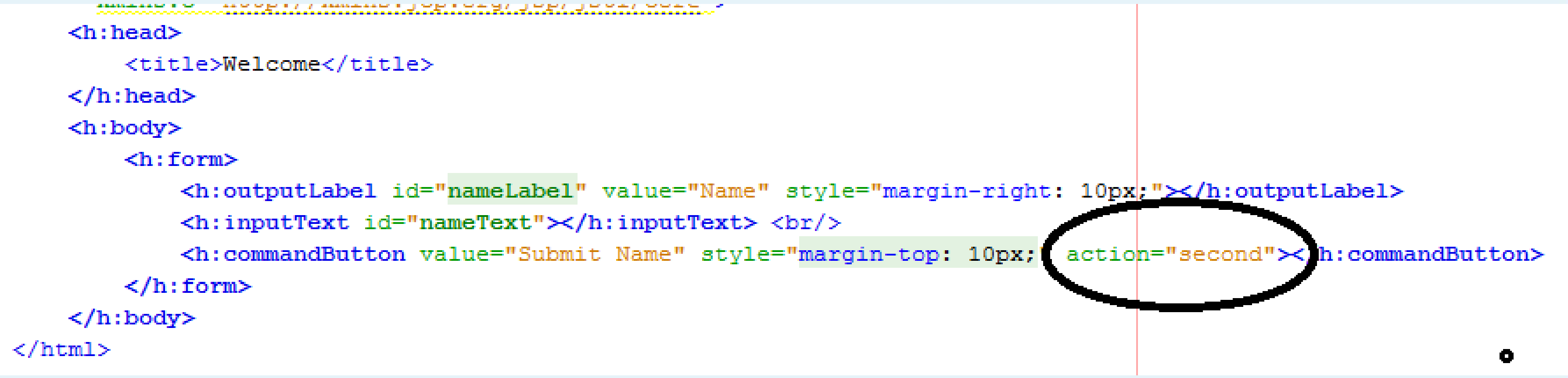
- Now you should have two JSF pages in your project.



Implicit Navigation

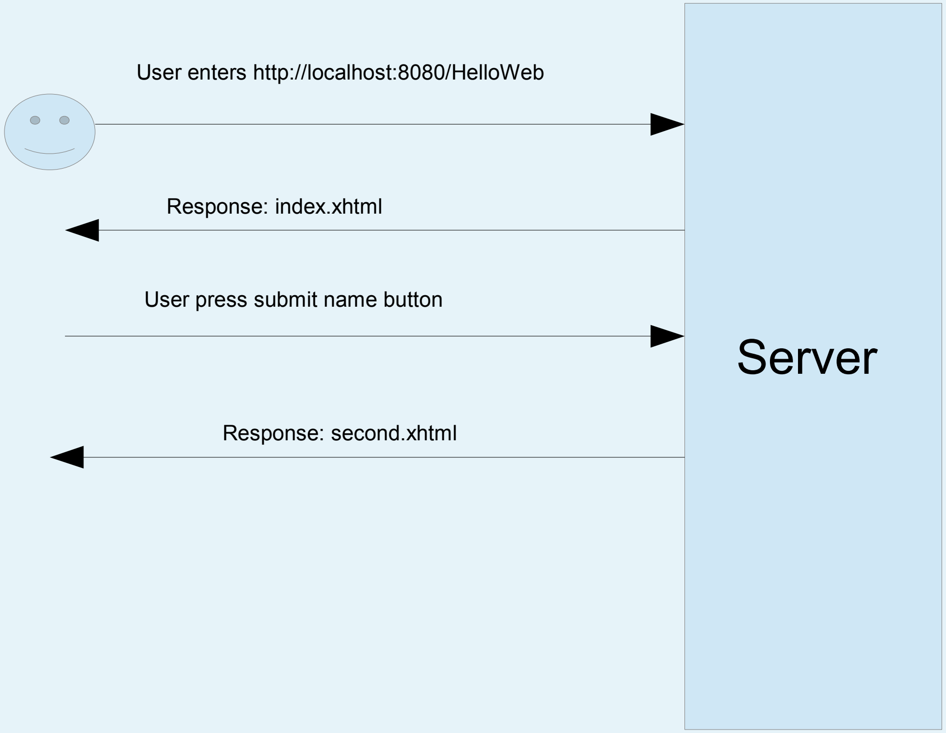
- You can define an *action* attribute for command button or a link.
- When user press the link or command button JSF automatically searches a .xhtml file with the same name defined in action attribute.
- For example navigating in our application from index.xhtml to second.xhtml

