JavaServer Faces

Student Workbook

itcourseware

TRAINING MATERIALS FOR IT PROFESSIONALS

JavaServer Faces

Author: Jamie Romero

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Author: Brandon Caldwell, Jim Gallentine

Editor: Jan Waleri

Editorial Staff: Ginny Jaranowski

Special thanks to: The many instructors whose ideas and careful review have contributed to the quality of this workbook and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Describe the JavaServer Faces (JSF) architecture.
- Build a JSF component tree with Core and HTML tag libraries.
- Create JavaBeans with properties and methods that are bound to JSF components.
- Describe the six phases of the JSF request-processing lifecycle.
- Use both Action Events and Value Change Events to react to user interface interactions.
- Use the data table component to present tabular information.
- Write your own validators and converters to ensure the legality of user input.
- Use the **<f:ajax>** tag to send an asynchronous request to the server.

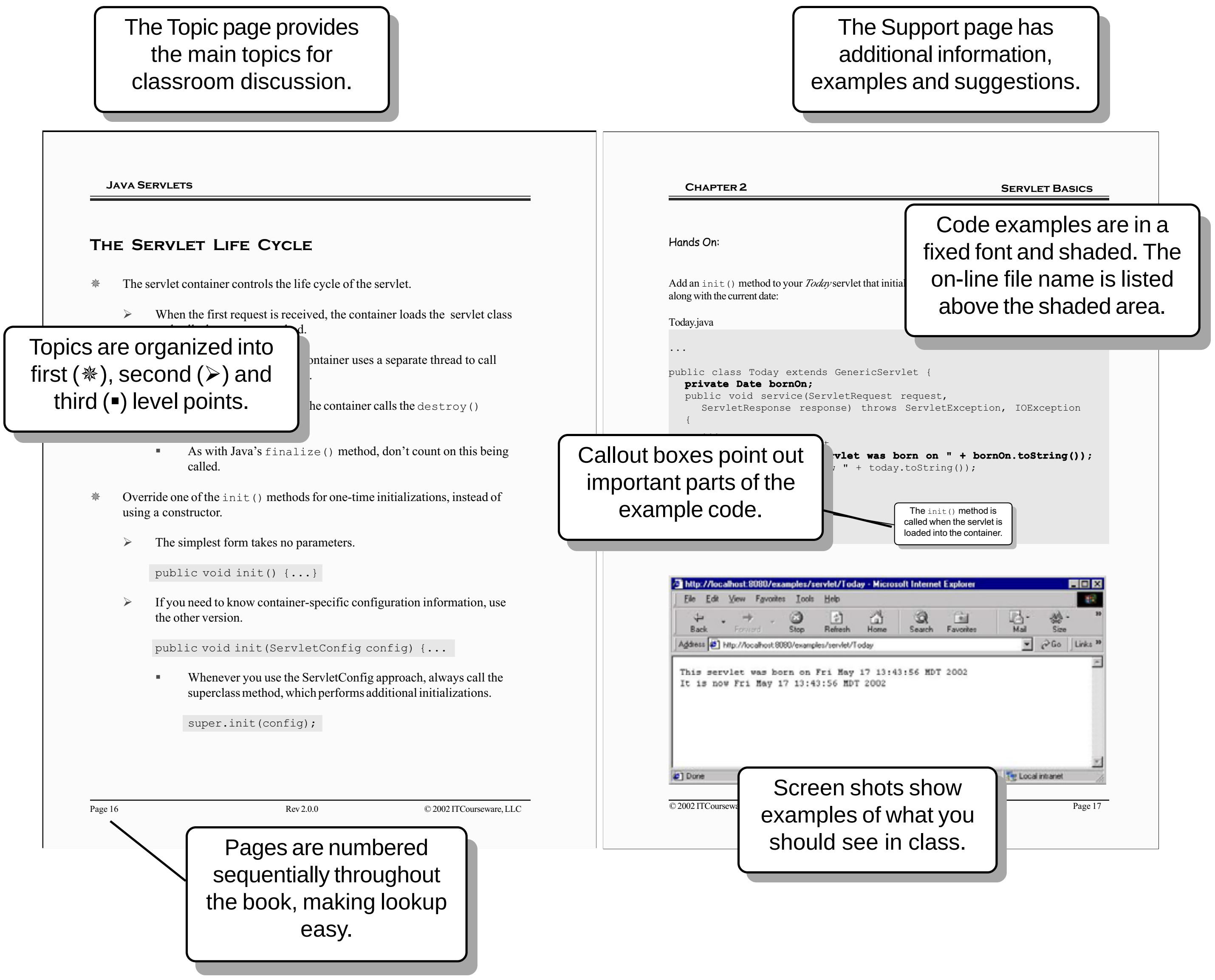
COURSE OVERVIEW

- **Audience**: Web developers who want to efficiently create complex web applications.
- **Prerequisites**: Java and HTML experience is recommended. Servlet and JSP experience is helpful.
- **Classroom Environment:**
  - A workstation per student.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.


