JavaServer Faces – Content

Preface

Let's start from revisiting essentials of Web pages technologies and techniques, HTML and XHTML, CSS and JavaScript.

The textbook provides high level overview of those subjects, see pp. 305-314.

Summarizing what we just learned about **HTML** and **XHTML**, here are some important points:

- The reason for XHTML to be developed was convoluted browser specific tags. For example, web pages coded in HTML may appear different in different browsers.
- Both HTML and XHTML languages in which web pages are written. They have many similarities and many differences.
- HTML and XHTML are both markup languages that are used for developing Web pages.
- HTML files have an extension .html or .htm.
- XHTML file have an extension .xhtml, .xht, .xml, .html, .htm .
- HTML is SGML based while XHTML is XML based.
- XHTML was derived from HTML to conform to XML standards. Hence XHTML is strict when compared to HTML and does not allow user to get away with lapses in coding and structure.
- HTML documents are composed of elements that have three components- a pair of element tags – start tag, end tag; element attributes given within tags and actual, textual and graphic content. HTML element is everything that lies between and including tags. (Tag is a keyword which is enclosed within angle brackets).
- XHTML documents has only one root element. All elements including variables must be in lower case, and values assigned must be surrounded by quotation marks, closed and nested for being recognized. This is a mandatory requirement in XHTML unlike HTML where it is optional. The declaration of DOCTYPE would determine rules for documents to follow.
- Aside from the different opening declarations for a document, the differences between an HTML 4.01 and XHTML 1.0 document—in each of the corresponding DTDs—are largely syntactic.
    - The underlying syntax of HTML allows many shortcuts that XHTML does not, such as elements with optional opening or closing tags, and even EMPTY elements which must not have an end tag.
    - By contrast, XHTML requires all elements to have an opening tag or a closing tag. XHTML, however, also introduces a new shortcut: an XHTML tag may be opened and closed within the same tag, by including a slash before the end of the tag like this: `<br/>`. The introduction of this shorthand, which is not used in the SGML declaration for HTML 4.01, may confuse earlier software unfamiliar with this new convention. A fix for this is to include a space before closing the tag, as such: `<br />`.

Cascading Style Sheets (CSS) help to define how to display HTML elements. As a presentation language, it was created to give content style and appearance. While HTML determines the structure and meaning of content on a web page, CSS determines the style and appearance of this

content. The two languages are independent of one another. CSS should not reside within an HTML document and vice versa.

JavaScript enables to provide dynamic content for client-side HTML based documents.

Introduction to history and architecture of JSF

In 2004 Sun Microsystems introduced a JavaServer Faces web framework in an effort to help simplify web application development. It is an evolution of the JavaServer Pages (JSP) framework adding a more organized development life cycle and the ability to more easily utilize modern web technologies.

Please study:

- Oracle Java EE 7 Tutorial, Chapter 7, JSF, sections 7.1 and 7.2.

- Textbook, Chapter 10, Understanding JSF and JSF Specification Overview, pp. 314 – 321.

JSF is using XML files for view construction and Java classes for application logic. It adheres to the MVC architecture.

JSF is request-driven, and each request is processed by a special servlet named the **FacesServlet**. This servlet builds the component trees, processing events, determining which view to process next, and rendering the response. JSF 1.x used a special resource file named the **faces-config.xml** for specifying application details such as navigation rules, registering listeners, etc. While we can still use this file with JSF 2.x, it is advisable to utilize annotations in place of XML where possible.

Here is a timeline of JSF releases:

JSF 1.0	Nov 2004	Introduced new concepts like stateless views, page flow and the ability to create portable resource contracts.
JSF 1.1	May 2004	Bug fix release. No specification changes.
JSF 1.2	Nov 2006	Many improvements to core systems and APIs. Coincides with Java EE 5. Initial adoption into Java EE.
JSF 2.0	Jul 2009	Major release for ease of use, enhanced functionality, and performance. Coincides with Java EE 6.
JSF 2.1	Nov 2010	Maintenance release 2 of JSF 2.0. Only very minor amount of specification changes.
JSF 2.2	May 2013	Introduced new concepts like stateless views, page flow and the ability to create portable resource contracts.

