BBM490

ENTERPRISE WEB ARCHITECTURE

Mert ÇALIŞKAN

Hacettepe University Spring Semester ’15

Week 8

Java Server Faces (JSF) for UI Development

We'll cover Java Server Faces, which is the component-based UI for web applications. JSF is also a part of Java Enterprise Edition.

What's JSF?

- It's component oriented User Interface Framework
- It's based on MVC pattern.
- It's part of the Java EE Platform.

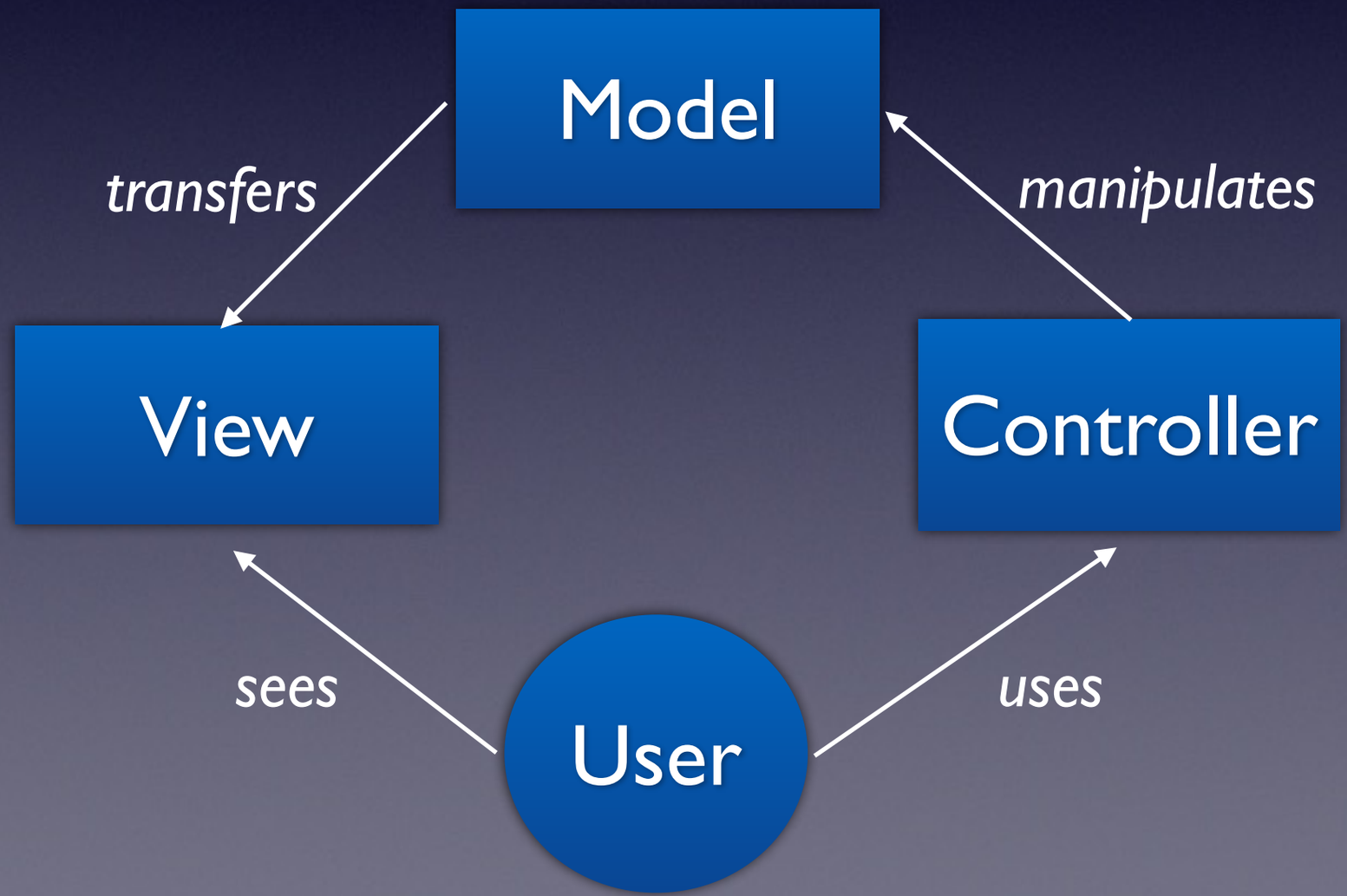
- JSF 1.0 released in March 2004.
- We now have JSF 2.2 with Java EE version 7. **(latest)**
- It provides AJAX infrastructure out of the box.
- There are 2 implementations of JSF, from Oracle and from Apache MyFaces.

Benefits of JSF

- JSF reduces the effort in creating and maintaining applications which will run on a Java application server 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 Architecture

- Model: Managed Beans
- View: JSF Pages
- Controller: Faces Servlet



FacesServlet

- FacesServlet is a servlet that manages the request processing lifecycle for web applications that are utilising JSF to construct the user interface.
- Handles all JSF requests, it acts like a gateway.
- It's configured in web.xml like where we configure all of our Servlets.
- It uses faces-config.xml for its configuration.
- FacesServlet is a class defined as *final*.

FacesServlet

- Faces Servlet follows the design pattern called: Front Controller

- This pattern provides a centralised entry point for handling all requests coming from the user.

- Spring MVC provides its own controller with Dispatcher Servlet, which dispatches the requests to the regarding @Controller classes.