The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

❋ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class

ontainer uses a separate thread to call

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

❋ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

`public void init() {...}`

➢ If you need to know container-specific configuration information, use the other version.

`public void init(ServletConfig config) {...`

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

`super.init(config);`

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

Topics are organized into first (❋), second (➢) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.

CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Today.java

```
...
public class Today extends GenericServlet {
  private Date bornOn;
  public void service(ServletRequest request,
    ServletResponse response) throws ServletException, IOException
  {
vlet was born on " + bornOn.toString());
; " + today.toString());
```

The `init()` method is called when the servlet is loaded into the container.

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

© 2002 ITCoursewa Page 17

Screen shots show examples of what you should see in class.

Suggested References

Geary, David and Cay Horstmann. 2010. ***Core JavaServer Faces, ThirdEdition***. Prentice Hall, Upper Saddle River, NJ. ISBN 0137012896.

Schalk, Chris and Ed Burns. 2010. ***JavaServer Faces 2.0: The Complete Reference***. McGraw-Hill Osborne Media, New York, NY. ISBN 0071625097.

Leonard, Anghel. 2010. ***JSF 2.0 Cookbook***. Packt Publishing, Birmingham, UK. ISBN 1847199526.

https://javaserverfaces.dev.java.net/
http://myfaces.apache.org/
http://www.jsfcentral.com/
http://www.jsftutorials.net/
http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

CHAPTER 2 - GETTING STARTED WITH JSF

OBJECTIVES

- Describe the JavaServer Faces architecture.
- Identify the most important parts of a JSF application, including JSF components and managed beans.
- Build and deploy a simple JSF application.

GUI Development

- Client-side Java developers typically use Java's Swing user interface packages to build Graphical User Interfaces (GUIs).
    - Swing provides dozens of components, from simple buttons and text fields, to complex tables and tree views.
    - Swing components generate events that invoke developer's listener code.
        - An *event* occurs when a user interacts with a GUI component.
        - An *event listener* is invoked when its associated event occurs.
- Server-side Java developers can use JavaServer Pages (JSPs) to generate HTML that is rendered in the client's browser.
    - HTML contains a limited set of user interface components.
        - Many developers resort to JavaScript, Java Applets, Flash, or other technologies for more complex, browser-based user interfaces.
    - When a change occurs that the backend code needs to react to, a request is submitted to the server.
        - A Java Servlet typically receives the request and acts as a controller, determining what to do next.
    - Common tasks, such as conversions, validation, and navigation, must be coded by hand.

The Model-View-Controller (MVC) architecture has been used for years in client-side GUIs. In MVC, the *model* represents the data to be presented. The *view* is made up of the code that handles presentation. The *controller* handles decision making, and serves as a conduit between the model and the view. MVC architectures excel at separating the different concerns involved with GUI development, thus making applications more maintainable.

Java web developers have adapted MVC for use in web applications. Java Servlets act as controllers, JavaBeans serve as the model, and JSPs handle the view.

The Jakarta Struts project was created to provide an MVC framework for Java Web Applications. Struts provides the controller Servlet for you, as well as good built-in support for validation and navigation. Struts focuses on the control aspect of web applications, rather than the view, or GUI, aspect.

JavaServer Faces

- JavaServer Faces (JSF) brings event-driven, component-based GUI development to the server-side.
  - JSF defines GUI components that are independent of a presentation technology.
    - It provides tag libraries that you use to interact with built-in components or allows you to build your own (or download) custom components.
  - JSF specifies an event model so that your server-side GUIs can be coded like client-side GUIs.
    - You can create listeners to handle events that are generated by JSF components.
  - JSF handles basic conversions, validation, and navigation for you, unlike JSP/Servlet applications, where you need to write this code yourself.
- JavaServer Faces clearly separates the view logic from the business and control logic.
  - JavaBeans provide a layer of abstraction between the user interface and the business logic.
  - The provided **FacesServlet** handles all control logic on your behalf, based on code level annotations or entries in *faces-config.xml*.
  - JSF tag libraries typically generate the view that is rendered within the client's browser.
- JavaServer Faces is a specification, not an implementation.
  - You will need a JSF implementation in order to use it.

Two major JSF implementations are available:

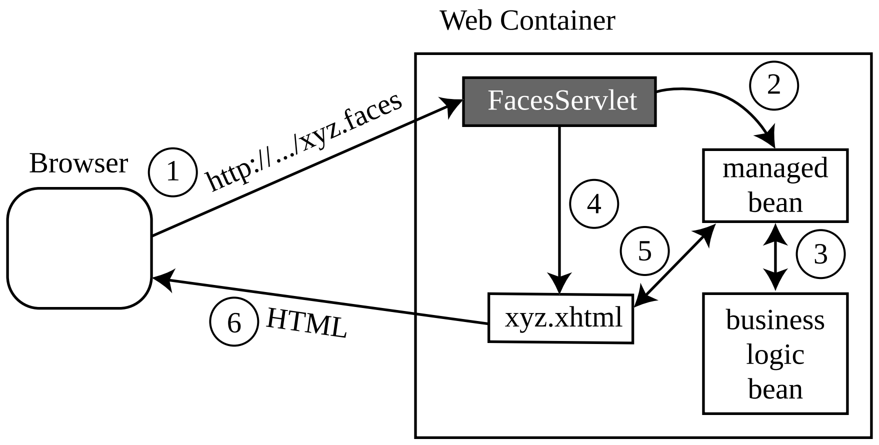
Oracle Mojarra, the JSF Reference Implementation — ***https://javaserverfaces.dev.java.net/***
Apache MyFaces Core Project — ***http://myfaces.apache.org/core20/index.html***