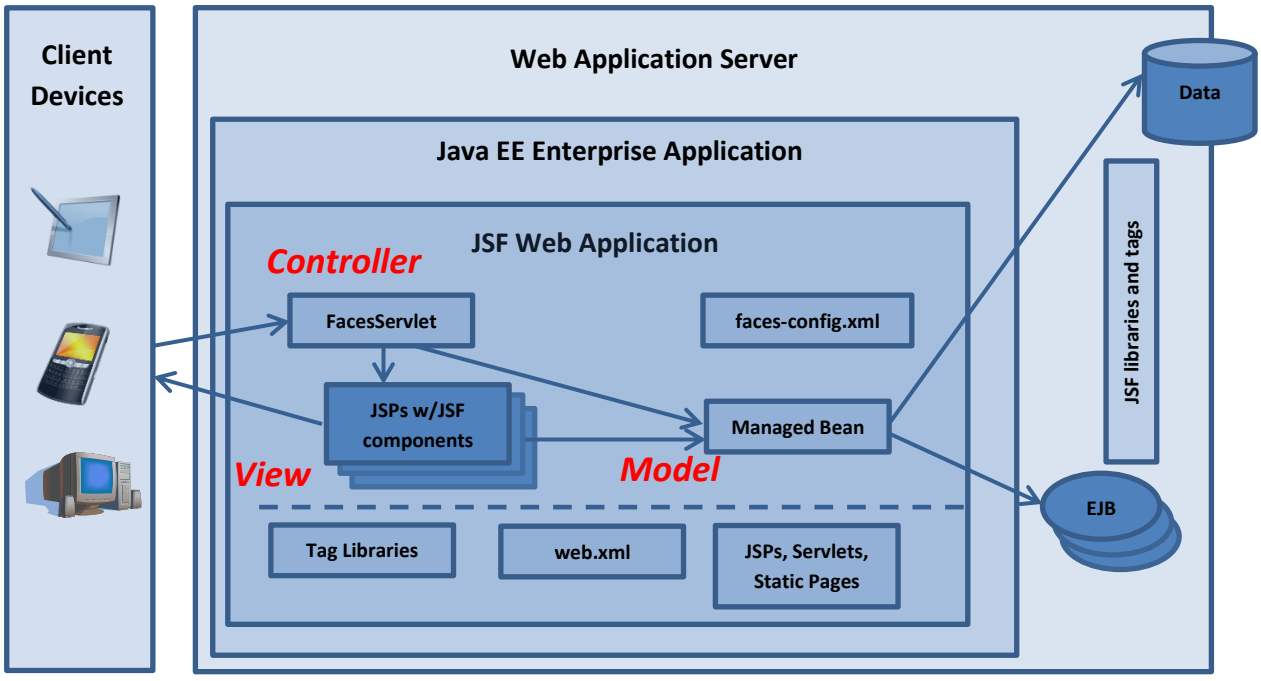
JSF technology is a framework for designing, developing and running server side User Interface Components and using them in a web application. It is very similar to Java Swing libraries that support Java based clients. While Java Swing framework is a toolkit to facilitate building Java client applications, the main purpose of JSF based applications is to help developing applications using Swing-like constructs like events, listeners, other Java components, Java Beans, etc. Additionally JSF provides UI widgets, like buttons, hyperlinks, check boxes, text fields, etc.), as well as an ability to plug-in third party components.

JSF technology architecture is based on the Model View Controller (MVC) design pattern for separating logic from presentation. Let's review MVC design pattern basics.

MVC design pattern structures an application using three separate modules:

Module	Description
Model	Carries Data and login
View	Shows User Interface
Controller	Handles processing of an application.

MVC design pattern facilitates "a separation of concerns" principle. It allows separating model and presentation to enable developers to set focus on their core skills and collaborate more clearly. Web Designers have to concentrate only on view layer rather than model and controller layer. Developers can change the code for model and typically need not to change view layer. Controllers are used to process user actions. In this process layer model and views may be changed.



In this architecture, we can find certain components that exhibit very critical behavior, e.g.:

- **FacesServlet** - a servlet that manages the request processing lifecycle for web applications that are utilizing JavaServer Faces to construct the user interface. **faces-config.xml** is an optional descriptor/configuration file. This servlet play a role of ***Controller*** in MVC pattern.

- **JavaServer Pages, JSF components, static pages, servlets –** those are Java components responsible for providing an output that will be sent back to a client. Latest JSF release 2.x recommends using **Facelets** here**.** Those components play role of ***View*** in MVC pattern.
- **Managed (Backing) Bean -** this bean is a ***Model*** of MVC pattern. It provides business logic and can also control navigation between pages.

It is very important to understand that JSF is back-end technology. It means that a JSF page has to be rendered in HTML on the server as a result of executing the component tree supported by variety of tag libraries, both in-house and third-party libraries. As a result the HTML page has a rich life cycle that handles many different phases.

So, let's start learning about JSF pages and components.

JSF Pages and Components

Please study:

- Oracle Java EE 7 Tutorial, Chapter 7, JSF (sections 7.3, 7.4), and Chapter 10, Using JSF Technology in Web Pages.

- Textbook, Chapter 10, Anatomy of JSF Pages and Anatomy of JSF Components, pp. 321 – 340.

WebLogic Server support of JSF

WebLogic Server 12.2.1.2 supports the JSF 2.2 specification (see

https://docs.oracle.com/middleware/1221/wls/NOTES/whatsnew.htm)

If you selected to install the server examples, you will find these JSF code examples:

"Incorporating AJAX in Web Applications,"

"Using Facelets and Templating," and

"Creating Bookmarkable Web Applications,"

in the *WL_HOME*`\samples\server\examples\src\examples\javaee6\jsf` directory of your WebLogic Server distribution, where *WL_HOME* is the top-level directory in which WebLogic Server is installed, for example, `c:\Oracle\wls12.2.1.2`.

In the next section I would like to show you an example of very simple JSF application. We will go over its components and flow of execution, and will also address supplying configuration files, packaging and deployment on the WLS.

An example of a simple JSF web application

Let's step through a process of creating a simple JSF web application that includes:

- a single XHTML page;
- a single JSF managed bean;
- required configuration files.

Let's assume that this application will be able to display a message produced by a JSF managed bean.

We will observer development, packaging, deployment and execution steps using WebLogic Server.

Before starting developing this application, I suggest creating development environment with the following directory structure:

+ **myFirstJSF**
  |
  +src (Java source code is kept here)
  |
  +web (run-time JSF application directory)
    |
    +WEB-INF
      |
      +classes (compiled Java classes)

Step 1. Displaying a JSF managed bean field value.

The following code creates XHTML page (a view) that will be used to display the JSF managed bean field value. Save it as **display.xhtml** file in **./myFirstJSF/web** directory.