Managed Beans

- It's a regular Java Bean registered with JSF container.
- Works as *model* for User Interface.
- Managed beans are accessed from JSF pages with Expression Language notation.
- 'Till JSF 1.2 they need to be registered in faces-config.xml (we will also cover the configuration xml in a bit) but with JSF 2.0 they can be defined with annotations also.

Configuring JSF App

- It's the application configuration resource file designed as an XML document.
- A faces-config tag encloses all other declarations.

```
<faces-config version="2.2"
              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.jcp.org/xml/ns/javaee/web-
facesconfig_2_2.xsd”>
</faces-config>
```

- There can be more than one configuration file. The content of the files will be merge at app. startup.

Configuring JSF App

- Configuration files could either reside under:
  - /META-INF/faces-config.xml in any of the JAR files in the web application's /WEB-INF/lib/ directory.
  - A context initialization parameter, javax.faces.application.CONFIG_FILES, that specifies one or more (comma-delimited) paths to multiple configuration files for your web application.
  - A resource named faces-config.xml in the /WEB-INF/ directory of your application.

Defining a Managed Bean

- It's easy to define the bean via faces-config.xml by using the <managed-bean> tag.

```
<managed-bean>
    <managed-bean-name>helloBean</managed-bean-name>
    <managed-bean-class>tr.edu.hacettepe.bbm490.HelloView</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

- When a page references a bean, the JavaServer Faces implementation initialises it according to its configuration in the application configuration resource file.

Defining a Managed Bean

- managed-bean-name element defines the key under which the bean will be stored in a scope.
- managed-bean-class element defines the fully qualified name of the JavaBeans component class used to instantiate the bean.
- managed-bean-scope element defines the scope in which the bean will be stored. The available values are *application*, *session*, *view*, *request* and *none*.

Dependencies

- The dependencies for JSF implementation can be defined as a bundled artefact,

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.faces</artifactId>
    <version>2.2.9</version>
</dependency>
```

- or as a 2 JAR variant,

```
<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.9</version>
</dependency>
<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.9</version>
</dependency>
```

API & IMPL

- As seen in previous slide, JSF comes with two dependencies,
  - jsf-api
  - jsf-impl
- API consists of base classes and interfaces which application programmers can write code by using it.
- The IMPL jar consists of the default JSF component implementations.
- API contains the UIComponent abstract class for instance while the IMPL contains the implementation of OutputText component.

Facelets Page

- Facelets is an open source Web Template System that is being used by JSF as the default view handler technology (with JSF 2.0).
- JSF pages can easily be created with Facelets. JSP is also supported but with previous versions of JSF, it was a problem to use JSP tags with JSF.
- Facelets pages are suffixed with **.xhtml**. It states for well-balanced XML based HTML. It has namespaces as in XML files and libraries can be used through these namespaces.

Sample Facelets

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
   <title>JSF Examples</title>
</h:head>
<h:body>
   #{helloBean.message}
</h:body>
</html>
```

- JSF introduces two namespaces for tags, **h:** and **f:**

src:jsf-simple-xml

Getters / Setters

- JSF parses the EL and generate ValueExpression object from the deferred expression.
- The expression is not immediately evaluated, but instead ValueExpression is handled and the getter behind it gets invoked.
- The getter method is usually invoked 1-2 times per JSF request-response cycle whether the component is an input/output component.
- Count get much higher if the expression resides inside a datatable or ui:repeat component for instance.

Annotation Based Configuration

- We can also configure our JSF Beans via Java Annotations.
- With the presence of @ManagedBean annotation a Java class gets registered as a managed bean. The name of the bean can be defined with name attribute of the annotation. If not exists, the class name is used with the first letter lower-cased.
- The scope of the managed bean gets managed by the scope annotations respectively.

Scope Annotations

- The annotations below applies the scope rules that we defined in previous slides.
- @NoneScoped
- @RequestScoped
- @ViewScoped
- @SessionScoped
- @ApplicationScoped

src:jsf-simple-annotations

Sample Facelets

- h is the standard HTML tag library for JSF. These tags get rendered to corresponding HTML output. This tag library contains JavaServer Faces component tags for all UIComponent + HTML RenderKit Renderer combinations defined in the JavaServer Faces Specification

```
<h:outputText value="#{helloBean.message}" />
```

- Here, for outputText component, the output will be only the text.

Sample Facelets

- f is the core JavaServer Faces custom actions that are independent of any particular RenderKit.

```
<h:outputText value="#{helloBean.currentDate}">
   <f:convertDateTime pattern="dd.MM.yyyy" />
</h:outputText>
```

- Here we are rendering a date field on the page and to format the date we have used an f tag, which does know nothing about the HTML rendering and such stuff. It just formatted the data.

Scopes of the bean

- application: It's the scope that is shared across all users' interactions in a web application. It's the scope that has thee longest lifetime.

- session: It's the scope that lives as long as the user's HTTP session lives. So it lives across the multiple HTTP request-response cycles.

- view: It's the scope that lives over multiple requests without leaving the current JSF view. It lives as long as we do post-back (we'll define post-back in a bit) on the same view.

Initial Request vs. Postback Request

- JSF recognises two methods for a request: initial and post-back.

- Initial request is the first request that browser does for loading the page - you can obtain such a request by accessing the application URL in browser or following a link.

- This kind of request is processed in Restore View Phase and Render Response Phase.

Initial Request vs. Postback Request

- Postback request happens when we click a button for submitting a form. Unlike the initial request, the postback request passes through all phases.

- JSF provides a method named **`isPostback`** that returns a boolean value - returns true for postback request and false for initial request.