Implicit Navigation



```
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText> <br/>
            <h:commandButton value="Submit Name" style="margin-top: 10px;" action="second"></h:commandButton>
        </h:form>
    </h:body>
</html>
```

Sequence

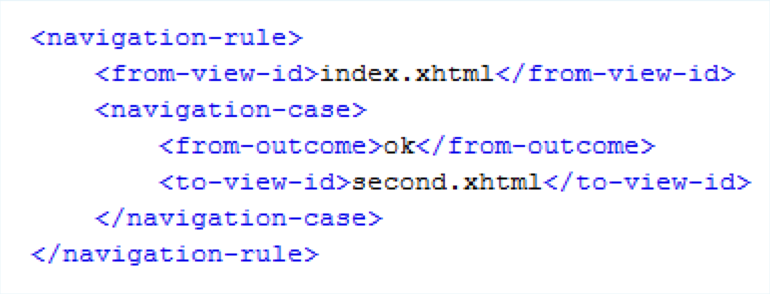

User enters http://localhost:8080/HelloWeb
Response: index.xhtml
User press submit name button
Response: second.xhtml
Server

Navigation Rules

- If navigation from one page to another requires more complex logic, then one possibility is to create a navigation rule file.
- Creating a config file: Right click over the project name in project tree.
- Select New->Other
- From *Categories* Select JavaServer Faces
- From *File Types* select J*SF Faces configuration.*
- *Press Next and the press Finish (don't change the file name)*

Navigation Rules

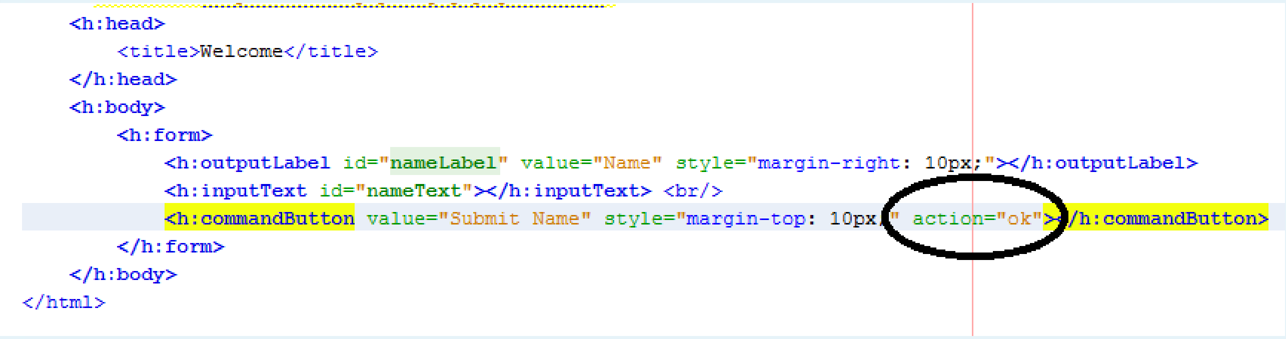
- The faces-config.xml file can contain different configurations for our project, like localization info or the navigation rules. The rules are defined by using XML tags.



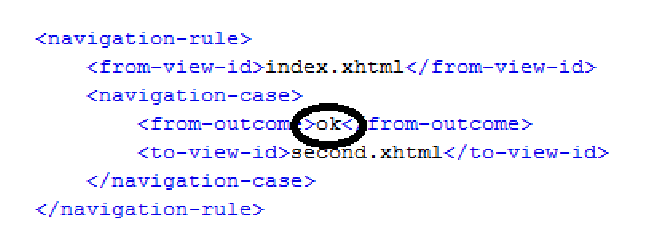
```
<navigation-rule>
    <from-view-id>index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>ok</from-outcome>
        <to-view-id>second.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Navigation Rules

- The <from-view-id> field contains the .xhtml file name, from where the navigation starts. The <to-view-id> field contains the target view file name. The from-outcome tag contains the name of the action where navigation starts.