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

    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>A Simple JSF Application</title>
    </h:head>

    <h:body>
        <p>
            This JSF app uses a request-scoped JSF managed bean to display the message
            below. In order to change the message, you need to modify it in the managed
            bean's constructor and then recompile the application, redeploy it and run it
            again.
```

```
                <br/>
                <br/>
                        #{helloWorldController.hello}
                <br/>
                        or
                <br/>
                        <h:outputText id="helloMessage" value="#{helloWorldController.hello}"/>
                </p>
        </h:body>

</html>
```

This code above is an example of a simple Facelets XHTML page. Although we will study Facelets in details letter in this module, let's analyze certain tags in this code.

First off, notice that it uses JSF tags (xmlns:h="http://java.sun.com/jsf/html) in order to display HTML "head" , "body" and outputText" elements. Alias "h" is used in the body of the page to display a specific component that belongs to the tag library (for example, <h:outputText>, etc.). In this respect, JSF is "reusing" previously existed JSP technologies features, such as JSP Standard Tag Library. However, JSF exhibits more complex life cycle than JSP, with the component tree generation and the rendering to be executed at different phases. This is very critical difference between JSP and JSF!

Second, this code uses a JSF expression #{helloWorldController.hello}, that references a JSF managed bean and the field to expose.

Note: pay attention on use of proper XML namespaces:

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

Those XML namespaces identify to WebLogic Server which version of JSF specification is used to produce a given XHTML page.

Step 2. Developing the JSF managed bean.

The following code is that of HelloWorldController, the JSF managed bean. Save it as **HelloWorldController.java** file in **./myFirstJSF/src/edu/jhu/jee/Felikson/leonid** directory.

```
package edu.jhu.jee.felikson.leonid;

                                                World";
import java.io.Serializable;
import javax.inject.Named;                                  }
import javax.enterprise.context.RequestScoped;
import javax.faces.bean.ManagedBean;

@Named(value = "helloWorldController")
@RequestScoped
@ManagedBean


                                                }
```

```
public class HelloWorldController implements Serializable {

    private String hello;

    public HelloWorldController() {
        System.out.println ("Instantiated helloWorldController");
        hello = "Hello World!";
    }

    public String getHello() {
        return hello;
    }

    public void setHello(String hello) {
        this.hello = hello;
    }
}
```

The main purpose of JSF managed (backing) Java bean is to synchronize values with components, to process business logic and to handle navigation between pages.

In this case, binding of the "hello" attribute of the helloWorldController bean is done by the following code:

```
#{helloWorldController.hello}
```

This code is an example of EL, Expression Language, used by JSF.

Step 3. Developing web.xml configuration file.

The following web.xml includes necessary configuration for FacesServlet. Save it as **web.xml** file in **./myFirstJSF/web/WEB-INF** directory.

```
<?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_3_0.xsd"
         version="3.0">

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

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

```
    </servlet-mapping>

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

</web-app>
```

Note: pay attention on use of proper XML namespaces and version 3.0 of web application specification (and Servlet API version):

```
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_3_0.xsd"
      version="3.0">
```

In JSF 2.0 development, it's recommended to set the "**javax.faces.PROJECT_STAGE**" to "**Development**", it will provide many useful debugging information to let you track the bugs easily. For deployment, just change it to "**Production**".

Note: Since JSF 2.1 specification it is allowed to package JSF application and then to deploy it without having web.xml configuration file in it.

Step 4. Compiling JSF managed bean

Note: Before working with any WebLogic based web application, we have to setup development environment variables, specifically CLASSPATH and PATH. For that,

In order to compile `HelloWorldController` bean, in DOS command shell go to `myFirstJSF/src` directory and execute the following command:

javac -d ../web/WEB-INF/classes ./edu/jhu/jee/felikson/leonid/*.java

As a result of successful compilation, `HelloWorldController.class` file will be saved in WEB-INF/classes directory, in subdirectories according to its respective package name.

Step 5. Packaging and deploying on the WLS instance.

Note: In order to deploy a web application on the WLS, we need to have an instance of WLS up and running. For that, in DOS command shell, please:

a) go to `C:\Oracle\wls12.2.1.2\user_projects\domains\lfDomain` directory (in your environment, preplace "`lfDomain`" with whatever WLS domain name you chosen when creating the WLS domain).

b) Run "`startweblogic`" script and observe outpur message as the WLS starts.

Now we have 2 options of how to deploy `myFirst` JSF application on the WebLogic Server as follows.

1) We can create "war" file by going to `myFirstJSF\web` directory and executing:

jar cfv web.war *

Then we should open up a web browser window, go to the WLS Admin console at http://localhost:7001/console , and then deploying `web.war` file using Admin console.

2) We can deploy `myFirst` application in exploded directory format ("as-is") by deploying it using the WLS Admin console, starting from `web` directory.

Step 6. Running JSF application.

In order to execute `myFirst` JSF application, open up a new web browser window and type the following URL:

http://localhost:7001/web/

or

http://localhost:7001/web/faces/display.xhtml

Due to a default page defined in web.xml as `display.xhtml`, you will get the same result on the browser window.

Note: This example demonstrates how closely related JSF and JSP technologies are. Indeed, the only difference in the two view pages is the use of JSP expressions #{} rather than the standard JSP value expression ${}.

The Lifecycle of the JSF Application

Please study:

- Oracle Java EE 7 Tutorial, Chapter 7, JSF, sections 7.6 and 7.7.

- Textbook, Chapter 10, Life Cycle, pp. 325 – 326.

So, after reading this material, you now have knowledge about phases of lifecycle for JSF applications. Every web application has a lifecycle. Such common tasks, as handling incoming requests, decoding parameters, modifying and saving state, and rendering web pages to the browser, are all performed during different phases of a web application lifecycle. Why do we care? It is because we can gain additional flexibility in designing and developing those applications. Although it is not necessary to understand the lifecycle of a JSF application, the information can be useful for creating more complex applications.

Some web application frameworks hide the details of the lifecycle from developers, whereas others require managing lifecycle phases manually. By default, JSF automatically handles most of the lifecycle actions. However, it also exposes the various stages of the request lifecycle, so that we can modify or perform different actions if an application requirements warrant it.