```
FacesContext.getCurrentInstance().isPostback();
```

Initial Request vs. Postback Request

- It can be useful to know when the request is initial or postback. For example, you may want to accomplish a task a single time, at initial request (e.g. initialisation tasks), or every time, except first time (e.g. display a message, which is not proper to appear when a page is displayed as a result of the initial request).

Scopes of the bean cont'd

- request: It's the scope that lives as long as the HTTP request-response lives. request scope is useful for simple GET requests that expose some data to the user without requiring storing the data.

- none: Lives as long as an EL expression evaluation. It's the scope that has the shortest lifetime.

The need of Bean Serialisation

- Starting with view scope, the managed beans will be stored in HttpSession so they need to implement **`java.io.Serializable`**.

- This is needed because most of the servers may need to persist session data to hard-disk to be able to survive heavy load and/or reviving sessions during server restart.

- ObjectOutputStream will be used to write them to hard-disk and ObjectInputStream to read them from hard-disk.

Main Elements of JSF

- UI Components
- Renderers
- Validators
- Backing Beans
- Converters
- Messages
- Navigation

Components

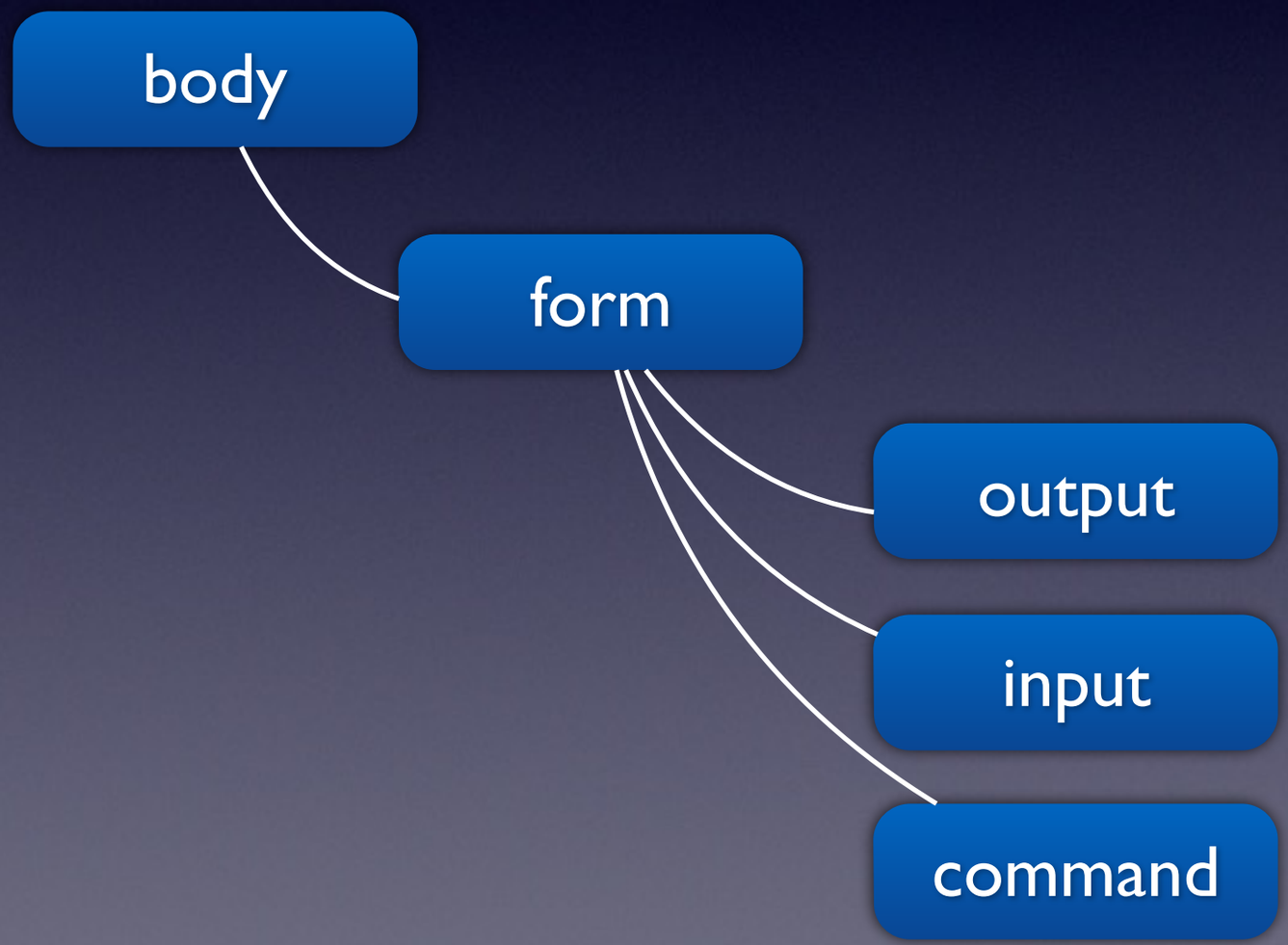
- JSF defines a standard set of components with the -impl.
- The View Layer (Facelets) uses these components to interact with the user and pass data to the managed beans.

- A JSF page gets represented as a tree of components like the sample below.