An implementation is guaranteed to support the standard components defined in the JSF specification. A JSF component library provides additional components and/or capababilities. Some of the more popular component libraries include:

IceFaces — ***http://www.icefaces.org/main/home/***
PrimeFaces — ***http://www.primefaces.org/***
JBoss RichFaces — ***http://www.jboss.org/richfaces***
Apache Trinidad — ***http://myfaces.apache.org/trinidad/index.html***
Apache Tomahawk — ***http://myfaces.apache.org/tomahawk/index.html***

A JSF Application

- A typical JSF application consists of Facelets and JavaBeans.
    - A *Facelet* is an XML file, typically XHTML, with embedded tags that describe how to construct a JSF view.
        - As of JSF 2.0, Facelets replace JSPs as the default JSF view technology.
    - JavaBeans, known as *managed beans*, serve as intermediaries to integrate the back-end business and data access logic with the rest of the application.


Web Container
FacesServlet
Browser
1
http://.../xyz.faces
2
managed bean
4
5
3
xyz.xhtml
business logic bean
6
HTML

JSF Components

- The user interface is typically generated by one or more Facelets.
- Use JSF tag libraries, rather than traditional JSP elements, to generate the view.
    - The *Core tag library* contains tags that are independent of a presentation technology.
    - The *HTML tag library* contains tags that are used to generate HTML output.
        - Typical HTML tags include **<h:form>**, **<h:inputText>**, and **<h:commandButton>**.
    - Other tag libraries can be incorporated for templating or to generate non-HTML content.
- Each JSF tag represents an underlying Java object, called a *component*.
    - Components are combined into a renderable component tree.
    - When it is time to generate the view, the tree is traversed and each component renders itself to HTML.

Examples/WebContent/sayHello.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Say Hello</title>
  </h:head>
  <h:body>
    <h:form>
      Name:
       <h:inputText value="#{helloBean.name}"/>
       <h:commandButton value="Submit"
         action="#{helloBean.createGreeting}" />
    </h:form>
  </h:body>
</html>
```

Managed Beans

- *Managed beans* are JavaBeans that separate the presentation logic from the business and data access logic in a JSF application.
  - Business and data access logic are typically contained in JavaBeans or Enterprise JavaBeans.
  - Presentation logic is usually embedded in Facelets.
- Declare properties in your managed beans via **get**s and **set**s, that correspond to components in the view.
  - Whenever a view component is rendered, it queries its associated property for its value.
    - Your **get()** method is called.
  - Whenever the user submits a request, the view component updates the property with its value.
    - Your **set()** method is called.
- Request submissions can invoke methods on your managed bean.
  - JSF uses the returned **String** to determine which Facelet to display next.

Examples/src/hello/HelloBean.java

```
package hello;

import  javax.faces.bean.ManagedBean;

@ManagedBean
public class HelloBean {
  private String name = "Duke";
  private String greeting;

  public HelloBean()  {
  }
  public String getName() {
    return name;
  }
  public String getGreeting() {
    return greeting;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String createGreeting() {
    greeting = "Hello " + name;
    return "displayGreeting";
  }
}
```

The **name** property is derived from the **getName()** and **setName()** methods.

The returned **String** is used to navigate to the **displayGreeting** Facelet.

Examples/WebContent/displayGreeting.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"
       xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Say Hello Results</title>
  </h:head>
  <h:body>
    <h4>#{helloBean.greeting}</h4>
  </h:body>
</html>
```

JSF Application Structure

- JSF applications are structured like other Java web applications.
  - Place all XHTML files at the root level of your application.
  - Place images, css files, and other supporting files in the *resources* subdirectory.
  - Place *web.xml* in the *WEB-INF* subdirectory.
  - JavaBeans and other supporting Java *.class* files go under the *WEB-INF/ classes* directory.
    - Make sure that the code's package structure is reflected appropriately with subdirectories.
  - *WEB-INF/lib* contains *.jar* files that your application depends on.
    - If you deploy a JSF application to a web container that does not support JSF, you can place the necessary JSF *.jar* files here.
- Bundle the entire directory structure within a *.war* file.
- Many developers use the **ant** build tool from Apache to build and deploy the *.war* file.
  - Embed instructions to **ant** within a file named *build.xml*.

