Java Technologies
Java Server Faces (JSF)

The Context

“As far as the customer is concerned, the interface **is** the product”, Jef Raskin

- A Web user interface allows the user to interact with content or software running on a remote server through a Web browser.
- *Web Components* (Servlets, JSP, Beans, etc.)

  = “The Bricks” needed for creating Web Apps
- **Frameworks** are sets of design patterns, APIs, and runtime implementations intended to simplify the design and coding process for building new applications.

  → Enable application developers to <u>concentrate on the unique feature set required by their applications</u>.

User Interface Design

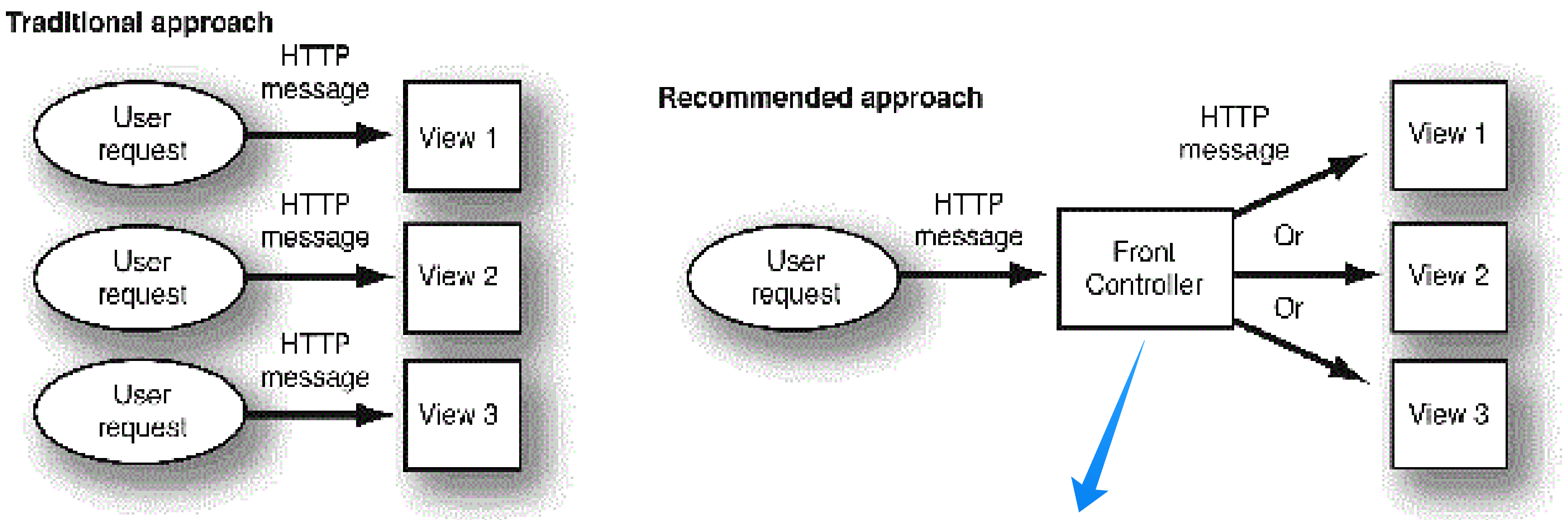
- **Components**
  - Input Controls
  - Navigational Components
  - Informational Components
  - Containers
- **Best practices** – UI should be:
  - **Simple**: “open, read, write, close”
  - **Consistent**: “same layout, but in different colors”
  - **Informative**: “clear messages – less complaints”
  - **Intelligent**: “defaults and workflows”
  - **Responsive**: “0.1s – 1s - 10s”
  - Beautiful : “not really”

Web Application Frameworks

provide support for:

- **Navigating** among the application's pages
- Creating **the presentation** (forms, views, etc.)
- **Accessing data** (beans, etc.) and **server resources** (JDBC connections, etc.)
- Data **validation** and **conversion** (web ↔ model)
- **Internationalization**
- **Secure access** to resources
- **Role separation**, ...

Front Controller



```
<!-- Request Mapping -->
<servlet-mapping>
  <url-pattern>*.do</url-pattern>
  <servlet-name>
    com.some.mvc.framework.FrontController
  </servlet-name>
</servlet-mapping>
```