```
<h:body>
    <h:form id=“form1”>
        <h:outputText id="lbl_name" value="Name" />
        <h:inputText id="txt_name" value="#{myBean.userName}" />
        <h:commandButton id="btn_save" value="Save User" action="#{myBean.save}" />
    </h:form>
</h:body>
```

Component Tree

- Components are used to separate business logic from presentation, they are represented as a tree.



output components

- components extend the class: javax.faces.component.UIOutput
- OutputText, OutputLink or OutputFormat are output components that can be used from the -impl.
- they all have id, rendered and value attributes coming from the UIComponentBase.
- rendered attribute provides the ability to set the visibility of the component on the page (true|false).

output components

```
<h:outputText id="lbl_out" value="#{myOutputBean.value}" />

<h:outputLink id="lnk_out" value="http://www.google.com">
   <h:outputText id="lbl_lnk_out" value="Go for Google" />
</h:outputLink>

<h:outputFormat id="out_format"
   value="parameter 1 : {0}, parameter 2 : {1}">
   <f:param value="Hacettepe University" />
   <f:param value="BBM490" />
</h:outputFormat>
```

src:jsf-components
link: http://localhost:8080/output.jsf

input components

- components extend the class: javax.faces.component.UIInput
- It's time to retrieve data from the user. JSF provides input components that provide features of the HTML input components like,
    - text input
    - text area input
    - checkbox
    - radiobutton
    - combo box
    - file upload and etc.

input components

- input components need to reside inside a form. JSF also provides a form component: <h:form>

```
<h:form>
   <h:inputText id="txt_user" value="#{myInputBean.username}" />
   <h:commandButton id="btn_save" value="Submit" />
   <h:outputText id="out_user" value="#{myInputBean.username}" />
</h:form>
```

- Here we have a managedBean MyInputBean class with a property username. The getter/setter of the property needed to retrieve data from the page and to provide the data to the page.
- UIInput also extends the UIOutput.

src:jsf-components
link: http://localhost:8080/input1.jsf

input components

- The JSF lifecycle gets executed to retrieve data from the HTML source and set it to the property of the managed bean (We will get to it soon…)
- The HTML output of the previous sample will be:

```
<form id="j_idt2" name="j_idt2" method="post" action="/BasicJSFWithUIComponents/
input1.jsf" enctype="application/x-www-form-urlencoded">
    <input type="hidden" name="j_idt2" value="j_idt2" />
    <input id="j_idt2:txt_user" type="text" name="j_idt2:txt_user" />
    <input id="j_idt2:btn_save" type="submit" name="j_idt2:btn_save"
value="Submit" />
    <span id="j_idt2:out_user"></span>
</form>
```

id's of components

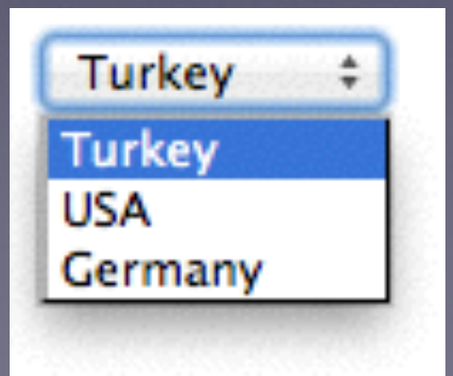
- As you see in the output, if you don't specify any id's for some of the components, you will get generated ones, which start like: j_id.
- To overcome this, it'd be better to define id's for all of the components. That'll offer us:
  - Keep the HTML output free of auto-generated JSF component IDs.
  - Handle problems that might occur with the client side unit testers, such as selenium.
  - Make the developer aware which components are naming containers and/or implicitly require outputting its ID.

input components

- Other samples for the input components could be like:
- combo box selection

```
<h:form>
   <h:selectOneMenu id="countries" value="#{myInputBean2.selectedCountry}">
      <f:selectItems value="#{myInputBean2.countries}" var="c" itemValue="#{c}"
itemLabel="#{c}" />
   </h:selectOneMenu>
   <h:commandButton id="btn_save" value="Submit" />
   <h:outputText id="out_country" value="#{myInputBean2.selectedCountry}" />
</h:form>
```

- The output would be like:



src:jsf-components
link: http://localhost:8080/input2.jsf

input components

- Checkbox selection

```
<h:form>
    <h:selectManyCheckbox id="countries" value="#{myInputBean3.selectedCountries}">
        <f:selectItems value="#{myInputBean3.countries}" var="c" itemValue="#{c}"
itemLabel="#{c}" />
    </h:selectManyCheckbox>
    <h:commandButton id="btn_save" value="Submit" />
    <h:outputText id="out_countries" value="#{myInputBean3.selectedCountries}" />
</h:form>
```

- output would be like: 

- You see that how the component definition resembles with the combo box selection? This is what JSF provides, it's very easy to create component oriented pages.

src:jsf-components

link: http://localhost:8080/input3.jsf

input components

- the list of input components that are provided by the implementation are:

- h:inputSecret
- h:inputText
- h:inputTextarea
- h:inputFile
- h:selectBooleanCheckbox
- h:selectManyCheckbox
- h:selectManyListbox
- h:selectManyMenu
- h:selectOneListbox
- h:selectOneMenu
- h:selectOneRadio

data components

- components extend the class: javax.faces.component.UIData
- DataTable iterate over an element of collection and display data on a tabular fashion.
- Component renders HTML Table with <th>, <tr> and <td>.
- Data Table component does not provide any filtering, sorting and any other enhanced features. (We will use PrimeFaces for this…soon…)

data components

- Sample for a dataTable to display a list of Country objects could be:

```
<h:dataTable value="#{myDataBean1.countries}" var="c" border="1">
    <h:column>
        <f:facet name="header"><h:outputText value="ID" /></f:facet>
        <h:outputText value="#{c.id}" />
    </h:column>
    <h:column>
        <f:facet name="header"><h:outputText value="Name" /></f:facet>
        <h:outputText value="#{c.name}" />
    </h:column>
    <h:column>
        <f:facet name="header"><h:outputText value="Population" /></f:facet>
        <h:outputText value="#{c.population}" />
    </h:column>
</h:dataTable>
```

- Output would be like:

ID	Name	Population
1	Turkey	75000000
2	USA	350000000
3	Germany	100000000

src:jsf-components

link: http://localhost:8080/data1.jsf

action components

- components extend javax.faces.component.UICommand
- We have input components to retrieve data from user and output and data components to view the data to the user. But we need to execute some actions on the data that we retrieved, like added a Country to the list we have.
- CommandLink and CommandButton components can be used for handling this kind of operations.

action components

- Let's create a page with dataTable where we can add new countries with the inputText components.
- The commandButton and its action would be like:

```
<h:commandButton id="btn_save" value="Add"
action="#{myActionBean1.addCountry}" />
```

- And the method for the action would be like:

```
public String addCountry() {
   countries.add(newCountry);
   newCountry = new Country();

   return null;
}
```

src:jsf-components

link: http://localhost:8080/action1.jsf

action components

- action will also be valid with any MethodExpression.

```
<h:commandLink value="submit" action="#{bean.edit(item)}" />

public void edit(Item item) {
// ...
}
```

- So you can pass objects from EL to the server side.

FacesContext

- FacesContext refers to a single JavaServer Faces request, and corresponding response.
- The instance of the context gets modified with each phase of request processing lifecycle (we will also cover the lifecycle).
- We can access the facesContext instance of current processing request via,

```
FacesContext.getCurrentInstance();
```

- We can add messages, check for validation status of the page via facesContext.

Adding a Message

- JSF provides 2 components to output a message to the user:
  - h:message – Output a single message for a specific component.
  - h:messages – Output all messages in current page.
- Adding the message is easy with FacesContext instance.

```
    public String addCountry() {
        countries.add(newCountry);
        newCountry = new Country();
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Country added
successfully"));
        return null;
    }
```

Adding a Message

- We are using addMessage which gets the id of the JSF component and an instance of FacesMessage. If the id is not provided the message will be treated as a message to the whole page. <h:messages> component will take of this message and render it to the user.
- We are creating a new FacesMessage instance with the message string. We can also provide a summary of and the detail of the message along with the severity.
- Severity values could either be: INFO, WARN, ERROR and FATAL.

JSF LifeCycle

- JSF consists of 6 phases of lifecycle that gets executed in order.
    - Restore view
    - Apply request values
    - Process validations
    - Update model values
    - Invoke application
    - Render response

JSF LifeCycle

- With the initial request on a JSF page,
  - Restore view
  - Render response
- phases will get executed. Since there is no component within the view tree that'd be stored in the session, the view will not be restored.
- The Render response phase will render the page as HTML source to be viewed to the user.

JSF LifeCycle

- With the form submit on a JSF page,
    - Restore view
        - The component tree gets restored from the session.
- Apply request values
    - The submitted form values are set to each component.

JSF LifeCycle

- With the form submit on a JSF page,
  - Process validations
    - If submitted values are converted and then gets validated. If any conversion/validation error occurs than phase steps to render response bypassing the rest.
  - Update model values
    - The model gets updated with the submitted values.

JSF LifeCycle

- With the form submit on a JSF page,
  - Invoke application
    - Actual code call happens here: like an action method from a command button.
  - Render response
    - The Render response phase will render the page as HTML source to be viewed to the user.

Renderers

- With the Renderer, JSF component gets encoded its generated HTML for the UI. decoding is getting submitted data from the user.
- Renderers are grouped by the RenderKits.
- Default Renderkit is HTML for the components.
- It provides the device independence w/o changing templating language or components themselves.
- A renderkit for mobile devices is also possible.

Converters

- We are retrieving the data from the client but it gets converted and validated automatically (LifeCycle of JSF).
- Think of a java.util.Date field in a User class, like birthdate.
- User will see this field on the page as dd/MM/yyyy which is a String but when he sends the data to the server it will be converted to java.util.Date automatically and then validated against rules and conditions that exists.

Converters

- JSF provides built-in converters for numbers and dates.
- The sample below formats 22333 as 22,333.0 after a post-back.

```
<h:inputText id="txt_value" value="#{myConverterBean.longValue}">
  <f:convertNumber pattern="#,##0.0#;(#)" />
</h:inputText>
```

- convertNumber is used with a specified pattern here.

src:jsf-converter
link: http://localhost:8080/converter1.jsf

Converters

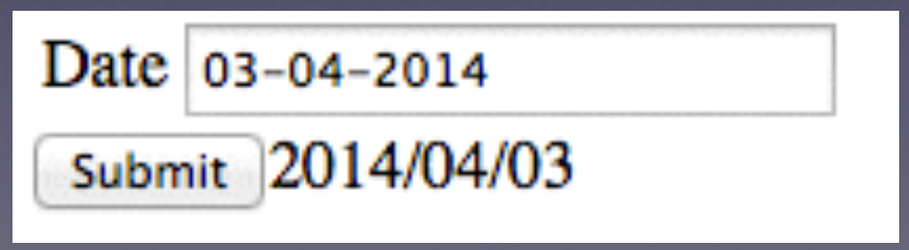
- Date conversion is also possible with f:convertDateTime and its *pattern* attribute.
- We can input date like,