Examples/build.xml

```
<?xml version="1.0"?>
<project name="HelloJSF" default="all" basedir=".">
  <property environment="env"/>
  <property name="buildDir" value="${basedir}/build/classes"/>
  <property name="distDir" value="${basedir}/dist"/>
  <property name="webinfDir" value="${basedir}/WebContent/WEB-INF"/>
  <property name="packageDir" value="${basedir}/src/hello"/>
  <property name="warName" value="HelloJSF"/>
  <property name="warFile" value="${warName}.war"/>

  <target name="all" depends="clean, init, compile, create_war,
    deploy_war"/>

  <target name="clean">
    <delete dir="${distDir}"/>
    <delete includeemptydirs="true">
      <fileset dir="${buildDir}" includes="**/*"/>
    </delete>
    <delete dir="${webinfDir}/classes"/>
  </target>

  <target name="init">
    <mkdir dir="${distDir}"/>
    <mkdir dir="${webinfDir}/classes"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${packageDir}" destdir="${buildDir}"
           includes="**.java" includeantruntime="false">
      <classpath>
        <fileset dir="${env.JSF_HOME}/lib"/>
      </classpath>
    </javac>
  </target>

  <target name="create_war" depends="compile">
    <copy todir="${webinfDir}/classes">
      <fileset dir="${buildDir}"/>
    </copy>
    <war destfile="${distDir}/${warFile}"
         basedir="${basedir}/WebContent"
         needxmlfile="false"/>
  </target>
  ...
</project>
```

Try It:

Start Tomcat by running *%TOMCAT_HOME%\bin\startup.bat*. Then build and deploy the JSF application by running **ant** within the chapter's *Examples* directory.

RUNNING THE APPLICATION

- *web.xml* declares the provided **FacesServlet** that handles all control logic.

```
<servlet>
  <servlet-name>FacesServlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
</servlet>
```

- The **<servlet-mapping>** element ensures that all URLs that end in the *.faces* suffix are redirected to the **FacesServlet**.

```
<servlet-mapping>
    <servlet-name>FacesServlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

  - You don't access a Facelet with an *.xhtml* extension, you will instead use a *.faces* extension.
  - Some developers prefer to map the **FacesServlet** to the *.jsf* extension instead.

- You can optionally define a welcome file to allow a user to access your application with only the web context.

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

  - Place a link in your welcome file to the initial page of your application.

Examples/WebContent/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://java.sun.com/xml/ns/javaee"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">
  <servlet>
     <servlet-name>FacesServlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
     <servlet-name>FacesServlet</servlet-name>
     <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
     <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

index.html is the welcome file for this application.

Examples/WebContent/index.html

```
<html>
  <head>
    <title>Demo</title>
  </head>
  <body>
    <h2>Getting Started with JSF Demo</h2>
    <a href="sayHello.faces">Say Hello</a>
  </body>
</html>
```

Use the *.faces* extension, rather than *.xhtml*.

Try It:

In a web browser go to ***http://localhost:8080/HelloJSF/*** to access the welcome file for the JSF application.

Note:

Older versions of JSF required an additional configuration file: *faces-config.xml*. If you properly use annotations and conventions, then you can most likely omit this file.

LABS

❶ Modify *src/hello/HelloBean.java*. Replace the **name** property with **firstName** and **lastName** properties. Change the **createGreeting()** method to use both the first and last names when creating the greeting.
(Solution: *src/hello/HelloBean.java*)

❷ Modify *WebContent/sayHello.xhtml* to prompt for both first and last names. Verify your changes by redeploying the application and visiting *sayHello.faces* in your browser.
(Solution: *WebContent/sayHello.xhtml*)

CHAPTER 8 - AJAX

OBJECTIVES

❅ Use the **<f:ajax>** tag to send an asynchronous request to the server.

❅ Identify which components to execute and render using **<f:ajax>** attributes.

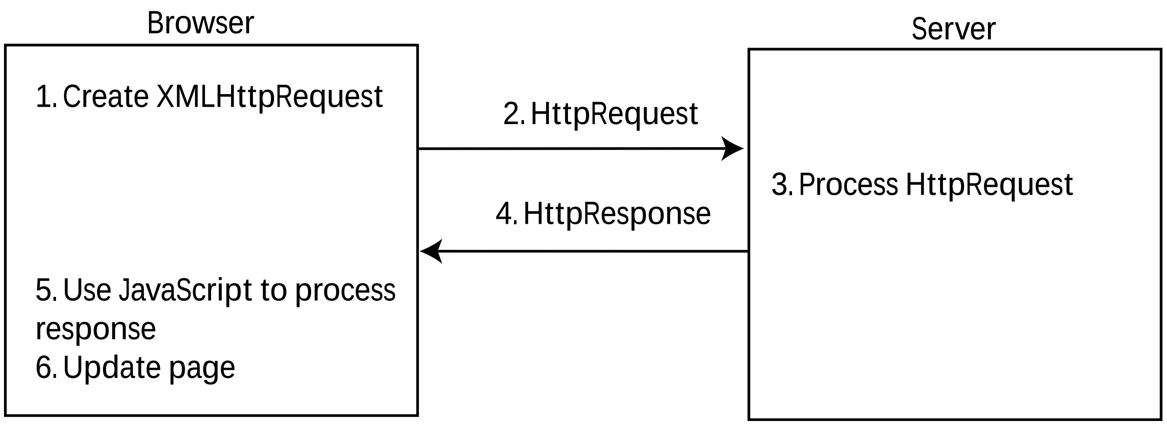
❅ Group multiple components within a single **<f:ajax>** tag.

❅ Run client-side JavaScript functions when an Ajax request is submitted and when an error occurs.

❅ Invoke server-side listener code when an Ajax request is submitted.

Ajax and JSF

- Traditional web applications submit all of their data to the server as an HTTP request, then update the full page given an HTTP response.
  - If all you want to do is validate a single form field and inform the user if it is invalid there may be some drawbacks:
    - The user experience is hampered by a full page reload.
    - The server load may be higher since it is processing more data than it needs to.
    - More bandwidth is consumed passing extra information back and forth that may not be necessary.
- Asynchronous JavaScript and XML (*Ajax*) allows you to exchange data with the server in the background without a full HTTP request/response.
  - You typically send a small subset of your form data to the server and update a small subset of your page as well.
    - Your web applications feel faster, more dynamic, and interactive because there is no full page reload and because the data exchange occurs asynchronously.
- JSF 2.0 added the **<f:ajax>** tag to enable you to create Ajax interactions in a standard way.
  - The underlying JavaScript library takes care of the data exchange with the server allowing you to focus on the specifics of which form components to send and which components to update.
- You can specify your managed bean as **@ViewScoped** to retain the bean instance across Ajax interactions



Examples/WebContent/color.xhtml