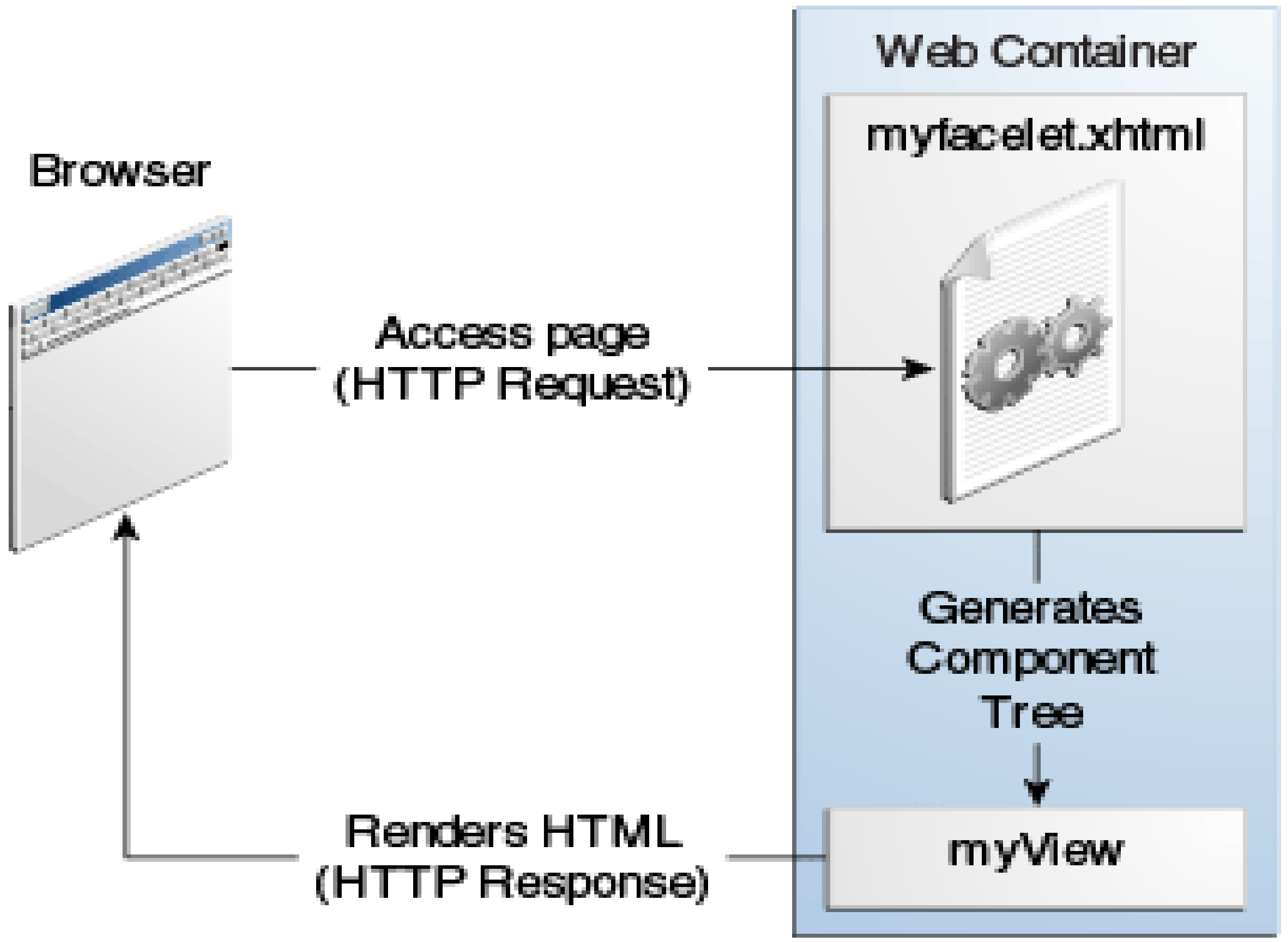
Provides a centralized entry point for handling requests

Java Server Faces

Server-side **component framework** for building web applications:

- Offers an **API for representing UI components** and managing their **server-side** state
- **Tag libraries** for using components in pages and for connecting them to server-side objects
- **Event based** interaction model
- **Support** for server-side validation, data conversion; page navigation; internationalization
- **Simplifies** the process of building and maintaining web applications with server-side user interfaces

Server-Side UI



JSF Implementations

“JSF is not what you've been told anymore”, Çağatay Çivici

Specifications: 1.0 (2004) → 1.2 (2006) → 2.0 (2009) → 2.1 (2010) → 2.2 (2013) → 2.3 (2017) → 3.0RC1 (2020)

Implementations

- Reference (included in GlassFish)
- **Open-Source, Ajax-enabled:**
  - PrimeFaces
  - ICEFaces
  - JBoss RichFaces
  - Apache MyFaces, ...

Rich Internet Applications

Creating a Simple JSF Application

- Create the backing bean(s)
  - ➔`PersonBean`
- Write the pages
  - ➔`inputName.xhtml, sayHello.xhtml`
- Create the resource bundle
  - ➔`Messages.properties`
- Define navigation rules
  - ➔`faces-config.xml`

PersonBean

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

@ManagedBean(name = "personBean")
@RequestScoped
public class PersonBean {
  private String name;

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

the name of the bean

the visibility domain

```
//CDI
@Named("personBean")
```
→ javax.inject.Named
```
@RequestScoped
```
→ javax.enterprise.context

inputName.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Hello</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel for="userName" value="Your name:"/>
      <h:inputText id="userName"
                   value="#{personBean.name}"
                   required="true"/>
      <h:commandButton id="submit"
                       value="Submit"
                       action="sayHello.xhtml"/>
      <h:message id="errors" style="color: red"
                 showSummary="true" showDetail="false"
                 for="userName"/>
    </h:form>
  </h:body>
</html>
```

Messages.properties

The Resource Bundle

contains messages displayed by the application

```
# key = value
title = Simple JSF Application
greeting = Hello
```

sayHello.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <f:loadBundle basename="hello.Messages" var="message"/>
  <h:head>
    <title>#{message.title}</title>
  </h:head>
  <h:body>
    <h:form>
      <h2>
        #{message.greeting} #{personBean.name} !
      </h2>
      <h:commandButton id="back" value="Back"
                             action="inputName.xhtml"/>
    </h:form>
  </h:body>
</html>
```

Page Navigation

- Replace actual file names with abstract actions:

```
<h:commandButton id="submit" value="Submit" action="sayHello.xhtml"/>
```

"sayHello.xhtml" → "hello"