```
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
            <h:inputText id="nameText"></h:inputText> <br/>
            <h:commandButton value="Submit Name" style="margin-top: 10px;" action="ok"></h:commandButton>
        </h:form>
    </h:body>
</html>
```



```
<navigation-rule>
    <from-view-id>index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>ok</from-outcome>
        <to-view-id>second.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Exercise

- Define the faces-config.xml file in your project.
- Define one navigation rule where you navigate from index.xhtml page to your second.xhtml page.

Auto navigation in Managed Bean

- You can also define navigation with managed bean.
- Few things you should know about managed beans before you use them...

Introduction to 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 or some other framework.*
- The managed bean contains the getter and setter methods, business logic or even a backing bean (a bean contains all the HTML form value).

Introduction to Managed Beans

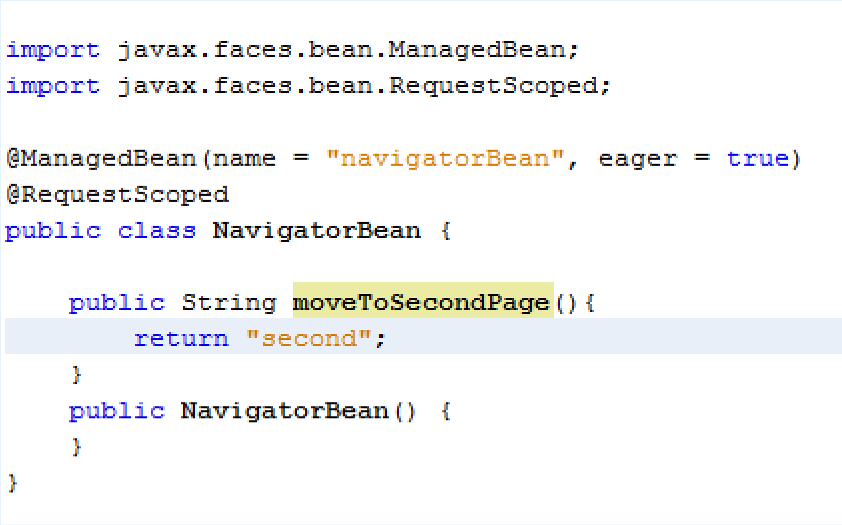
- *Managed beans works as Model for UI component.*
- *In JSF 1.2,a managed bean had to register it in JSF configuration file such as faces-config.xml.*
- *From JSF 2.0 onwards, Managed beans can be easily registered using annotations. This approach keeps beans and there registration at one place and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

Introduction to Managed Beans

- @ManagedBean Annotation
  - 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.
  - 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.

Example

- Just create a regular java class file. Use annotations to make class as an manage bean.



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

@ManagedBean(name = "navigatorBean", eager = true)
@RequestScoped
public class NavigatorBean {