```
...
  <h:body>
    <h2>Color</h2>
    <h:form id="form">
      <h:panelGrid columns="2">
        Enter a Color:
        <h:inputText id="colorInput" value="#{colorBean.color}">
          <f:ajax event="keyup" execute="colorInput"
            render="colorOutput"/>
        </h:inputText>
      </h:panelGrid>
      <h:outputText id="colorOutput" value="#{colorBean.color}"
        style="background-color:#{colorBean.color}"/>
    </h:form>
  </h:body>
</html>
```

Examples/src/ajax/ColorBean.java

```
...
@ManagedBean(name="colorBean")
@ViewScoped
public class ColorBean implements Serializable {
  ...
}
```

Try It:

Run **ant** in the chapter's *Examples* directory. Visit ***http://localhost:8080/Ajax/color.faces***. Type in the name of a color in the text box to see your keystrokes echoed and the echo background change to the color you typed.

EVENTS

- Define the **<f:ajax>** tag as a child of the component that fires the event.

```
<h:inputText id="input" value="#{bean.property}">
  <f:ajax event="keyup"/>
</h:inputText>
```

- Specify the **event** attribute to identify the circumstance in which the Ajax request is triggered.

  - Command components trigger an **action** event when clicked.

```
<h:commandButton id="button" value="#{bean.method}">
  <f:ajax event="action"/>
</h:commandButton>
```

  - Modifying an input component results in a **valueChange** event.

```
<h:inputText id="input" value="#{bean.property}">
  <f:ajax event="valueChange"/>
</h:inputText>
```

  - You can also use JavaScript event names without the **on** prefix.

    Examples:
    - **blur** – an element loses focus
    - **focus** – an element gains focus
    - **keyup** – keyboard key is released
    - **click** – mouse clicks an element
    - **mouseover** – mouse is moved over an element

Examples/WebContent/color2.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"
       xmlns:h="http://java.sun.com/jsf/html"
       xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Color</title>
  </h:head>
  <h:body>
    <h2>Color</h2>
    <h:form id="form">
      <h:panelGrid columns="2">
        Enter a Color:
         <h:inputText id="colorInput" value="#{colorBean.color}">
           <f:ajax event="keyup" execute="colorInput"
             render="colorOutput"/>
        </h:inputText>
        Choose a Color:
         <h:selectOneMenu id="colorMenu" value="#{colorBean.color}">
           <f:selectItems value="#{colorBean.colors}"/>
           <f:ajax event="valueChange" execute="colorMenu"
             render="colorOutput"/>
        </h:selectOneMenu>

      </h:panelGrid>
       <h:outputText id="colorOutput" value="#{colorBean.color}"
         style="background-color:#{colorBean.color}"/>
    </h:form>
  </h:body>
</html>
```

Trigger an Ajax event whenever the dropdown menu's value changes.