- Define the navigation rules in **faces-config.xml**

```
<faces-config>
  <navigation-rule>
    <from-view-id>/inputName.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>hello</from-outcome>
      <to-view-id>/sayHello.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Configure JSF Environment in web.xml

```
<web-app>
  <context-param>
    <param-name>javax.faces.application.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
  </context-param>

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

  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Configure JSF Environment in faces-config.xml

```
<faces-config version="2.2" … >
    <application>
        <message-bundle>
            Messages
        </message-bundle>
        <resource-bundle>
            <base-name>Messages</base-name>
            <var>msg</var>
        </resource-bundle>
        <locale-config>
            <default-locale>en</default-locale>
            <supported-locale>ro</supported-locale>
        </locale-config>

    </application>
    <navigation-rule id="main"> …
    ...
</faces-config>
```

Default JSF messages

Global Resource Bundles

Supported Locales

Navigation Rules

JSF Architecture

“The learning curve is steep”, “We recommend teams use simple frameworks and embrace and understand web technologies including HTTP, HTML and CSS.”, etc.

- **User Interface Component** Model
- **Component Rendering** Model
- **Backing / Managed Beans** Model
- **Conversion** Model
- **Event and Listener** Model
- **Validation** Model
- **Navigation** Model

User Interface Component Classes

- Specify the component **functionality**, such as holding component state, maintaining a reference to objects, driving event handling and rendering for a set of standard components.
- Base class: **UIComponentBase** subclass of **UIComponent**, which defines the default state and behavior
- ~~No definition of visual representation~~
- Examples:

```
– UICommand    UIForm      UIPanel
– UIOutput     UIInput     UIMessage
– UIData       UIColumn    UIGraphic ...
```

Behavioral Interfaces

- The component classes also implement one or more behavioral interfaces, each of which defines certain behavior for a set of components whose classes implement the interface.
  - *ActionSource, ActionSource2*

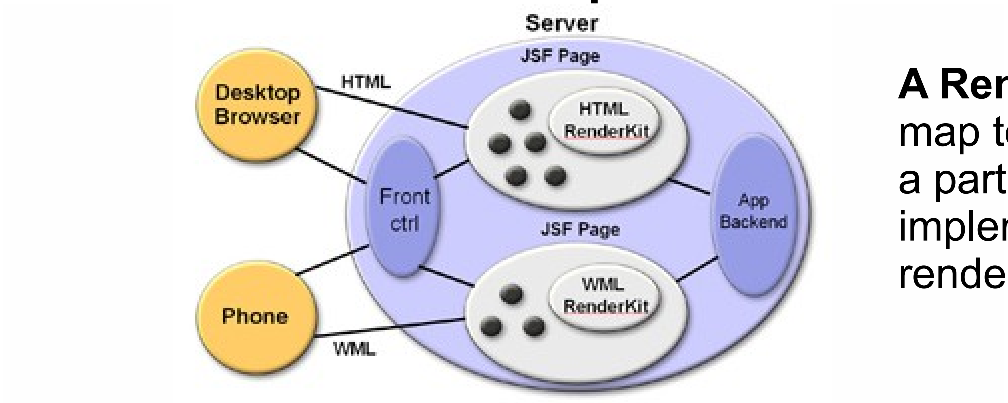
    ActionSource is an interface that may be implemented by any concrete UIComponent that wishes to be a source of ActionEvents, including the ability to invoke application actions via the default ActionListener mechanism.
  - *ValueHolder, EditableValueHolder*
  - *NamingContainer, StateHolder, …*

Examples:

```
UICommand  implements  ActionSource2, StateHolder
UIOutput   implements  StateHolder, ValueHolder
UIInput    implements  EditableValueHolder, StateHolder, ValueHolder
```

Component Rendering Model

- **Renderer class** → the **view** of a component
  - define the behavior of a component once
  - create multiple renderers (Strategy Design Pattern)



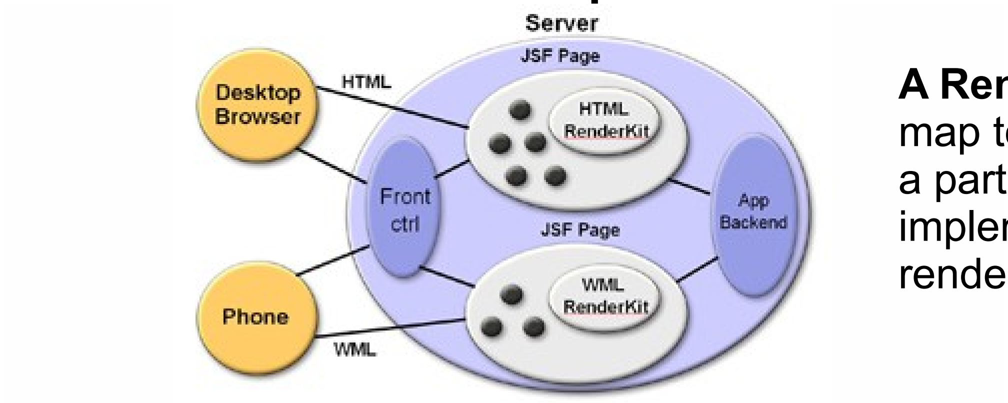
A Render kit defines how component classes map to component tags that are appropriate for a particular client. The JavaServer Faces implementation includes a standard HTML render kit for rendering to an **HTML** client.