```
<h:inputText id="txt_date" value="#{myConverterBean2.date}">
   <f:convertDateTime pattern="dd-MM-yyyy" />
</h:inputText>
```

- and then get the output formatted like,

```
<h:outputText value="#{myConverterBean2.date}">
   <f:convertDateTime pattern="yyyy/MM/dd" />
</h:outputText>
```

- The result will be as:



src:jsf-converter

link: http://localhost:8080/converter2.jsf

Validators

- JSF provides built-in validators to validate the data input via UI components.
- To validate a length of the String,

```
<h:inputText id="txt_value" value="#{myValidatorBean1.value}">
   <f:validateLength minimum="5" maximum="8" />
</h:inputText>
```

- To validate a range of a numeric value,

```
<h:inputText id="txt_value" value="#{myValidatorBean2.value}">
   <f:validateLongRange minimum="5" maximum="100" />
</h:inputText>
```

src:jsf-validator
link: http://localhost:8080/validator1.jsf
link: http://localhost:8080/validator2.jsf

Validators

- There is also a regex validator that can be used as,

```
<h:inputSecret id="passwordInput" value="#{userData.password}">
    <f:validateRegex pattern="((?=.*[a-z]).{6,})" />
</h:inputSecret>
```

- We can also create a custom validator like an email validator for instance like,

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

   public void validate(FacesContext context, UIComponent component,
         Object value) throws ValidatorException {

         // do some controls and if doesn't fit
         // throw Validator Ex. with a Faces message
         throw new ValidatorException(new FacesMessage("error!"));
      }
   }
}
```

Validators

- And the email validator can be used as,

```
<h:inputText id="email" value="#{user.email}">
   <f:validator validatorId="emailValidator" />
</h:inputText>
```

Navigation

- ‘Till now we only used one JSF page did our stuff inside it.
- Well it’s not feasible for an enterprise application.
- User will click a button/link and will eventually will want to navigate.
- We need a way to navigate between JSF pages in order to interact with the user.
- JSF provides a multiple ways to handle navigation for the user interactions.

Navigation

- Navigation rules are those rules provided by JSF Framework which describe which view is to be shown when a button or link is clicked.
- Navigation rules can either be defined in,
  - via faces-config.xml
  - via within action method return values
- Navigation rules can contain conditions based on which resulted view can be shown.

Navigation

- With the rule below we can navigate between first.xhtml and second.xhtml easily like,

```
<navigation-rule>
   <from-view-id>first.xhtml</from-view-id>
   <navigation-case>
      <from-outcome>go</from-outcome>
      <to-view-id>second.jsf</to-view-id>
   </navigation-case>
</navigation-rule>
```

- Any action that will be invoked on the first page gets a result output as “go”, The first.xhtml will get forwarded to second.xhtml

Navigation

- We can also do conditional navigation case here like,

```
<navigation-rule>
   <from-view-id>second.xhtml</from-view-id>
   <navigation-case>
      <from-action>#{secondPage.navigate}</from-action>
      <from-outcome>false</from-outcome>
      <to-view-id>third.jsf</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-action>#{secondPage.navigate}</from-action>
      <from-outcome>true</from-outcome>
      <to-view-id>fourth.jsf</to-view-id>
   </navigation-case>
</navigation-rule>
```

- Where we're doing a navigation according to the outcome.

Navigation

- There is also the implicit navigation where we don't need to invoke any method on the server side but only we can navigate between pages by stating their names.

```
<h:commandButton id="goto" value="Goto Next Page" action="fourth" />
```

- Here we're only stating the name of the page as "fourth" and JSF will look for fourth.xhtml within the same path of the requester page.

src:jsf-navigation
link: localhost:8080/first.jsf

Passing Data Between JSF Pages and Managed Beans

- There are 4 ways to pass a parameter value from JSF page to backing bean:
  - Method Expression (w/ JSF 2.0)
  - f:param
  - f:attribute
  - f:setPropertyActionListener

w/ Method Expressions

- While invoking a method from an EL, we can pass parameter like,

```
<h:commandButton action="#{user.editAction(id)}" />
```

- And the backing bean method would be like:

```
public String editAction(String id) {
     //id = "delete"
   }
```

- This is possible if your servlet container supports Servlet 3.0 / EL 2.2.

w/ <f:param>

- We can pass parameter with <f:param> tag and get it back through request parameter.
- The page would be like:

```
<h:commandButton action="#{user.editAction}">
   <f:param name="action" value="delete" />
</h:commandButton>
```

- And the method would be like:

```
public String editAction() {

    Map<String,String> params =
          FacesContext.getExternalContext().getRequestParameterMap();
    String action = params.get("action");
         //...

  }
```

- If your bean is request-scoped, JSF can set it w/ @ManagedProperty(value="#{param.action}")

w/ <f:attribute>

- We can pass parameter with <f:attribute> tag and get it back through action listener.
- The page would be like:

```
<h:commandButton action="#{user.editAction}"
                 actionListener="#{user.attrListener}">
   <f:attribute name="action" value="delete" />
</h:commandButton>
```

- And the method definitions would be like:

```
public void attrListener(ActionEvent event){

   String action =
       (String)event.getComponent().getAttributes().get("action");

  }

  public String editAction() {
   //...
  }
```

w/ <f:setPropertyActionListener>

- We can pass parameter with <f:setPropertyActionListener> tag by setting directly into the property of our backing bean.
- The page would be like:

```
<h:commandButton action="#{user.editAction}" >
    <f:setPropertyActionListener target="#{user.action}" value="delete" />
</h:commandButton>
```