Try It:

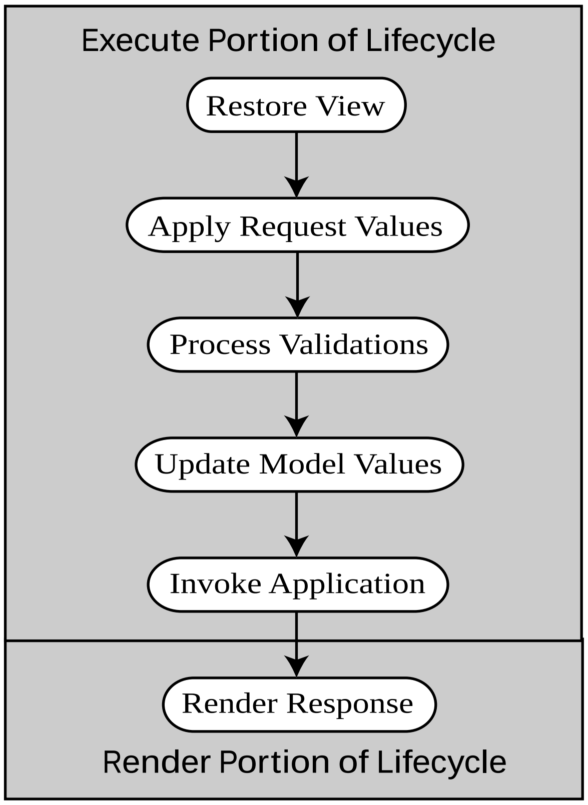
Visit ***http://localhost:8080/Ajax/color2.faces***. Both the text box and the dropdown menu will generate Ajax events.

EXECUTE AND RENDER

- Specify which subset of the components to execute and which to render using attributes of the **<f:ajax>** tag.
  - Executed components are passed to the server where the first five steps of the JSF lifecycle occur.
    - Conversions, validations, managed bean property updates, and bound action methods occur as usual.
    - Every Ajax request will have one or more executed components.
  - Rendered components are encoded back to the browser as part of the Render Response phase of the lifecyle.
    - Your Ajax requests can identify zero or more components to render.
- List the ids of each component to execute or render within the **<f:ajax>** tag's **execute** and **render** attributes.

```
<h:inputText id="input" value="#{bean.property}">
 <f:ajax event="keyup" execute="input" render="output"/>
</h:inputText>
```

  - Use spaces to separate multiple component ids.
  - JSF predefines four keywords that you can use instead of component ids:
    - **@this** – the component id of the parent of the **<f:ajax>** tag; **@this** is the default for the **execute** attribute.
    - **@form** – all component ids of the enclosing form.
    - **@all** – all components ids.
    - **@none** – no components ids; **@none** is the default for the **render** attribute.



Examples/WebContent/color3.xhtml

```
...
  <h:body>
        ...
         <h:inputText id="colorInput" value="#{colorBean.color}">
           <f:ajax event="keyup" execute="@this" render="colorOutput"/>
        </h:inputText>
        Choose a Color:
         <h:selectOneMenu id="colorMenu" value="#{colorBean.color}">
           <f:selectItems value="#{colorBean.colors}"/>
           <f:ajax event="valueChange" execute="@this"
             render="colorOut"/>
        </h:selectOneMenu>
        ...
    </h:form>
  </h:body>
</html>
```

Note:

You should not mix Ajax and regular HTTP requests. Instead, make the regular request an Ajax request as well.

GROUPING

- You can also place the **<f:ajax>** tag around a group of components so that all of the containing components will trigger an Ajax event.

```
<f:ajax>
  ...
   <h:inputText id="text" value="#{bean.text}"/>
   <h:selectOneMenu id="menu" value="#{bean.menuitem}">
  ...
</f:ajax>
```

  - The default event for command components is an **action** event; for input components it is a **valueChange** event.

  - You can explicitly specify the event that triggers the Ajax request using the **event** attribute on the **<f:ajax>** tag.

```
<f:ajax event="blur">
  ...
</f:ajax>
```

- You're allowed to nest **<f:ajax>** tags within each other:

```
<f:ajax event="blur">
  ...
     <h:inputText id="text" value="#{bean.text}">
       <f:ajax event="keyup"/>
     </h:inputText>
  ...
</f:ajax>
```

  - In the above example, both a **blur** and **keyup** event on the text box will trigger an Ajax request.

Examples/WebContent/color4.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"
       xmlns:h="http://java.sun.com/jsf/html"
       xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Color</title>
  </h:head>
  <h:body>
    <h2>Color</h2>
    <f:ajax render="colorOutput">
      <h:form id="form">
        <h:panelGrid columns="2">
          Enter a Color:
           <h:inputText id="colorInput" value="#{colorBean.color}"/>
          Choose a Color:
           <h:selectOneMenu id="colorMenu" value="#{colorBean.color}">
             <f:selectItems value="#{colorBean.colors}"/>
          </h:selectOneMenu>
        </h:panelGrid>
         <h:outputText id="colorOutput" value="#{colorBean.color}"
           style="background-color:#{colorBean.color}"/>
      </h:form>
    </f:ajax>
  </h:body>
</html>
```

Use the **render** attribute to identify which component(s) to update with the Ajax response.

Make sure to specify an **id** for each executed child component of **<f:ajax>.**

VALIDATION

- Upon submitting an Ajax request, validation and conversions occur as usual for any executed components.

  - Make sure to specify your error message component as a rendered component so that any error messages will be displayed.