- A **component tag** identifies the **renderer**

Component	Representation	Tag
UICommand	HtmlCommandButton	**commandButton**
	HtmlCommandLink	**commandLink**
UISelectOne	HtmlSelectOneMenu	**selectOneMenu**
	HtmlSelectOneRadio	**selectOneRadio**
	HtmlSelectOneListbox	**selectOneListbox**

Component Rendering Model

- **Renderer class** → the **view** of a component
  - define the behavior of a component once
  - create multiple renderers (Strategy Design Pattern)

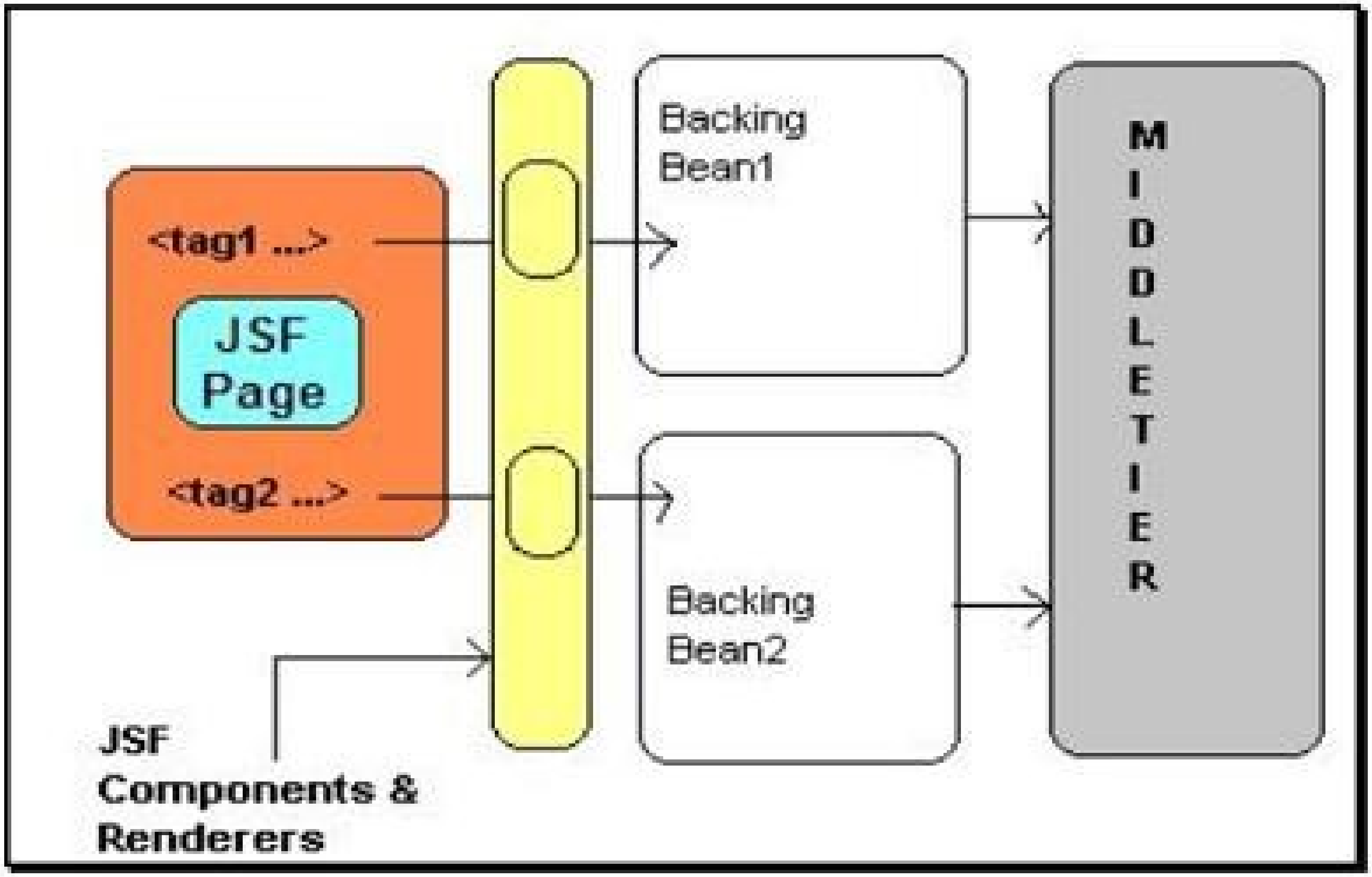


A Render kit defines how component classes map to component tags that are appropriate for a particular client. The JavaServer Faces implementation includes a standard HTML render kit for rendering to an **HTML** client.

- A **component tag** identifies the **renderer**

Component	Representation	Tag
UICommand	HtmlCommandButton	**commandButton**
	HtmlCommandLink	**commandLink**
UISelectOne	HtmlSelectOneMenu	**selectOneMenu**
	HtmlSelectOneRadio	**selectOneRadio**
	HtmlSelectOneListbox	**selectOneListbox**

Backing / Managed Beans Model



Value Binding

Value Binding

PersonBean → String name;

```
<h:inputText id="userName" value="#{personBean.name}"/>
```

```
@ManagedBean(name = "personBean")
@SessionScoped
public class PersonBean implements Serializable {

  private String name;

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

Reference Binding

Reference Binding
PersonBean → UIInput nameComponent;
```
<h:inputText id="userName" binding="#{personBean.nameComponent}"/>
```