    public String moveToSecondPage(){
        return "second";
    }
    public NavigatorBean() {
    }
}
```

Managed Beans

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

Managed Beans

Scope	Description
@RequestScoped	Bean lives as long as the HTTP request-response lives. It get created upon a HTTP request and get destroyed when the HTTP response associated with the HTTP request is finished.
@NoneScoped	Bean lives as long as a single EL evaluation. It get created upon an EL evaluation and get destroyed immediately after the EL evaluation.
@ViewScoped	Bean lives as long as user is interacting with the same JSF view in the browser window/tab. It get created upon a HTTP request and get destroyed once user postback to a different view.
@SessionScoped	Bean lives as long as the HTTP session lives. It get created upon the first HTTP request involving this bean in the session and get destroyed when the HTTP session is invalidated.
@ApplicationScoped	Bean lives as long as the web application lives. It get 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 get 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.

Managed Beans

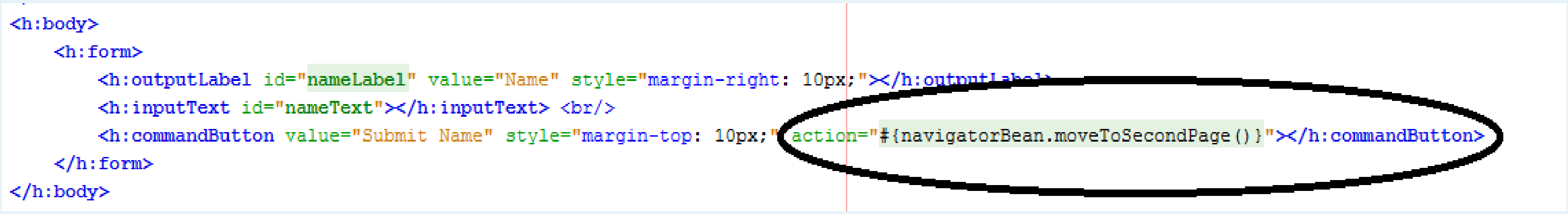
- @ManagedProperty Annotation
  - JSF is a simple static Dependency Injection(DI) framework.Using @ManagedProperty annotation a managed bean's property can be injected in another managed bean.

The Expression Language (EL)

- Now that we have a bean and JSF view we need to bind the bean property to JSF file.
- For value binding the universal Expression Language (EL) is used (to access bean and / or methods).

Example

- Calling NavigatorBean method when button in JSF page is pressed.

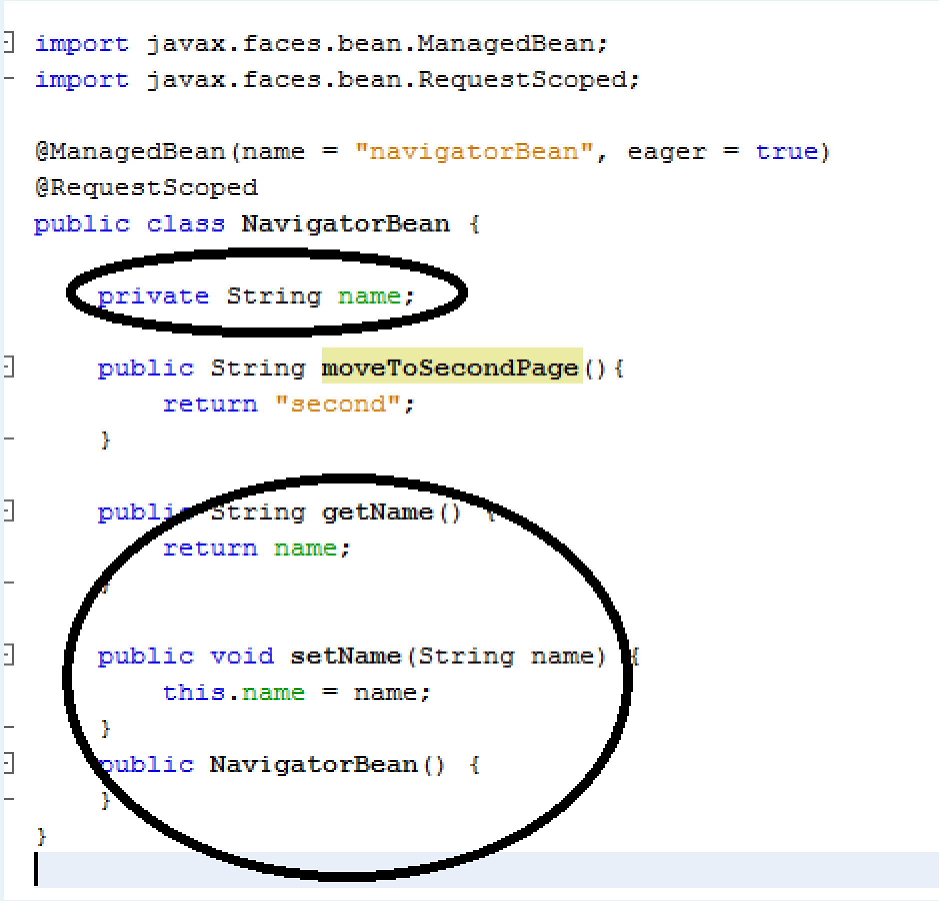