```
<h:inputText id="input" label="textbox"
 value="#{bean.property}">
  <f:ajax event="keyup"/>
  <f:validator validatorId="myValidator"/>
</h:inputText>
<h:message for="input" id="validationMessage"/>
```

- Validation and conversion errors will cause the Render Response phase to run earlier in the lifecycle, skipping managed bean updates and method bindings.

Examples/WebContent/validate.xhtml

```
...
  <h:body>
    <h2>Validate</h2>
    <h:form id="form">
      <h:panelGrid columns="3">
        Enter a Color:
         <h:inputText id="validation" label="Validation"
             value="#{colorBean.color}">
           <f:ajax event="keyup"
             render="validationMessage colorOutput"/>
           <f:validator validatorId="colorValidator"/>
        </h:inputText>
         <h:message for="validation" styleClass="error"
           id="validationMessage"/>
      </h:panelGrid>
       <h:outputText id="colorOutput" value="#{colorBean.color}"
         style="background-color:#{colorBean.color}"/>
    </h:form>
  </h:body>
</html>
```

Examples/src/ajax/ColorValidator.java

```
...
@FacesValidator("colorValidator")
public class ColorValidator implements Validator {
  public void validate(FacesContext context, UIComponent comp,
      Object value) throws ValidatorException {

    if (value == null) {
      return;
    }
    List<String> colorsList =
      Arrays.asList((new ColorBean()).getColors());
    if (!colorsList.contains(value.toString())) {
      FacesMessage message;
      message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        "Invalid color. ",
        "The provided color is not valid. ");
      throw new ValidatorException(message);
    }
  }
}
```

Try It:

Visit ***http://localhost:8080/Ajax/validate.faces***. Type a non-color in the text box to see the validation error.

THE ONEVENT ATTRIBUTE

- JSF can invoke client-side JavaScript to monitor an Ajax event's progress.

- Define a function that takes a **data** object as a parameter and register it with the **onevent** attribute.

```
function callback(data) {
  if (data.status == "begin") { ... }
  ...
}
```

```
<h:inputText id="input" value="#{bean.property}">
 <f:ajax event="keyup" onevent="callback"/>
</h:inputText>
```

  - The function is called three times for each successful Ajax request.

  - Query the **data.status** property to determine what stage the Ajax request is in:

    - **begin** – The function was called immediately before the request.
    - **complete** – The function was called immediately after the request completed. For unsuccessful requests, it is called before an error handler function is called.
    - **success** – The function was called immediately after the response.

  - The **data.source** property contains the element that initiated the event.

  - Use the **data.response** property to access the Ajax response as XML.

    - Similarly, the **data.responseText** contains the response as text.

  - Retrieve the Ajax request's response code with **data.responseCode**.

    - **responseXML**, **responseText**, and **responseCode** are all undefined in the begin phase.

Examples/WebContent/echo.xhtml

```
...
  <h:head>
     <h:outputScript library="js" name="eventListener.js"/>
     <title>Echo</title>
  </h:head>
  <h:body>
    <h2>Echo</h2>
    <h:form id="form">
      <h:panelGrid columns="3">
        Enter Text:
         <h:inputText id="colorInput" value="#{textBean.text}">
           <f:ajax event="keyup" execute="@this" render="output"
             onevent="eventCallback"/>
        </h:inputText>
         <h:outputText id="jsEventOutput" value=""/>
      </h:panelGrid>
       <h:outputText id="output" value="#{textBean.text}"/>
      <br/>
    </h:form>
  </h:body>
</html>
```

Use the **<h:outputScript>** element to reference an external JavaScript file.

Examples/WebContent/resources/js/eventListener.js

```
function eventCallback(data) {
   var eventOutput = document.getElementById("form:jsEventOutput");
  if (data.status == "begin") {
    eventOutput.innerHTML = "Please wait . . .";
  }
  else if (data.status == "complete") {
    eventOutput.innerHTML = "Ajax Request Complete";
  }
  else if (data.status == "success") {
    eventOutput.innerHTML = "Done";
  }
}
...
```

Use the **getElementById()** function to recursively search the DOM tree for an element that matches the given id.

Try It:

Visit ***http://localhost:8080/Ajax/echo.faces***. Type a few characters into the input box to see the event status display next to it. The bean is coded to sleep for 250 milliseconds per Ajax request so you can see the text display.

THE ONERROR ATTRIBUTE

- You can be notified of an Ajax error by registering a JavaScript function with the **onerror** attribute.

```
<h:inputText id="input" value="#{bean.property}">
 <f:ajax event="keyup" onerror="errorFunction"/>
</h:inputText>
```

- Define an error function that takes a **data** object as a parameter, just like the event function.