```
public class PersonBean {
  ...
  private UIInput nameComponent;

  public UIInput getNameComponent() {
    return nameComponent;
  }
  public void setNameComponent(UIInput nameComponent) {
    this.nameComponent = nameComponent;
  }
  public void someMethod() {
    nameComponent.setRendered(false);
  }
}
```

The Conversion Model

- When a component is bound to a managed bean, the application has two views of the component's data:
  - The **model view**, in which data is represented as **data types**, such as `java.util.Date,double,etc.`
  - The **presentation view**, in which data is represented in a manner that can be read or modified by the user, such as a **text string** in the format `2014-10-29` or `$1,234.99`
- JSF automatically converts component data between these two views when the bean property associated with the component is of one of the types supported by the component's data.

Using Standard Converters

```
<h:inputText value="#{personBean.salary}"
     converter="javax.faces.convert.BigDecimalConverter"/>

<h:outputText value="#{personBean.birthday}">
  <f:convertDateTime pattern = "yyyy-MM-dd"/>
</h:outputText>

<h:outputText value="#{product.price}" >
  <f:convertNumber type="currency" />
</h:outputText>
```

Using Custom Converters

```
@FacesConverter(value = "urlConverter")
  // or @FacesConverter(forClass = "java.net.URL")
  // or using <converter> in faces-config.xml

public class URLConverter implements Converter {

  // Presentation -> Model
  public Object getAsObject(FacesContext context,
      UIComponent component, String newValue) throws ConverterException {
    try {
      return new URL(newValue);
    } catch(Exception e) {
      return new ConverterException("Hey, conversion error!");
    }
  }
  // Model -> Presentation
  public String getAsString(FacesContext context,
      UIComponent component, Object value) throws ConverterException {
    return String.valueOf(value);
  }
}
```

```
<h:inputText id="url" value="#{bean.url}" converter="urlConverter" />
```

Converter Example

```
public abstract class AbstractConverter<T> implements Converter {
  public Object getAsObject(FacesContext context, UIComponent component, String value) {
        if (value == null || value.trim().equals("")) return null;
        try {
            return getAsObject(value);
        } catch (NoResultException e) {
            System.err.println("cannot convert " + value + "\n" + e);
            return null;
        }
    }
  public String getAsString(FacesContext context, UIComponent component, Object value) {
        if (value == null) return "";
        return getAsString(value);
    }
    protected abstract T getAsObject(String value);

    protected String getAsString(Object value) {
        return ((T) value).toString();
    }
}
```

```
@FacesConverter(forClass = Country.class)
public class CountryConverter extends AbstractConverter<Country> {

    @Override
    protected Object getAsObject(String value) {
        return CountryRepository.findByName(value);
    }
}
```

Converter Example (xhtml)

```
<p:selectOneMenu id="city"
                 value="#{personEdit.city}"
                 converter="#{cityConverter}">
   <f:selectItems value="#{personEdit.cities}" />
   <f:selectItem itemValue="#{null}" itemLabel="-" />
</p:selectOneMenu>

<p:selectOneMenu id="country" required="true"
                 value="#{personEdit.country}">
  <f:selectItems value="#{personEdit.countries}" />
</p:selectOneMenu>
```

```
@ManagedBean
public class PersonEdit  {
  private Country country;
  private City city;
  private List<Country> countries;
  private List<City> cities;   …
  //getters, setters
}
```

Event and Listener Model

- **Application Events**
  - Generated by *UIComponent*s
  - `ActionEvent`
  - `ValueChangeEvent`
- **System Events**
  - Generated during the execution of an application at predefined times. They are applicable to the entire application rather than to a specific component: `PostConstructApplicationEvent`, `PreDestroyApplicationEvent`, `PreRenderViewEvent`
- **Data Model Events**
  - Occurs when a new row of a UIData component is selected

Registering Listeners

An application developer can implement listeners as classes or as managed bean methods.

- **ActionListener**

```
<h:commandLink id="create"
               action="createPerson">
  <f:actionListener type="somepackage.SomeActionListener" />
</h:commandLink>
```

- **ValueChangedListener**

```
<h:inputText id="name" size="50"
             value="#{personBean.name}"
             required="true"
             valueChangeListener="#{personBean.nameChanged}">
  <f:valueChangeListener type="somepackage.SomeValueChangeListener" />
</h:inputText>
```

Implementing Event Listeners

Using a class

```
public class SomeActionListener implements ActionListener {
    @Override
    public void processAction(ActionEvent ae)
                          throws AbortProcessingException {
      UIComponent ui = ae.getComponent();
      System.out.println("Event source is: " +
           ui.getClass().getName());
    }
}
```

Using a backing bean method

```
public class PersonBean {
    …
     public void nameChanged(ValueChangeEvent event) {
        System.out.println("Value change: " +
           event.getOldValue() " -> "event.getNewValue());
    }
}
```

Validation Model

- We need to **validate the local data** of editable components (such as text fields) before the corresponding model data is updated to match the local value.
  - **Standard validators**

```
<h:inputText id="name" value="#{personBean.name}"
             required="true"/>
  <f:validateLength minimum="1" maximum="50"/>
</h:inputText>
```

  - **Custom validators**
    - bean methods
    - *Validator* classes

Using Custom Validators

- Using a validation method inside a bean

```
public void validateEmail(FacesContext context,
    UIComponent component, Object value) {
      String email = (String) value;
      if (email.indexOf('@') == -1) {
        ((UIInput)component).setValid(false);
        context.addMessage(component.getClientId(context),
          new FacesMessage("Bad email"));
    }
}
<h:inputText value="#{user.email}" validator="#{user.validateEmail}"/>
```

- Using a validator class