- The lifecycle of a JSF application starts and ends with the client making a request for the web page, and the server responds with the page.
- The lifecycle consists of two main phases: **execute** and **render**.
- During the execute phase, several actions can take place:

    - The application view is built or restored.
    - The request parameter values are applied.
    - Conversions and validations are performed for component values.
    - Managed beans are updated with component values.
    - Application logic is invoked.
- For a first, initial request, only the view is built. For subsequent, postback requests, some or all of the other actions can take place.
- In the render phase, the requested view is rendered as a response to the client. Rendering is typically the process of generating output, such as HTML or XHTML, that can be read by the client, usually a browser.

Let's look into activities that take place during lifecycle phases while executing the above example of `myFirst` JSF application when it is deployed on the WebLogic Server.

1. When the `myFirst` application is built and deployed on the WebLogic Server, the application is in an **uninitiated** state.
2. When a client makes an initial request for the `display.xhtml` web page, the `myFirst` Facelets application is compiled.
3. The compiled Facelets application is executed, and a new component tree is constructed for the `myFirst` application and is placed in a `javax.faces.context.FacesContext`.
4. The component tree is populated with the component and the managed bean property associated with it, represented by the EL expression `helloWorldController.hello`.
5. A new view is built, based on the component tree.
6. The view is rendered to the requesting client as a response.
7. The component tree is destroyed automatically.
8. On subsequent, postback requests, the component tree is rebuilt, and the saved state is applied.

A few important points.

1. Every JSF managed bean has to be developed as a JavaBean, following all the Java Beans specification requirements. In particular, all managed beans should be specified as serializable so that they can be persisted to disk by the container if necessary.
2. JSF managed bean is responsible for providing the application logic for a JSF-based web application. It is basically the backbone of a JSF view.
3. Usually, there is one JSF managed bean per each JSF view (that's why it's called a *backing* bean).
4. By default, a JSF managed bean can be referred to within a JSF view using the name of the bean with a lowercase first letter. However, using @ManagedBean annotation, the string that is used to reference the bean from within a view can be changed.
5. Properties of JSF managed bean are available for use within JSF views by utilizing JSF EL expressions, contained within the #{ and } character sequences and that are readable and writable.
6. The expression syntax used to access the bean's method is #{*beanName.methodName*}.

Facelets

Facelets is the *view declaration language* for JSF. It can be considered as the replacement for JSP.

What is the motivation for using facelets?

- It is very common to have multiple pages that share the same basic layout and same general look.
- So, similarly to how following object oriented paradigm enables good code reuse, using facelets allows voiding code repetition in facelets.

Please study:

- Oracle Java EE 7 Tutorial, Chapter 8, Facelets, Chapter 9, Expression Language, and Chapter 12, Developing with JSF Technology.

Summarizing about Facelets, I remember reading somewhere the following:

"If you're not using Facelets, you're not getting the most you can out of JSF."

That is very true, in my opinion.

- The real power of Facelets is in promoting and supporting a new way of thinking about JSF.
- In JSF 2.0, Facelets is the default view declaration language (VDL) replacing JavaServer Pages (JSP).
- Facelets is a JSF-centric view technology.
- What differentiates Facelets and JSPs?
    - JavaServer Pages is a templating language that produces a servlet. The body of the JSP becomes the equivalent of a servlet's doGet() and doPost() methods (that is, it becomes the jspService() method).
    - The JSF custom tags (such as f:view and h:form) are just calls to the JSF components to render themselves in their current state.
    - The life cycle of the JSF component model is independent from the life cycle of the JSP-produced servlet.
    - That independence is where the confusion comes in.
    - Unlike JSP, Facelets is a templating language built from the ground up with the JSF component life cycle in mind. With Facelets, we produce templates that build a component tree, not a servlet. This allows for greater reuse because we can compose components out of a composition of other components.
    - Facelets obviates the need to write custom tags to use JSF components. Facelets uses JSF custom components natively. Very little special coding is needed to bridge JSF and Facelets: all we have to do is declare the JSF components in a Facelet tag library file. We can use JSF components directly within the Facelets templating language without any additional development.
- Facelets is based on compositions. A composition defines a JSF UIComponents structure in a Facelets page. A Facelets application may consist of compositions defined in different Facelets pages and run as an application.
- JSF Validators and Converters may be added to Facelets. Facelets provides a complete expression language (EL) and JavaServer Pages Standard Tag Library (JSTL) support. Templating, re-use, and ease of development are some of the advantages of using Facelets in a web application. We will take a look into a few examples of Facelets later in this Module.

An example of another JSF web application

Let's step through a process of creating a JSF web application that will be very easy to get up and running.

For this purpose, I am going to use *hello1* application discussed in Oracle's Java EE 7 Tutorial, section 7.3.

This application comprises of:

- two XHTML pages with component tags;
- a single JSF managed bean;
- required JSF configuration files.

Let's assume that this application will be able to display a message produced by a JSF managed bean.

We will observer development, packaging, deployment and execution steps using WebLogic Server.

Step 1. Two xhtml JSF pages, `index.xhtml` and `response.xhtml`.