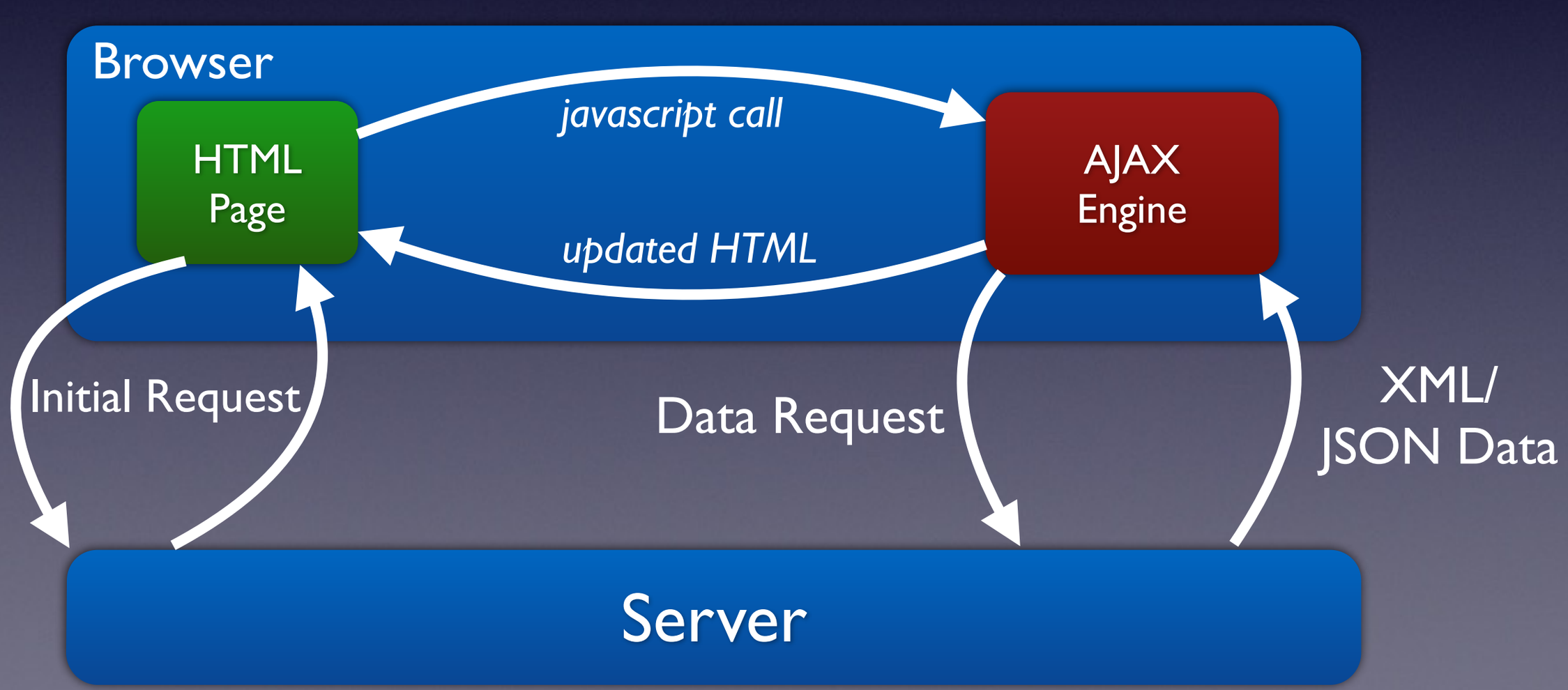
- And our backing would be like:

```
public String action;

public void setAction(String action) {
   this.action = action;
}

public String editAction() {
}
```

src:jsf-passing-data

Remember AJAX?

- Client side make server side calls by partially submitting the page (specific form elements) and gets partial data from server to update the page partially.



JSF and AJAX

- There are 2 main concepts for AJAX with JSF
  - partial processing
    - Indicates the components that should be processed on the server.
  - partial rendering
    - Indicates the components that should be rendered (re-rendered) on the client side.

Benefits of AJAX

- On Client Side
  - We can update JSF elements (h:outputText, h:inputText, h:selectOneMenu, etc.)
    - We don’t have to write JavaScript
- On Server Side
  - AJAX calls know about JSF managed beans
  - We don’t have to write servlets

Usage of f:ajax

```
<h:form>
   <h:commandButton action="#{ajaxBean1.increment}"
                                                value="Increment">
      <f:ajax />
   </h:commandButton>
   <h:outputText id="value" value="#{ajaxBean1.counter}" />
</h:form>
```

<f:ajax>

- JSF provides the f tag for enabling AJAX for JSF components.
- <f:ajax> tag introduces 2 attributes for handling partial processing and partial rendering.
    - execute / partial processing
    - render / partial rendering
- If f:ajax exists, it's an AJAX request. that's it..!
- There is no need for implementing javascript stuff.

<f:ajax> defaults

- We haven't specified what to execute and what to render in the sample below. But JSF will not throw any error for this.

```
<h:form>
   <h:commandButton action="#{ajaxBean1.increment}" value="Increment">
      <f:ajax />
   </h:commandButton>
   <h:outputText id="value" value="#{ajaxBean1.counter}" />
</h:form>
```

- By default JSF executes the element that triggered the request and re-render the same element, which is <h:commandButton>

<f:ajax>

- There are 4 other attributes of the f:ajax tag.
- event: It's the DOM event to respond to (like keyup, blur and etc.)
- listener: To Execute Java code on the backing bean.
- onevent: It's a JavaScript function to run when an event is fired.
- onerror: It's a JavaScript function to run when an error is occurred.