```
@FacesValidator(value="emailValidator")
public class EmailValidator implements Validator {
  public void validate(FacesContext context,
       UIComponent component, Object value) {
    … throw new ValidatorException("Bad email");
  }
}
<h:inputText id="email" value="#{user.email}" >
  <f:validator validatorId="emailValidator" />
</h:inputText>
```

Using Bean Validators

- **The Bean Validation model** is supported by constraints in the form of annotations placed on a field, method, or class of a JavaBeans component, such as a managed bean.
- Constraints can be built in or user defined.
- Example

```
  public class Person {

    @Pattern(regexp = "^$|[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\.[a-z0-9!
#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-
9](?:[a-z0-9-]*[a-z0-9])?")
    @Size(max = 100)
    @NotNull
    private String email;
    …
}
```

Validation error messages

- Use *validatorMessge* tag attribute

```
<p:inputText id="email"
             value="#{personEdit.entity.email}"
             validatorMessage="#{msg['email.invalid']}">
  ...
</p:inputText>
```

- Or override the defaults: *Messages.properties*

```
# ==================================================================
# Converter Errors
# ==================================================================
javax.faces.converter.DateTimeConverter.DATE =
 {2}: ''{0}'' could not be understood as a date.
javax.faces.converter.DateTimeConverter.DATE_detail =
 {2}: ''{0}'' could not be understood as a date. Example: {1}
...
# ===================================================================
# Validator Errors
# ===================================================================
javax.faces.validator.LengthValidator.MAXIMUM=
 {1}: Validation Error: Length is greater than allowable maximum of ''{0}''
javax.faces.validator.LengthValidator.MINIMUM=
 {1}: Validation Error: Length is less than allowable minimum of ''{0}''
...
```

The Navigation Model

- **Navigation** is a <u>set of rules for choosing the next page</u> or view to be displayed after an application action, such as when a button or link is clicked.

```
<h:commandButton id="submit" value="Submit" action="success"/>
```

ActionEvent → default ActionListener → NavigatorHandler

- Navigation can be
  - **implicit**: “success” → success.xhtml
  - **user-defined**: configured in the application configuration resource files (faces-configx.xml)

```
<navigation-rule>
    <from-view-id>/greeting.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

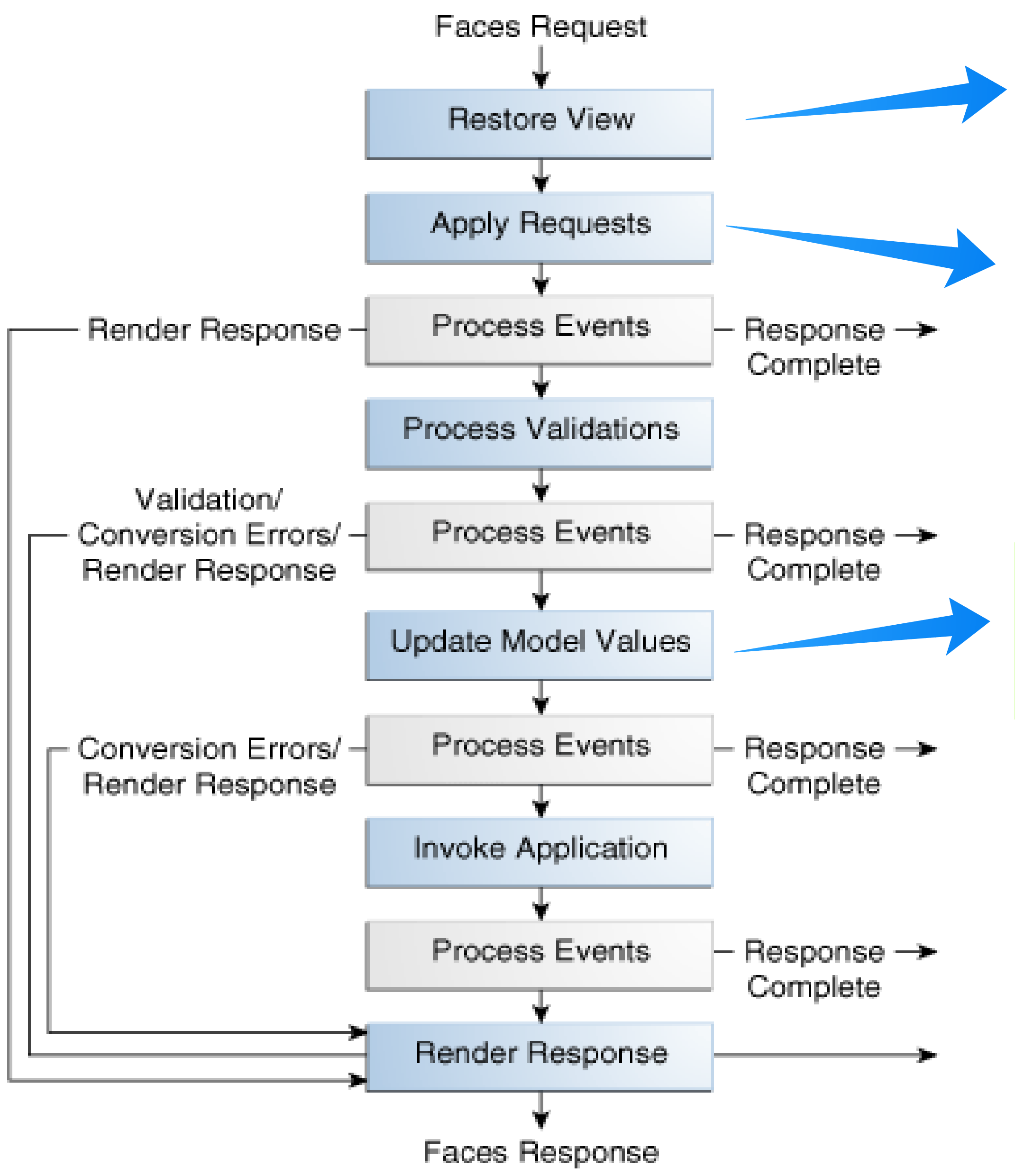
$$\begin{tikzcd}
\text{/greeting.xhtml} \arrow[d, "\text{success}"] \\
\text{/response.xhtml}
\end{tikzcd}$$