Here is `index.xhtml` code below:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"

      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html" >

<h:head>

    <title>Facelets Hello Greeting</title>

</h:head>
```

```
<h:body>
    <h:form>
        <h:graphicImage url="#{resource['images:duke.waving.gif']}"  alt="Duke waving his
        hand"/>
            <h2>Hello, my name is Duke. What's yours?</h2>
            <h:inputText id="username" title="My name is: " value="#{hello.name}"
                required="true" requiredMessage="Error: A name is required." maxlength="25" />
            <p></p>
            <h:commandButton id="submit" value="Submit" action="response">
            </h:commandButton>
            <h:commandButton id="reset" value="Reset" type="reset">
            </h:commandButton>
        </h:form>
        <div class="messagecolor">
            <h:messages showSummary="true" showDetail="false" errorStyle="color: #d20005"
                infoStyle="color: blue"/>
        </div>
    </h:body>
</html>
```

Here is `response.xhtml` code below.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"

      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html" >

  <h:head>

    <title>Facelets Hello Response</title>

  </h:head>

  <h:body>

    <h:form>

      <h:graphicImage url="#{resource['images:duke.waving.gif']}" alt="Duke waving his
      hand"/>

        <h2>Hello, #{hello.name}!</h2>

        <p></p>

        <h:commandButton id="back" value="Back" action="index" />

     </h:form>

  </h:body>

</html>
```

Step 2. Developing `hello` JSF managed bean.

```
package javaeetutorial.hello1;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.faces.bean.ManagedBean;

@RequestScoped
@ManagedBean (name="hello")
```

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

Step 3. Now you can compile `hello` bean, package hello application into war file format, deploy it on the WLS and execute it by using URL **http://localhost:7001/hello1/ or http://localhost:7001/hello1/faces/index.xhtml**

One more complex JSP application

The following JSF application demonstrates another use of JSF view with the managed bean that provides business logic. The application calculates two numbers that are entered by the user and then adds, subtracts, multiplies or divides them depending upon the user's selection of the operation. The managed bean is responsible for defining fields for each of the numbers that will be entered by the user, as well as a field for the result of the calculation. The managed bean is also responsible for creating a list of `Strings` that will be displayed within an `h:selectOneMenu` element within the JSF view and retaining the value that is chosen by the user.

The code below is a Java bean that will be for calculation purposes.

```
package edu.jhu.jee.felikson.leonid;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.application.FacesMessage;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.faces.model.SelectItem;

@SessionScoped
@ManagedBean(name="calculationController")
public class CalculationController implements Serializable {

   private int num1;
   private int num2;
   private int result;
   private String calculationType;
   private static String ADDITION = "Addition";
   private static String SUBTRACTION = "Subtraction";
```

```
private static String MULTIPLICATION = "Multiplication";
private static String DIVISION = "Division";
List<SelectItem> calculationList;

public CalculationController() {
    // Initialize variables
    num1 = 0;
    num2 = 0;
    result = 0;
    calculationType = null;
    // Initialize the list of values for the SelectOneMenu
    populateCalculationList();
    System.out.println("initialized the bean!");
}

public int getNum1() {
    return num1;
}

public void setNum1(int num1) {
    this.num1 = num1;
}

public int getNum2() {
    return num2;
}

public void setNum2(int num2) {
    this.num2 = num2;
}

public int getResult() {
    return result;
}

public void setResult(int result) {
    this.result = result;
}

public String getCalculationType() {
    return calculationType;
}

public void setCalculationType(String calculationType) {
    this.calculationType = calculationType;
}

public List<SelectItem> getCalculationList(){
    return calculationList;
}
```

```
    private void populateCalculationList(){
        calculationList = new ArrayList<SelectItem>();
        calculationList.add(new SelectItem(ADDITION));
        calculationList.add(new SelectItem(SUBTRACTION));
        calculationList.add(new SelectItem(MULTIPLICATION));
        calculationList.add(new SelectItem(DIVISION));
    }

    public void performCalculation() {
        if (getCalculationType().equals(ADDITION)){
            setResult(num1 + num2);
        } else if (getCalculationType().equals(SUBTRACTION)){
            setResult(num1 - num2);
        } else if (getCalculationType().equals(MULTIPLICATION)){
            setResult(num1 * num2);
        } else if (getCalculationType().equals(DIVISION)){
            try{
                setResult(num1 / num2);
            } catch (Exception ex){
                FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR, "Invalid
Calculation", "Invalid Calculation");
                FacesContext.getCurrentInstance().addMessage(null, facesMsg);
            }
        }
    }
}
```

The following JSF view contains JSF components that are displayed as text boxes into which the user can enter input data, a pick-list of different calculation types for user to choose from, a component responsible for displaying the result of calculation, and an **h:commandButton** component for submitting the form values.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

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

    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>My second JSF app</title>
    </h:head>
    <h:body>
        <f:view>