```
function errorFunction(data) {
  ...
}
```

  - The **data.status** property has different possible values for **onerror** than **onevent**:

    - **httpError** – The response status is null, undefined, or not in the 200 range.
    - **emptyResponse** – The server did not provide a response.
    - **malformedXML** – The response returned XML that is not well-formed.
    - **serverError** – The response contains error information from the server.

  - The **data** object contains the same properties for **onerror** as **onevent** with three additional properties: **data.description**, **data.errorName**, and **data.errorMessage**.

Examples/WebContent/echo2.xhtml

```
...
  <h:body>
    <h2>Echo</h2>
    <h:form id="form">
      <h:panelGrid columns="4">
        Enter Text:
         <h:inputText id="colorInput" value="#{textBean.text}">
           <f:ajax event="keyup" execute="@this" render="output"
             onevent="eventCallback" onerror="errorCallback"/>
        </h:inputText>
         <h:outputText id="jsEventOutput" value=""/>
         <h:outputText id="jsErrorOutput" styleClass="error" value=""/>
      </h:panelGrid>
       <h:outputText id="output" value="#{textBean.text}"/>
      <br/>
    </h:form>
  </h:body>
</html>
```

Examples/WebContent/resources/js/eventListener.js

```
...
function errorCallback(data) {
   var errorOutput = document.getElementById("form:jsErrorOutput");
   errorOutput.innerHTML = data.status;
}
```

Examples/src/ajax/TextBean.java

```
...
@ManagedBean(name="textBean")
public class TextBean {
  ...
  public void setText(String text) {
    ...
    if (text.equals("error")) {
      FacesContext context = FacesContext.getCurrentInstance();
      ExternalContext extContext = context.getExternalContext();
      extContext.setResponseStatus(500);
    }
  }
  ...
}
```

Force an error to test the **onerrror** callback function.

Try It:

Visit ***http://localhost:8080/Ajax/echo2.faces***. Type the word "error" into the input box to see the error status display next to the text box.

The listener Attribute

- **onEvent** and **onError** trigger client-side JavaScript function calls.

- To invoke a server-side method, use the **<f:ajax> listener** attribute.

```
<f:ajax event="keyup" listener="#{bean.methodName}"/>
```

  - The value of the attribute should be an EL method expression that identifies a method in your managed bean implementation.

  - The method's signature should be:

```
public void methodName(javax.faces.event.AjaxBehaviorEvent)
   throws javax.faces.event.AbortProcessingException {}
```

  - The method is called once for each Ajax request during the Invoke Application phase of the lifecyle.

Examples/WebContent/echo3.xhtml

```
...
  <h:body>
    <h2>Echo</h2>
    <h:form id="form" prependId="false">
      <h:panelGrid columns="4">
        Enter Text:
        <h:inputText id="colorInput" value="#{textBean.text}">
          <f:ajax event="keyup" execute="@this" render="output"
            onevent="eventCallback" onerror="errorCallback"
            listener="#{textBean.ajaxListener}"/>
        </h:inputText>
        <h:outputText id="jsEventOutput" value=""/>
        <h:outputText id="jsErrorOutput" styleClass="error" value=""/>
      </h:panelGrid>
      <h:outputText id="output" value="#{textBean.text}"/>
      <br/>
    </h:form>
  </h:body>
</html>
```

Identify the listener method by name using JSF EL.

Examples/src/ajax/TextBean.java

```
...
@ManagedBean(name="textBean")
public class TextBean {
  ...
  public void ajaxListener(AjaxBehaviorEvent event)
    throws AbortProcessingException {
    System.out.println("ajaxListener called");
  }
}
```

Try It:

Visit ***http://localhost:8080/Ajax/echo3.faces***. As you type each character in the text box, watch for messages in the Tomcat console.

Labs

❶ Study the calculator application in the *StarterCode* directory. Modify *WebContent/calculator.xhtml* so that each command button triggers an Ajax request. Modify *src/calculator/jsf/MathBean.java* so that the bean instance is stored in the view scope.
(Solutions: *Solutions/WebContent/ajaxCalculator.xhtml, Solutions/src/calculator/jsf/MathBean.java*)

❷ Modify your solution to ❶ so that both **<h:message>** tags are displayed for Ajax requests that result in a conversion or validation error.
(Solution: *Solutions/WebContent/ajaxCalculator2.xhtml*)

❸ Modify your solution again. This time only use one **<f:ajax>** tag for all four command buttons. Don't forget to display conversion and validation messages.
(Solution: *Solutions/WebContent/ajaxCalculator3.xhtml*)

❹ How many Ajax requests are invoked when adding two numbers in lab ❸? Create a listener that prints out information about each request to the Tomcat console. Use the passed in event object to retrieve the component that triggered the Ajax event.
(Solutions: *Solutions/WebContent/ajaxCalculator4.xhtml*, *Solutions/src/calculator/jsf/MathBean2.java*)

❺ (Optional) Study the library book application in the *StarterCodeLab5* directory. Modify *WebContent/bookSearch.xhtml* so that when you click on a radio button it does not submit a full HTTP request/response. Instead, it should use Ajax to execute and render only the components necessary to enable/disable the proper part of the screen. Don't forget to add id's to all of the components that will participate in the Ajax request and response.
(Solution: *SolutionsLab5/WebContent/ajaxBookSearch.xhtml*)

Lab ❺ retrieves data from a database. Make sure that the database is running before accessing any Facelets. To start the Derby database, run *startNetworkServer.bat*, which can be found under *%DERBY_HOME%\bin.*