The Lifecycle of a JSF Application

- Receive incoming HTTP request made by the client
  - *initial requests and postbacks*
- Decode parameters from the request
  - *validate and convert the input data*
- Modify and save the state
  - *A page is represented by a tree of components, called a* ***view****. The view is built considering the state saved from a previous submission of the page.*
- Render the response to the client

FacesContext contains all of the per-request state information related to the processing of a single JavaServer Faces request, and the rendering of the corresponding response. It is passed to, and potentially modified by, each phase of the request processing lifecycle.

Request – Response in JSF



Builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the FacesContext instance

Each component in the tree extracts its new value from the request parameters by using its decode method. The value is then stored locally on each component.

The data is valid. The corresponding server-side object properties (the bean properties pointed at by an input component's value attribute) are set to the components' local values.

Delegates authority to the appropriate resource for rendering the pages. If the request is a postback and errors were encountered, the original page is rendered again.

Monitoring the Lifecycle

```
public class PhaseLogger implements PhaseListener {
  public PhaseLogger() {
    System.out.println("PhaseLogger created.");
  }
  public void beforePhase(PhaseEvent event) {
    System.out.println("BEFORE - " + event.getPhaseId());
  }
  public void afterPhase(PhaseEvent event) {
    System.out.println("AFTER - " + event.getPhaseId());
  }
  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

faces-config.xml

```
<lifecycle>
  <phase-listener>
    myapp.listeners.PhaseLogger
  </phase-listener>
</lifecycle>
```

The *immediate* attribute

- The **immediate** attribute indicates whether user inputs are to be processed early in the application lifecycle or later.
- “Flag indicating that, if this component is activated by the user, notifications should be delivered to interested listeners and actions immediately (that is, during Apply Request Values phase) rather than waiting until Invoke Application phase.”
- Use case: the *Cancel button* in a dialog

JSF and AJAX

- **AJAX** = Asynchronous JavaScript and XML.
- Uses JavaScript (XMLHttpRequest)to send data to the server and receive data from the server asynchronously.
- The javascript code exchanges data with the server and updates parts of the web page without reloading the whole page.
- **<f:ajax>** (in PrimeFaces <p:ajax>)

Ajax Events and Listeners

- Events: *click, keyup, mouseover, focus, blur, etc*

```
<h:outputText id="random" value="#{someBean.randomNumber}"/>
<h:selectBooleanCheckbox value="Check box" >
    <f:ajax event="click" render="random"/>
</h:selectBooleanCheckbox>
```

- Listeners

```
<f:ajax listener="#{someBean.someAction}" render="someComponent" />

public class SomeBean {
  public void someAction(AjaxBehaviorEvent event) {
      //do something ...
  }
}
```

- Frequent use cases

```
 -  selectOne: change
 -  calendar: change, dateSelect
 -  dataTable: rowSelect, rowDblSelect
 -  autoComplete: itemSelect
 -  dialog: close
```

PrimeFaces

```
<p:ajax
  event="keyup"
  process="list of comp ids"
  update="list of comp ids" />
```

Naming Containers

- **Naming containers** affect the behavior of the *UIComponent.findComponent(java.lang.String)* and *UIComponent.getClientId()* methods
- Forms, Trees, DataTables, etc.
- Example

```
<h:outputText id="test" value="Hello 1" />
<h:form id="form1">
  <h:outputText id="test" value="Hello 2" />
</h:form>
<h:form id="form2">
  <h:outputText id="test" value = "Hello 3" />
</h:form>
...
  <f:ajax event="click" render="test"/>
  <f:ajax event="click" render=":form1:test"/>
  <f:ajax event="click" render=":form2:test"/>
```

relative

:absolute

Poll

- **Polling** is the continuous checking of other programs or devices by one progam or device to see what state they are in.
- **<p:poll>** makes ajax calls periodically:

```
<p:poll interval="3"
        listener="#{counterBean.increment}"
        update="counter" />
<h:outputText id="counter"
              value="#{counterBean.count}" />
```

Web Push

- One-way, **server-to-client**, **websocket** based communication.
- **<f:websocket>** (JSF 2.3+)

```
<f:websocket channel="push">
  <f:ajax event="updateMyData" render=":dataTable" />
</f:websocket>
```

- **@Push**

```
@Inject @Push
private PushContext push;
…
public void someMethod() {
    push.send("updateMyData");
}
```

What is the *websocket* protocol?

Facelets

The Context

- JSF offers an **API for representing UI components** and managing their **server-side** state
  - html_basic (h), jsf_core (f) custom tag libraries
- <u>What language</u> do we use to actually write the pages?
  - First attempt: JSP
  - ...

Using JSP in JSF

hello.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<f:view>
  <html>
    <head>
      <title>JSF - JSP</title>
    </head>
    <body>
      <h:outputText value="Hello"/>
    </body>
    </html>
</f:view>
```

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

~~http://localhost:8080/myapp/hello.jsp~~

http://localhost:8080/myapp/faces/hello.jsp

What Is Facelets?