            <h2>Perform a Calculation</h2>
            <p>
```

```
            Use the following form to perform a calculation on two numbers.
            <br/>
            Enter
            the numbers in the two text fields below, and select a calculation to
            <br/>
            perform, then hit the "Calculate" button.
            <br/>
            <br/>
            <h:messages errorStyle="color: red" infoStyle="color: green" globalOnly="true"/>
            <br/>
            <h:form id="calulationForm">
               Number1:
               <h:inputText id="num1" value="#{calculationController.num1}"/>
               <br/>
               Number2:
               <h:inputText id="num2" value="#{calculationController.num2}"/>
               <br/>
               <br/>
               Calculation Type:
               <h:selectOneMenu id="calculationType" value="#{calculationController.calculationType}">
                  <f:selectItems value="#{calculationController.calculationList}"/>
               </h:selectOneMenu>
               <br/>
               <br/>
               Result:
               <h:outputText id="result" value="#{calculationController.result}"/>
               <br/>
               <br/>
               <h:commandButton action="#{calculationController.performCalculation()}"
value="Calculate"/>
            </h:form>
         </p>
      </f:view>
   </h:body>
</html>
```

After compiling JSF managed bean, packaging and deploying this application on the Weblogic Server and invoking it from by http://localhost:7001/web2/faces/calc.xhtml , we can observe its results on the web browser.

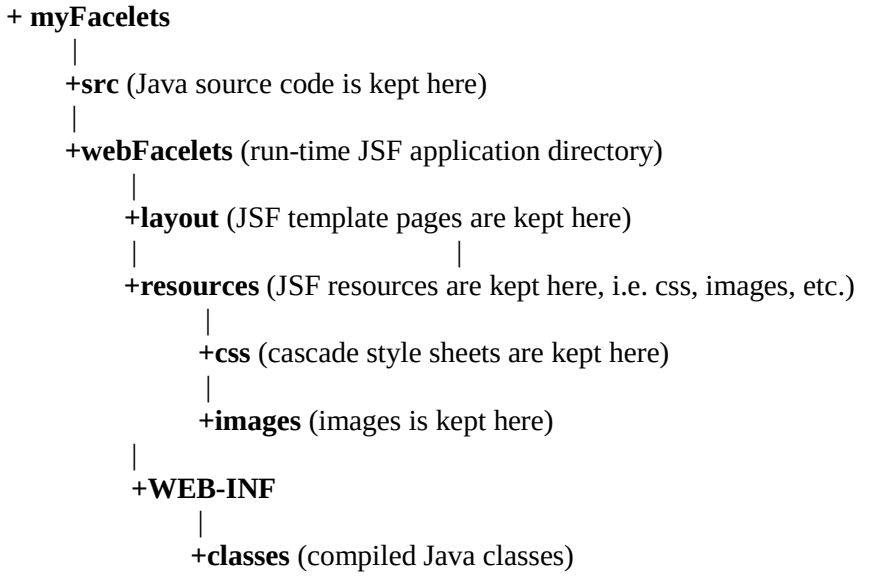
How to create a page template

At the beginning of the Web age, while working on first generation of web applications, developers have been creating web pages that consisted of many HTML tables supporting structuring layout and lots of redundancy across application web pages. It was a very tedious, time consuming and ineffective process. Later, with use of CSS technologies it became easier to add more style and better organization to web pages. Also, CSS allowed developers to separate style from the markup, which in turn reduced overlap and redundancy and added more consistency.

With developing Facelets technologies, we now have a **view definition language** that helps organizing JSF views. One of the main features of Facelets is an ability to develop page templates that makes it easier to develop web pages with similar structure.

The following JSF application demonstrates creation and use of a page template. The example is adopted from "Java EE 7 Recipes" book written by Josh Juneau (ISBN 978-1-4302-4425-7) and adjusted and tuned for use with WebLogic Server 12c.

Step 1. First off, before starting developing this application, I suggest creating development environment with the following directory structure:



Step 2. Now, we have to create a page template using Facelets view definition language.

The template demonstrates defining the standard layout for all web pages for the bookstore web site, displaying number of a number of book titles on the left side of the screen, a header at the top, a footer at the bottom, and a main view in the middle. When a book title is clicked in the left menu, the middle view changes displaying the list of authors for the selected book.

In order to create a template, we will first develop a new XHTML view file; then we will add the appropriate HTML/JSF/XML mark up to it. Content from other views will displace the `ui:insert` elements in the template once the template has been applied to one or moer JSF views. The following template called `custom_template.xhtml` is saved in `layout` sub-directory and its code is shown below.

```
<?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"
      xmlns:h="http://java.sun.com/jsf/html">

   <h:head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <h:outputStylesheet library="css" name="default.css"/>
      <h:outputStylesheet library="css" name="cssLayout.css"/>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{faceletsAuthorController.storeName}</title>
   </h:head>

   <h:body>

      <div id="top">
         <h2>#{faceletsAuthorController.storeName}</h2>
      </div>
      <div>
         <div id="left">
            <h:form id="navForm">
                  <h:commandLink
                  action="#{faceletsAuthorController.populateJavaRecipesAuthorList}" >Java
                  7 Recipes</h:commandLink>
               <br/>
               <br/>
                  <h:commandLink
                  action="#{faceletsAuthorController.populateJavaEERecipesAuthorList}">Ja
                  va EE 7 Recipes </h:commandLink>
            </h:form>
         </div>
         <div id="content" class="left_content">
            <ui:insert name="content">Content</ui:insert>
         </div>
      </div>
      <div id="bottom" style="position: absolute;width: 100%;bottom: 20px;">
         Written by Josh Juneau, Apress Author
      </div>

   </h:body>

</html>
```

This template defines the overall structure for the application views. It uses CSS style sheets, that are saved in `resources/css` sub-directory, to declare the formatting for each of the elements within the template as follows.

Here is `cssLayout.css` code.

```
#top {
   width: 100%;
   height: 100px;
   background-color: #036fab;
   color: white;
   padding: 5px;
   margin-bottom: 10px;
}

#content{
        width: 100%;
        height:90%;
        margin-right:16px;
        margin-bottom:16px;

}                                World";

#bottom {                                }
   width: 100%;
   background-color: #c2dfef;
   padding: 5px;
   margin: 10px 0px 0px 0px;
}

#left {                                  }
   float: left;
   background-color: #ece3a5;
   padding: 5px;
   width: 150px;
}

#right {
   float: right;
   background-color: #ece3a5;
   padding: 5px;
   width: 150px;
}

.center_content {
   position: relative;
```

```
    background-color: #dddddd;
    padding: 5px;
}

.left_content {
    position: relative;
    background-color: #dddddd;
    padding: 5px;
    margin-left: 170px;

}

.right_content {
    background-color: #dddddd;
    padding: 5px;
    margin: 0px 170px 0px 170px;
}

#top a:link, #top a:visited {
  color: white;
  font-weight : bold;
  text-decoration: none;
}

#top a:link:hover, #top a:visited:hover  {
  color: black;
  font-weight : bold;
  text-decoration : underline;
}
```

Here is `default.css` code below.