```
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;" action="#{navigatorBean.moveToSecondPage()}"></h:commandButton>
    </h:form>
</h:body>
```

Passing Data

- Passing data between pages is done by using managed beans.
- You define a property for your ManageBean class and get and set methods for it.
- Append new property for you bean.

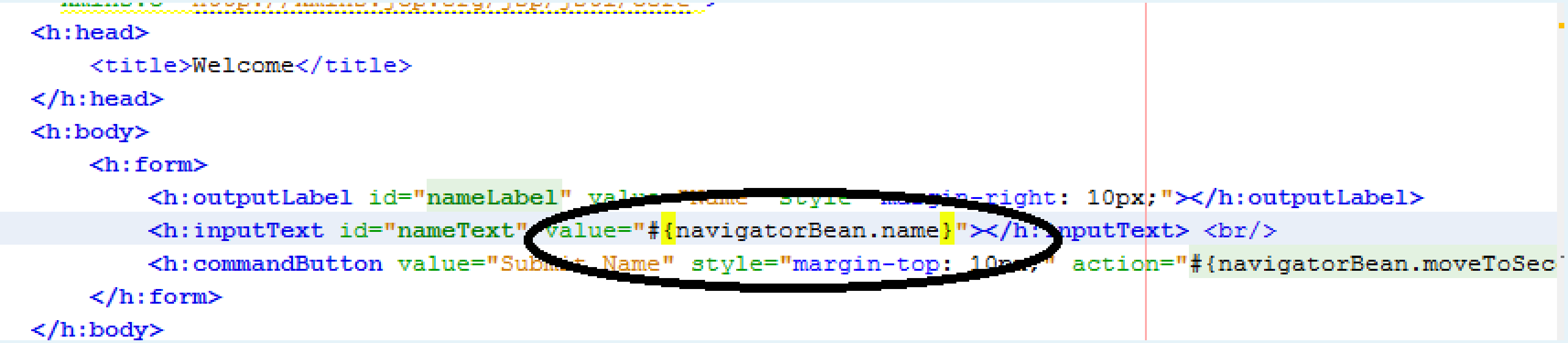
Passing Data



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

@ManagedBean(name = "navigatorBean", eager = true)
@RequestScoped
public class NavigatorBean {

    private String name;

    public String moveToSecondPage(){
        return "second";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public NavigatorBean() {
    }
}
```

Passing Data

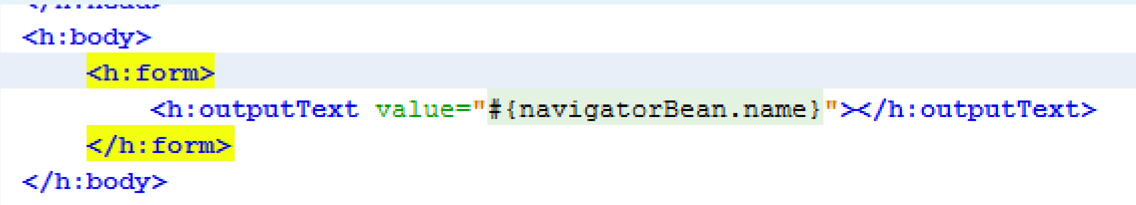
- In index.xhtml define the value for your inputText element. This will store what ever is typed to the field in our managed bean when form is submitted (the button is pressed).



```
<h:head>
    <title>Welcome</title>
</h:head>
<h:body>
    <h:form>
        <h:outputLabel id="nameLabel" value="Name" style="margin-right: 10px;"></h:outputLabel>
        <h:inputText id="nameText" value="#{navigatorBean.name}"></h:inputText> <br/>
        <h:commandButton value="Submit Name" style="margin-top: 10px;" action="#{navigatorBean.moveToSec
    </h:form>
</h:body>
```

Passing Data

- In second.xhtml define an outputText field. From value field define the name attribute from our manage bean class. The ouputText field will call automatically the get function from class to retrieve the name attribute value.



```
</h:head>
<h:body>
    <h:form>
        <h:outputText value="#{navigatorBean.name}"></h:outputText>
    </h:form>
</h:body>
```