- **Page declaration language** that is used to build JavaServer Faces views using HTML style templates and to build component trees.
- Use of **XHTML** for creating web pages
- Support for **Facelets tag libraries** in addition to **JavaServer Faces** and **JSTL** tag libraries
- Support for the Expression Language (EL)
- **Templating** for components and pages
- **Composite components**
- Faster compilation time, High-performance rendering, etc.

Using Facelets

hello.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Hello JSF - Facelets</title>
  </h:head>
  <h:body>
    <h:outputText value="Hello"/>
  </h:body>
</html>
```

http://localhost:8080/myapp/faces/hello.xhtml

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

Facelets Libraries

	URI: http://java.sun.com/	Prefix
JSF Facelets	jsf/facelets	ui:
JSF Composite	jsf/composite	composite:
JSF HTML	jsf/html	h:
JSF Core	jsf/core	f:
JSTL Core	jsp/jstl/core	c:
JSTL Functions	jstl/functions	fn:

Facelets Templates

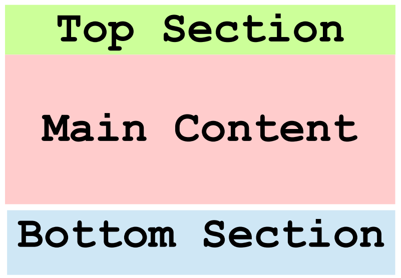
- JavaServer Faces technology provides the tools to implement user interfaces that are **easy to extend and reuse**.
- Templating is a useful Facelets feature that allows you to create **a page that will act as the base**, or **template**, for the other pages in an application.
- By using templates, you can <u>reuse code</u> and avoid recreating similarly constructed pages. Templating also helps in maintaining a standard look and feel in an application with a large number of pages.

Using Facelets Templates

template.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Template</title>
  </h:head>

  <h:body>
    <div id="top" class="top">
      <ui:insert name="top">Top Section</ui:insert>
    </div>
    <div id="content" class="main">
      <ui:insert name="content">Main Content</ui:insert>
    </div>
    <div id="bottom" class="bottom">
      <ui:insert name="bottom">Bottom Section</ui:insert>
    </div>
  </h:body>
</html>
```



Creating a Page from a Template

hello.xhtml

```
<ui:composition template="/WEB-INF/template.xhtml"
                xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://xmlns.jcp.org/jsf/html"
                xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
   <ui:define name="top">
      Welcome to Facelets
   </ui:define>
   <ui:define name="content">
     <h:graphicImage value="#{resource['images:hello.jpg']}"/>
     <h:outputText value="Hello Facelets!"/>
   </ui:define>
   <ui:define name="bottom">
     <h:outputLabel value="Power to the Facelets"/>
   </ui:define>
</ui:composition>
```

Using Parameters

- template.xhtml

```
<html ...>
    <f:view contentType="text/html" encoding="UTF-8">
        <h:head>
            <f:facet name="first">
                <title>
                    #{pageTitle}
                </title>
            </f:facet>
        </h:head>
</html>
```

- hello.xhtml

```
<ui:composition template="/WEB-INF/template.xhtml" ...>

    <ui:param name="pageTitle" value="#{msg['main.title']}" />
    ...
</ui:composition>
```

Composite Components

- A composite component is a special type of **template that acts as a component**.
- Any component is essentially a piece of reusable code that behaves in a particular way.

  A composite component consists of a collection of markup tags and other existing components. This reusable, user-created component has a customized, defined functionality and can have validators, converters, and listeners attached to it like any other component.
- Using the resources facility, the composite component can be stored in a library that is available to the application from the defined resources location.

Creating a Composite Component

resources/ezcomp/email.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:composite="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>This content will not be displayed</title>
  </h:head>
  <h:body>

    <composite:interface>
      <composite:attribute name="value" required="false"/>
    </composite:interface>

    <composite:implementation>
      <h:outputLabel value="Email id: " />
      <h:inputText value="#{cc.attrs.value}" />
    </composite:implementation>

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

Using a Composite Component

somePage.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ez="http://java.sun.com/jsf/composite/ezcomp/">
  <h:head>
    <title>Using a sample composite component</title>
  </h:head>
  <body>
    <h:form>
      <ez:email value="Enter your email id" />
    </h:form>
  </body>
</html>
```

Web Resources

- Web resources are any software artifacts that the web application requires for proper rendering, including images, script files, and any user-created component libraries.
- Resource identifiers are unique strings that conform to the following format:

[locale-prefix/][library-name/][library-version/]resource-name[/resource-version]

```
<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

This tag specifies that the image named wave.med.gif is in the directory **resources/images**.

PrimeFaces SHOWCASE

- PrimeFaces is a popular open source framework for JavaServer Faces featuring over 100 components, touch optimized mobilekit, client side validation, theme engine and more.
- Check out:

  https://www.primefaces.org/showcase/index.xhtml