```
body {
    background-color: #ffffff;
    font-size: 12px;
    font-family: Verdana, "Verdana CE",  Arial, "Arial CE", "Lucida
Grande CE", lucida, "Helvetica CE", sans-serif;
    color: #000000;
    margin: 10px;
}

h1 {
    font-family: Arial, "Arial CE", "Lucida Grande CE", lucida, "Helvetica
CE", sans-serif;
    border-bottom: 1px solid #AFAFAF;
```

```
    font-size:  16px;
    font-weight: bold;
    margin: 0px;
    padding: 0px;
    color: #D20005;
}

a:link, a:visited {
  color: #045491;
  font-weight : bold;
  text-decoration: none;
}

a:link:hover, a:visited:hover  {
  color: #045491;
  font-weight : bold;
  text-decoration : underline;
}
```

Here is `styles.css` code below.

```
.authorTable{
        border-collapse:collapse;
}
.authorTableOdd{
        text-align:center;
        background:none repeat scroll 0 0 #CCFFFF;
        border-top:1px solid #BBBBBB;
}

.authorTableEven{
        text-align:center;
        background:none repeat scroll 0 0 #99CCFF;
        border-top:1px solid #BBBBBB;
}


.tocTableOdd{
    background: #c0c0c0;
}

.tocTableEven{
    background: #e0e0e0;
}
```

```
.col1{
    text-indent: 15px;
    font-weight: bold;
}
```

A few important points.

1. A Facelets template consists of HTML and JSF markup, and it is used to define a page layout. Sections of the template can specify where page content will be displayed through the usage of the `ui:insert` tag. Any areas within the template that contain a `ui:insert` tag can have content inserted into them from a template client.
2. In general, a Facelets template includes the required namespaces, followed by HTML, JSF and Facelets tags according to required layout design.
3. Be careful using correct XML namespaces. For use with WebLogic Server 12c, the following XML namespaces are essential:
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html
4. Every template must include `<html>`, `<head>` or `<h:head>` and `<body>` or `<h:body>` elements.
5. The `ui:insert` tag contains a name attribute, which is set to the name of corresponding `ui:define` element that will be included in the view.
6. As you noticed, this template uses EL expressions to reference JSF managed bean called `AuthorController`. We will introduce Java code for this bean during following discussion.

Step 3. Now we can apply a template to the views of the web application. For that, we will use `ui:composition` within each view that will utilize the template. The `ui:composition` tag should be used to invoke the template, and `ui:define` tag should be placed where content should be inserted. We will define three view, `view_a.xhtml`, `view_b.xhtml` and `view_c.xhtml`. All views have to be saved in `webFacelets` directory.

Here is the first view called `view_a.xhtml`. This view will be used to display the authors of the ***Java 7 Recipes*** book. The template is applied to the view; and individual `ui:define` tags are used to specify the content that should be inserted into the page/view.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!--
Book:  Java EE7 Recipes
Recipe: 4-1
Author: J. Juneau
-->
<!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"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    <body>

        <ui:composition template="./layout/custom_template.xhtml">

            <ui:define name="top">

            </ui:define>

            <ui:define name="left">

            </ui:define>

            <ui:define name="content">
                <h:form id="componentForm">
                    <h1>Author List for Java 7 Recipes</h1>
                    <p>
                        Below is the list of authors.  Click on the author's last name
                        for more information regarding the author.
                    </p>

                        <h:graphicImage id="javarecipes" style="width: 100px; height:
                        120px" library="image" name="java7recipes.png"/>
                    <br/>
                        <h:dataTable id="authorTable" border="1"
                        value="#{faceletsAuthorController.authorList}"
                                var="author">
                        <f:facet name="header">
                            Java 7 Recipes Authors
                        </f:facet>
                        <h:column>
                            <h:commandLink id="authorName"
                        action="#{faceletsAuthorController.displayAuthor(author.last)}"
```

```
                                value="#{author.first} #{author.last}"/>
                    </h:column>
                </h:dataTable>
                <br/>
                <br/>


            </h:form>
        </ui:define>

        <ui:define name="bottom">
            bottom
        </ui:define>

    </ui:composition>

</body>
</html>
```

Here is the second view called `view_b.xhtml`. This view will be used to display the authors of the ***Java EE 7 Recipes*** book. Again, the template is applied to the view; and individual `ui:define` tags are used to specify the content that should be inserted into the page/view.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!--
Book:  Java EE7 Recipes
Recipe: 4-2
Author: J. Juneau
-->
<!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"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    <body>

        <ui:composition template="./layout/custom_template.xhtml">

            <ui:define name="top">

            </ui:define>
```

```
        <ui:define name="left">

        </ui:define>

        <ui:define name="content">
            <h:form id="componentForm">
                <h1>Author List for Java EE 7 Recipes</h1>
                <p>
                    Below is the list of authors.  Click on the author's last name
                    for more information regarding the author.
                </p>

                <h:graphicImage id="javarecipes" library="image" style="width:
100px; height: 120px" name="java7recipes.png"/>
                <br/>
                <h:dataTable id="authorTable" border="1"
value="#{faceletsAuthorController.authorList}"
                             var="author">
                    <f:facet name="header">
                        Java 7 Recipes Authors
                    </f:facet>
                    <h:column>
                        <h:commandLink id="authorName"
action="#{faceletsAuthorController.displayAuthor(author.last)}"
                                       value="#{author.first} #{author.last}"/>
                    </h:column>
                </h:dataTable>
                <br/>
                <br/>