<f:ajax>

- Functional working example of the previous code would be:

```
<h:form>
   <h:commandButton action="#{ajaxBean1.increment}" value="Increment">
      <f:ajax render="value" />
   </h:commandButton>
   <h:outputText id="value" value="#{ajaxBean1.counter}" />
</h:form>
```

- Here we have render attribute on the f:ajax stating that after processing itself (commandButton) the outputText should be updated.
- We haven't specified execute here but we could.

src:jsf-ajax
link: http://localhost:8080/ajax1.jsf

execute & render

- We have stated the usage of execute and render with id's of the components. If we need to specify multiple id's we can give them with space delimited.
- There are also reserved words starting with @ instead of the id's.
  - @form - Refers to all component ids in the form that contains the AJAX component.
  - @this - Refers to the id of the element that triggers the request.

execute & render

- @none - no component will be processed/re-rendered.
- @all - all components in page are submitted and processed - is like a full page submit.
- PrimeFaces extends this list with new words like @parent, @child, @next, @previous and etc.
- Also PrimeFaces provides a p:ajax component which is an enhanced version of the f:ajax.
- We'll get to these later while discussing PrimeFaces.

event="..."

- Each AJAX request is fired by an event indicating a programmatic action.
- JSF defines default events for components; 'action' for commandButton, 'valueChange' for inputText components for instance.
- event attribute can be used to set specific event for a component.
- Values could be: click, focus, keyup or mouseover.

event="keyup"

- Here actions gets triggered on keyup event of the input field. So each time we input data into the inputText component the outputText component will get updated.

```
<h:form>
    <h:inputText value="#{ajaxBean2.text}">
        <f:ajax event="keyup" render="text" />
    </h:inputText>
    <br/>
    <h:outputText id="text" value="#{ajaxBean2.text}" />
</h:form>
```

- We also don't need to specify the process attribute here.

src:jsf-ajax-event
link: http://localhost:8080/ajax2.jsf

encapsulation with <f:ajax>

- It's also possible to encapsulate input components with f:ajax to state that the fields will be processed via an AJAX event and with the result of the process component(s) will be updated via render attribute.

```
<f:ajax event="keyup" render="result">
   <h:inputText value="#{ajaxBean3.name}" /><br/>
   <h:inputText value="#{ajaxBean3.lastname}" /><br/>
   <h:inputText value="#{ajaxBean3.age}" /><br/>
</f:ajax>
```

So a Question…

- We called it AJAX, Async JavaScript and… wait it contains JavaScript but we haven't included any JavaScript files into our JSF page.
- JSF auto includes the jsf.js JavaScript file whenever we use the <f:ajax> tag.
- Via this script file the AJAX call can be done by JSF.

Spring with JSF

- We used JSF beans with no problem. But it's just another bean container. We were using Spring since the start of the workshop and enjoying its features. so how about using Spring beans inside JSF pages with the EL?
- It's possible with the help of Spring.
- We'll modify web.xml, appContext.xml and faces-config.xml along with the pom.xml so that we'll be using our Spring beans in with Facelet Pages.
- No JSF bean will be used, that is one option to leverage Spring instead of JSF in container-wise.

src:jsf-with-spring

JSF Templating

- In web applications, most pages are follow a similar web interface layout and styling, for example, same header and footer.
- With Facelets tags it's easy to provide a standard web interface layout.
- Facelets provide 4 tags with the ui: namespace.
- We'll create a template page, a header and footer snippets and 2 different pages that use the template we created.

JSF Templating

- ui:insert: Used in template file, it defines the part that is going to be replaced by the file that loads the template. The content gets replaced by the content defined inside the "ui:define" tag.
- ui:define: Defines the content that will be inserted into template with a matching "ui:insert" tag.

  ui:insert and ui:define parts match with the *name* attribute.

JSF Templating

- ui:include: includes content from another XHTML page.
- ui:composition: – If used with "template" attribute, the specified template is loaded, and the children of this tag, with ui:define, defines the content.
- Let's give an example of a template that consists of 3 sections; header, content and footer.
- header and footer inserted from other xhtml files and we demonstrated the templating by defining 3 different pages.

JSF Templating

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

<h:body>
    <div id="page">
        <div id="header">
            <ui:insert name="header">
                <ui:include src="header.xhtml" />
            </ui:insert>
        </div>

        <ui:insert name="content"/>

        <div id="footer">
            <ui:insert name="footer">
                <ui:include src="footer.xhtml" />
            </ui:insert>
        </div>
    </div>
</h:body>
</html>
```

JSF Templating

- header.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:ui="http://java.sun.com/jsf/facelets">
<body>
   <ui:composition>
      <h1>This is the header</h1>
   </ui:composition>
</body>
</html>
```

- footer.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:ui="http://java.sun.com/jsf/facelets">
<body>
   <ui:composition>
      <h1>This is the footer</h1>
   </ui:composition>
</body>
</html>
```

JSF Templating

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   template="template.xhtml">
   <ui:define name="content">
      <h:form>
         This is First Page
         <br/>
         <h:commandButton id="btnGo" action="#{page1.navigate}" value="Go to
Next Page" />
      </h:form>
   </ui:define>
</ui:composition>
```

src:jsf-templating