            </h:form>
        </ui:define>

        <ui:define name="bottom">
            bottom
        </ui:define>

    </ui:composition>

  </body>
</html>
```

Finally here is the third view called `view_c.xhtml`. This view will be used to display the individual author details. Again, the same template is applied to the view.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Book:  Java EE7 Recipes
Recipe: 4-2
Author: J. Juneau
-->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>Recipe 4-1: Facelets Page Template</title>
  </h:head>
  <h:body>
    <ui:composition template="./layout/custom_template.xhtml">

      <ui:define name="top">

      </ui:define>

      <ui:define name="left">
```

```
        </ui:define>
        <ui:define name="content">
            <h:form id="componentForm">
                <h1>#{faceletsAuthorController.current.first}
                #{faceletsAuthorController.current.last}</h1>
                <p>
                    <h:graphicImage id="java7recipes" library="image" style="width:
                    100px; height: 120px" name="java7recipes.png"/>
                    <br/>
                    #{faceletsAuthorController.current.bio}
                </p>
            </h:form>
        </ui:define>
        <ui:define name="bottom">
            bottom
        </ui:define>
    </ui:composition>
</h:body>
</html>
```

Step 4. Now we can provide Java code for AuthorController managed bean, which is responsible for providing all the business logic for the bookstore application.

```
package org.javaeerecipes.chapter04.recipe04_01;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
```

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

/**
 * Recipe 4-1
 *
 * @author juneau
 */
@ManagedBean(name = "faceletsAuthorController")
@SessionScoped

public class AuthorController implements Serializable {

   private List<Author> authorList;
   private String storeName = "Acme Bookstore";

   private String juneauBio =
         "Josh Juneau has been developing software"
         + " since the mid-1990s. PL/SQL development and database programming"
         + " was the focus of his career in the beginning, but as his skills developed,"
         + " he began to use Java and later shifted to it as a primary base for his"
         + " application development. Josh has worked with Java in the form of graphical"
         + " user interface, web, and command-line programming for several years. "
         + "During his tenure as a Java developer, he has worked with many frameworks"
         + " such as JSF, EJB, and JBoss Seam. At the same time, Josh has extended his"
         + " knowledge of the Java Virtual Machine (JVM) by learning and developing
applications"
```

```
        + " with other JVM languages such as Jython and Groovy. His interest in learning"
        + " new languages that run on the JVM led to his interest in Jython. Since 2006,"
        + " Josh has been the editor and publisher for the Jython Monthly newsletter. "
        + "In late 2008, he began a podcast dedicated to the Jython programming language.";
    private String deaBio = "This is Carl Dea's Bio";
    private String beatyBio = "This is Mark Beaty's Bio";
    private String oConnerBio = "This is John O'Connor's Bio";
    private String guimeBio = "This is Freddy Guime's Bio";
    private Author current;
    private String authorLast;
                                                    World";
                                                    }
    /**
     * Creates a new instance of RecipeController
     */
    public AuthorController() {
                                                    }
        populateJavaRecipesAuthorList();
                                                    }
    }

    public String populateJavaRecipesAuthorList() {

        authorList = null;

        authorList = new ArrayList<>();
        authorList.add(new Author("Josh", "Juneau", juneauBio));
        authorList.add(new Author("Carl", "Dea", deaBio));
```

```
    authorList.add(new Author("Mark", "Beaty", beatyBio));
    authorList.add(new Author("John", "O'Conner", oConnerBio));
    authorList.add(new Author("Freddy", "Guime", guimeBio));
    return "recipe04_01a";
}

public String populateJavaEERecipesAuthorList() {
    System.out. println("initializng authors list");
    authorList = new ArrayList<>();
    authorList.add(new Author("Josh", "Juneau", juneauBio));
    return "recipe04_01b";
                                        World";
                                                }
}

public String displayAuthor(String last) {
    for (Author author : authorList) {
        if (author.getLast().equals(last)) {          }
            current = author;                        }
        }
    }
    return "recipe04_01c";
}

/**
 * @return the authorList
 */
```

```
public List getAuthorList() {
    return authorList;
}

/**
 * @return the current
 */
public Author getCurrent() {
    return current;
}

/**
 * @param current the current to set
 */
public void setCurrent(Author current) {
    this.current = current;
}

/**
 * @return the authorLast
 */
public String getAuthorLast() {
    return authorLast;
}

/**
```

```
     * @param authorLast the authorLast to set
     */
    public void setAuthorLast(String authorLast) {
        this.authorLast = authorLast;
    }

    /**
     * @return the storeName
     */
    public String getStoreName() {
        return storeName;
    }

    /**
     * @param storeName the storeName to set
     */
    public void setStoreName(String storeName) {
        this.storeName = storeName;
    }
}
```

Here is Java code for Author Java bean (which is a simple Java bean, not JSF managed bean).

```
package org.javaeerecipes.chapter04.recipe04_01;

/**
 * Recipe 4-1
 * @author juneau
 */
public class Author implements java.io.Serializable {
    private String first;
    private String last;
    private String bio;

    public Author(){
        this.first = null;
        this.last = null;
        this.bio = null;
    }

    public Author(String first, String last, String bio){
        this.first = first;
        this.last = last;
        this.bio = bio;
    }
    /**
     * @return the first
     */
    public String getFirst() {
```

```
    return first;
}

/**
 * @param first the first to set
 */
public void setFirst(String first) {
    this.first = first;
}

/**
 * @return the last
 */
public String getLast() {
    return last;
}

/**
 * @param last the last to set
 */
public void setLast(String last) {
    this.last = last;
}

/**
 * @return the bio
```

```
     */
    public String getBio() {
        return bio;
    }

    /**
     * @param bio the bio to set
     */
    public void setBio(String bio) {
        this.bio = bio;
    }
}
```

Both AuthorController and Author source have to be saved in `src` directory.

Step 5. Make sure to compile both `Author.java` and `AuthorController.java` Java code and place cmpiled classes in WEB/INF/classes subdirectory, according to respective package structure based directory.

Step 6. You can now deploy exploded directory webFacelets on the WebLogic Server, using Admin console.

Step 7. After deploying it, you can execute your application by using the following URL in your web browser: **`http://localhost:7001/webFacelets/`**